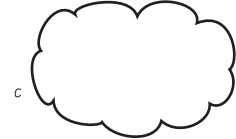
**9**

processes

and system calls



\* **Breaking** **boundaries**+

Thanks, Ted. Since you taught me how to make system calls, I haven’t looked back. Ted? Ted, are you there?

**It’s time to think outside the box.**

You’ve already seen that you can build complex applications by connecting small tools

together on the command line. But what if you want to *use other programs* from inside

your own code? In this chapter, you’ll learn how to use **system services** to create and

control ***processes***. That will give your programs access to *email*, the *Web,* and *any other tool you’ve got installed*. By the end of the chapter, you’ll have the power to go ***beyond C.***

**this is a new chapter 397**

***system()***

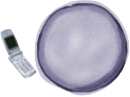
System calls are your hotline to the OS

C programs rely on the operating system for pretty much everything. They make **system calls** if they want to talk to the hardware. System calls are just functions that live

*inside* the operating system’s **kernel**. Most of the code in the C Standard Library depends on them. Whenever you call printf() to display something on the command

line, somewhere at the back of things, a system call will be made to the operating system to send the string of text to the screen.

Certainly. I shall perform those tasks immediately.



I want to display this on

the command line, then play

this music track, then send this message to the network …



Let’s look at an example of a system call. We’ll begin with one called (appropriately) **system()**.

system() takes asingle string parameter and executes it as if you had typed it on the command line:

**system("dir D:");** This will print out the contents of the D: drive. **system("gedit");**This will launch an editor on Linux.

**system("say 'End of line'");**This will read to you on the Mac.

The system() function is an easy way of running other programs from your code—particularly if you’recreating a quick prototype and you’d sooner call external programs rather than write lotsandlots of C code.

**398 *Chapter 9***

***processes and system calls***



Code Magnets

This is a program that writes timestamped text to the end of a logfile. It would have been perfectly possible to write this entire program in C, but the programmer has used a call to system()as a quick way of dealing with the file handling.

See if you can complete the code that creates the operating system command string that displays the text comment, followed by the timestamp.

#include <stdio.h> #include <stdlib.h> #include <time.h>

 This function returns a string

char\* now() {

containing the current date and time.

time\_t t;

time (&t);

return asctime(localtime (&t)); }

/\* Master Control Program utility.

Records guard patrol check-ins. \*/

int main() {

char comment[80]; char cmd[120];

( , , ); ( ,

,



, );

system(cmd); return 0;

}

 **sprintf** 



|  |
| --- |
| **"echo '%s %s' >> reports.log"** |



 **comment** 

**fgets**

**comment**

**80**

 **cmd** 

**scanf**

**stdout**

|  |
| --- |
| **printf** |

|  |
| --- |
| **120** |

|  |
| --- |
|  |

**now()**

***you are here*** **399**



|  |
| --- |
| **stdin** |

|  |
| --- |
|  |

|  |
| --- |
|  |

***magnets moved***



Code Magnets Solution

This is a program that writes timestamped text to the end of a logfile. It would have been perfectly possible to write this entire program in C, but the programmer has used a call to system()as a quick way of dealing with the file handling.

You were to complete the code that creates the operating system command string that displays the text comment, followed by the timestamp.

#include <stdio.h> #include <stdlib.h> #include <time.h>

char\* now() {

time\_t t; time (&t);

return asctime(localtime (&t)); }

Master Control Program utility.

/\*

Records guard patrol check-ins. \*/

It needs to store the text in the comment array.

int main() {

There is room

The data will come from the Standard Input: the keyboard.

for only 80

characters.

char comment[80]; char cmd[120];



|  |
| --- |
| **stdin** |

**fgets**

**80**

**comment**

. (

);

Using fgets for unstructured

text.

,

,



(

**cmd**

**sprintf**

sprintf will print the characters to

The formatted string will be stored in the cmd array.

a string.

The command will append

the comment to a file.

,

This is the command template.

|  |
| --- |
| **"echo '%s %s' >> reports.log"** |



**now()**

**comment**

);

,

This runs the

system(cmd); return 0;

contents of

The comment will

The timestamp appears second.

the cmd string.

appear first.

}

|  |
| --- |
| **120** |

|  |
| --- |
| **printf** |



**stdout**

**scanf**

**400 *Chapter 9***

***processes and system calls***

 ~~Test Drive~~

Let’s compile the program and then watch it in action:

This will compile

the program.  This runs the 

|  |
| --- |
| File Edit Window Help Who’sYourUser |
| **> gcc guard\_log.c -o guard\_log > ./guard\_log**  **Checked in Crom - a compound interest program. > ./guard\_log**  **Blue Leader reports breach in jet walls. >** |

This is a comment. Another comment

program.

Running it a second time

Now, when you look in the same directory as the program, there’sa new file that’s been created called *reports.log*:

|  |
| --- |
| Checked in Crom - a compound interest program. Thu Oct 29 11:25:53 2015  Blue Leader reports breach in jet walls.  Thu Oct 29 11:26:06 2015 |

These are the timestamps.

This is the

reports.log file

the program

created.

The program worked. It read a comment from the command line and called the echo command to add the comment to the end of the file.

**reports.log**

Eventhough you could have written the whole program inC, by using system(), you simplified the program and got it working with very little work.

~~bt"iesxions~~

Q: **Does the system()function get compiled into my program?**

No. The system()function—like all system calls—

A:

doesn’t live in your program. It lives in the main operating system.

Q: **So, when I make a system call, I’m making a call to some external piece of code, like a library?**

Kind of. But the details depend on the operating system.

A:

On some operating systems, the code for a system call lives inside the kernel of the operating system. On other operating systems, it might simply be stored in some dynamic library.

***you are here*** **401**

***yikes***

Then someone busted into the system…

There’sa downside to the system() function. It’s

quick and easy to use, but it’s also kinda sloppy. Before getting into the problems with system(), let’s see

ALERT! ALERT! Main system security has been breached!

what it takes to break the program.

The code worked by stitching together a string containing a command, like this:

|  |
| --- |
| echo ' |

|  |
| --- |
| <timestamp> |

|  |
| --- |
| ' >> reports.log |

|  |
| --- |
| <comment> |

But what if someone entered a comment like this?



|  |
| --- |
| echo ' |

|  |
| --- |
| <timestamp> |

|  |
| --- |
| ' >> reports.log |

|  |
| --- |
| **' && ls / && echo '** |

1 l 

By *injecting* some command-line code into the text,

you can make the program run **whatever code you like**:

|  |
| --- |
| File Edit Window Help Yikes |
| **> ./guard\_log**  **' && ls / && echo '**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Applications Developer**  **Library**  **Network**  **Space Paranoids >** | **Source** | **System Users**  **Volumes bin**  **cores** | **dev etc home**  **mach\_kernel net** | **private sbin**  **tmp usr**  **var** | |

The user can use

the program to  run any command

she likes on the computer.

This is a listing of the root directory.

Is this a big problem? If a user can run guard\_log, she can just as easily run some other program. But

what if your code has been called from a *web server*? Or if it’s processing data from a *file*?

**402 *Chapter 9***

***processes and system calls***

Security’s not the only problem

This example injects a piece of code to list the contents of the root directory, but it could have *deletedfiles* or *launched a virus*. But you shouldn’t just worry about security.

★

★

★

Geek

Bits

**What if the comments contain apostrophes?**

That might break the quotes in the command.

**What if the PATH variable causes the system() function to call the wrong program?**

**What if the program we’re calling needs to have a specific set of environment variables set up first?**

The system() function is easy to use, but most of the time, you’re going to need something more structured—some way of calling a *specific* program, with a set of command-line arguments and maybe even some *environment variables*.

|  |
| --- |
| What’s the kernel?  On most machines, system calls are functions that live inside the **kernel** of the operating system. But  what is the kernel? You never actually *see* the kernel on the screen, but it’s always there, controlling your computer. The kernel is the most important program on your computer, and it’s in charge of **three things:**  **Processes**  No program can run on the system without the kernel loading it into memory. The kernel creates  processes and makes sure they get the resources they need. The kernel also watches for processes that become too greedy or crash.  **Memory**  Your machine has a limited supply of memory, so the kernel has to carefully ration the amount of memory each process can take. The kernel can increase the **virtual memory size** by quietly loading and unloading sections of memory to disk.  **Hardware**  The kernel uses **device drivers** to talk to the equipment that’s plugged into the computer. Your program can use the keyboard and the screen and the graphics processor without knowing too much about them, because the kernel talks to them on your behalf.  **System calls are the functions that your program uses to talk to the kernel.** |

***you are here*** **403**

***exec()***

The exec() functions give you more control

When you call the system() function, the operating

system has to interpret the command string and decide

which programs to run and how to run them. And that’s

where the problem is: the operating system needs to *interpret* the string, and you’ve already seen how easy it is to get that wrong. So, the solution is to remove the **ambiguity** and

tell the operating system precisely which program you want to run. That’s what the **exec()** functions are for.

exec() functions replace the current process

A process is just a program running in memory. If you

type **taskmgr** on Windows or **ps -ef** on most other

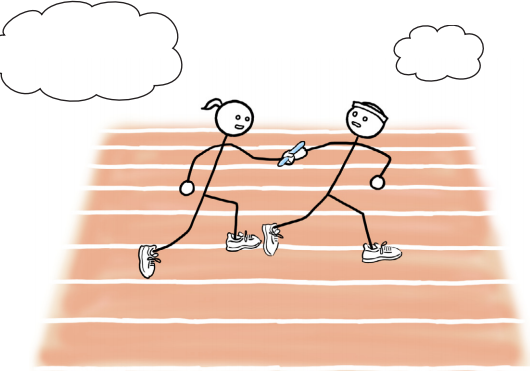
machines, you’ll see the processes running on your system. The operating system tracks each process with a number called the **process identifier** (**PID**).

The exec() functions **replace the current process** by running some other program. You can say which *command- line arguments* or *environment variables* to use, and when the

new program starts it will have exactly the same PID as the old one. It’s like a relay race, where your program hands

over its process to the new program.

A process is a program running in memory.



OK, I’m handing over to you now, sendmail. This is the data you need. Don’t let me down.



**404 *Chapter 9***

I’m all over it.

***processes and system calls***

There are many exec() functions

Overtime, programmers have created several different versions of exec(). Each version has a slightly different name and its own set of parameters. Eventhough there are lots of versions, there are really just two groups of exec() functions: the **list** functions and the **array** functions.

The list functions: execl(), execlp(), execle()

The list functions accept command-line arguments as a list of parameters, like this:

|  |  |
| --- | --- |
| ¥  ¥  ¥ | **The program.**  This might be the full pathname of the program—execl()/  execle()—or just a command name to search for—execlp()— but the first parameter tells the exec() function what program it  will run.  **The command-line arguments.**  You need to list one by one the command-line arguments you want to use. Remember: the ***first*** command-line argument is always the name of the program. That means the first two parameters passed to a list version of exec() should always be the ***same string***.  **NULL.**  That’s right. After the last command-line argument, you need a **NULL**. This tells the function that there are no more arguments. |

The exec() functions are in unistd.h.

|  |
| --- |
| **Spaces** **in**  **command** **line** **arguments**  **can** **confuse**  **MinGW.**  *If you pass two arguments “I like” and “turtles,” MinGW programs might send* **three** *arguments: “I,” “like,” and*  *“turtles. ”* |

¥

**Environment variables (maybe).**

If you call an exec() function whose name ends with ...e(), you can also pass an array of environment variables. This is just an array of strings like "POWER=4", "SPEED=17", "PORT=OPEN", ....

execL = a LIST of arguments. These are the arguments.

**execl("/home/flynn/clu", "/home/flynn/clu", "paranoids", "contract", NULL)**

The second

parameter

should be

the same as

You should end the list with NULL.

execLP = a LIST of arguments + search on the PATH.

These are the arguments.

**"paranoids", "contract", NULL)**

**execlp("clu", "clu",**

the first.

These are the arguments.

**execle("/home/flynn/clu", "/home/flynn/clu", "paranoids", "contract", NULL, env\_vars)**

execLE = a LIST of arguments env\_vars is an array of strings

个

+ ENVIRONMENT variables. containing environment variables.

***you are here*** **405**

***array functions***

The array functions: execv(), execvp(), execve()

If you already have your command-line arguments stored in an array, you might find these two versions easier to use:

VECTOR of arguments.

execV = an array or 

VECTOR of arguments

execVP = an array/

+ search on the PATH.

**execv("/home/flynn/clu", my\_args);**

The arguments need to be stored

**execvp("clu", my\_args);**

in the my\_args string array.

The only difference between these two functions is that

**execvp** will search for the program using the PATH variable.

How to remember the exec() functions

You can figure out which exec() function you need by

constructing the name. Each exec() function can be followed by one or two characters that must be**l**, **v**, **p**, or **e**. The

characters tell you which feature you want to use. So, for the execle() function:

execle = exec + l + e = LIST of arguments + an ENVIRONMENT

The land vcharacters always come before p ande, and the p ande characters are optional.

Character

l

v

p e

Uses

List of args

Array/vector of args Search the path

Environment vars

Take a list of arguments.



Use an array

**l**

All exec() functions begin with exec.

of environment

strings.

V

Search

You don’t have to

include p or e.

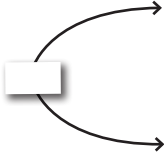
for the program on the path.



**p**

**exec**

**e**

Take a vector/array

of arguments.



**v**

**406 *Chapter 9***

***processes and system calls***

Passing environment variables

Every process has a set of *environment*

*variables*. These are the values you see when you type set or envon the command

line, and they usually tell the process

useful information, such as the location

of the home directory or where to find

the commands. C programs can read

environment variables with the **getenv()** system call. You can see getenv() being used in the diner\_info program on the right.

If you want to run a program using

command-line arguments *and* environment

variables, you can do it like this: **diner\_info.c**

Each variable in the The last item in the

You can create a environment is name=value. array must be NULL.

|  |
| --- |
| 个  getenv() in stdlib.h lets you read environment variables. |

#include <stdio.h> #include <stdlib.h>

int main(int argc, char \*argv[]) {

printf("Diners: %s\n", argv[1]);

printf("Juice: %s\n", **getenv** ("JUICE")); return 0;

}

set of environment **char \*my\_env[] = {"JUICE=peach and apple", NULL};** rioray

**execle("diner\_info", "diner\_info", "4", NULL, my\_env);**

 个

execle passes a list of arguments and an environment. my\_env contains the environment.

The execle() function will set the command-line

arguments and environment variables and then replace the current process with diner\_info.

|  |
| --- |
| File Edit Window Help MoreOJ  **> ./my\_exec\_program Diners: 4**  **Juice: peach and apple >** |

But what if there’sa problem?

If there’sa problem calling the program, the existing process will keep running. That’s useful, because it means that if you can’t start that second process, you’ll be able to recover from the error and give the user more information on what went wrong. And luckily, the C Standard Library provides some built-in code to help you with that.



|  |
| --- |
| **If** **you’re**  **passing** **an** **environment** **on** **Cygwin,**  **be** **sure** **to**  **include** **a** **PATH** **variable.**  *On Cygwin, the* PATH  *variable is needed when*  *programs are loaded. So, if you’re passing environment variables on Cygwin, be sure to include* PATH=/usr/bin*.* |

***you are here*** **407**

***errno***

Most system calls go wrong in the same way

Because system calls depend on something *outside* your

program, they might go wrong in some way that you can’t control. To deal with this problem, most system calls go

wrong in the same way.

Take the execle() call, for example. It’s really easy to

see when an exec() call goes wrong. If an exec() call is successful, the current program stops running. So, if the program runs *anything* after the call to exec(), there must have been a problem:



**Guaranteed**

**Standard of**

**Failure**

If execle() worked,

this line of code

would never run. 

execle("diner\_info", "diner\_info", "4", NULL, my\_env); puts("Dude - the diner\_info code must be busted");

But just telling *if* a system call worked is not enough. You

normally want to know *why* a system call failed. That’s why

most system calls follow the **golden rules of failure**.

The **errno** variable is a global variable that’sdefined in *errno.h*, along with a whole bunch of standard error values, like:

Operation not permitted No such file or directory No such process

Bad haircut

EPERM=1

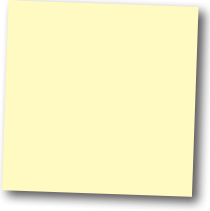
ENOENT=2

ESRCH=3

EMULLET=81

This value is

not available on all systems.



The Golden Rules

of Failure

\* Tidy up as much as you can.

\* Set the errno variable to an error value.

\* Return -1.

Now you *could* check the value of errno against each of these values, or you could look up a standard piece of error text using a function in *string.h* called **strerror()**:

**puts(strerror(errno));** strerror() converts an error number into a message.

So, if the system can’t find the program you are running and it sets the errno variable to ENOENT, the above code will display this message:

No such file or directory

**408 *Chapter 9***

***processes and system calls***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Different machines have different commands to tell you about their network configuration. On Linux and Mac machines, there’s the /sbin/ifconfig program, and on Windows there’s a command called ipconfig that’s stored somewhere on the **command path**.  This program tries to run the /sbin/ifconfig program and, if that fails, it will try the  ipconfig command. There’s no need to pass arguments to either command. Think carefully. What type of exec() commands will you need? | | |
| #include <stdio.h> | |  | What headers will you need? |
| i{nt main() itwdwoti/ifconfig. is idcird  if ( )  if (execlp( ) { fprintf(stderr, "Cannot run ipconfig: %s", );  return 1; 个  } What do you think goes here?  return 0; } | | | |

***you are here*** **409**

***exercise solved***

|  |  |
| --- | --- |
|  | Different machines have different commands to tell you about their network configuration. On Linux and Mac machines, there’s the /sbin/ifconfig program, and on Windows there’s a command called ipconfig that’s stored somewhere on the **command path**.  This program tries to run the /sbin/ifconfig program and, if that fails, it will try the  ipconfig command. There’s no need to pass arguments to either command. Think carefully. What type of exec() commands will you need? |
| #include <unistd.h>  You need this for the exec() functions.  #include <stdio.h>  #include <errno.h>  You need this· for the errno variable.  #include <string.h> This will let you display errors with strerror().  i{nt main() hectl)e ilave the e ) il,oig.  if ( · execl(“/sbin/ifconfig”, “/sbin/ifconfig”, NULL) == -1 · )  if (execlp( · “ipconfig”, “ipconfig”, NULL) == -1 · ) {  el)iill 一fprintf(stderr, "Cannot run ipconfig: %s", strerror(errno) · );  i } return 1; dvle1 in n(y) i.  the path. return 0; } | |

***processes and system calls***

~~bestiom~~

Q: **Isn’t system()just easier to use than exec()?**

Yes. But because the operating system needs to interpret the string you pass to system(), it can be a bit buggy. Particularly if you create the command string dynamically.

A:

A:

Q:**Why are there so many exec()functions?**

Over time, people wanted to create processes in different ways. The different versions of exec()were created for more flexibility.

Q: **Do I always have to check the return value of a system call? Doesn’t it make the program really long?**

If you make system calls and don’t check for errors, your

A:

code will be shorter. But it will probably also have more bugs. It is better to think about errors when you first write code. It will make it much easier to catch bugs later on.

Q: **If I call an exec()function, can Ido anything afterward?**

No. If the exec()function is successful, it will change the process so that it runs the new program instead of your program. That means the program containing the exec()call will stop as soon as it runs the exec()function.

A:

|  |  |
| --- | --- |
| ■ System calls are functions that live in the operating system.  ■ When you make a system call, you are calling code outside your program.  ■ system() is a system call to run a command string.  ■ system() is easy to use, but it can cause bugs. | ■ The exec() system calls let you run programs with more control.  ■ There are several versions of the exec() system call.  ■ System calls usually, but not always, return –1 if there’s a problem.  ■ They will also set the errno variable to an error number. |

***you are here*** **411**

***mixed messages***



**Mixed Messages**

The guys over at Starbuzz have come up with a new order-generation program that they call **coffee**:

#include <stdio.h> #include <stdlib.h>

int main(int argc, char \*argv[]) {

char \*w = getenv("EXTRA"); if (!w)

w = getenv("FOOD"); if (!w)

w = argv[argc - 1];

char \*c = getenv("EXTRA"); if (!c)

c = argv[argc - 1];

printf("%s with %s\n", c, w); return 0;

}

To try it out, they’ve created this test program. Can you match up these code fragments to the output they produce?

|  |  |  |  |
| --- | --- | --- | --- |
| #include <string.h> #include <stdio.h> #include <errno.h>  int main(int argc, char \*argv[]){ |  | Candidate code goes | here. |
| |  | | --- | |  |   fprintf(stderr,"Can't create order: %s\n", strerror(errno)); return 1;  }  return 0; } | | | |

**412 *Chapter 9***

***processes and system calls***

**Candidates:** thpi 



**char \*my\_env[] = {"FOOD=coffee", NULL};**

**if(execle("./coffee", "./coffee", "donuts", NULL, my\_env) == -1){ fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));**

**return 1;**

**}**

**Possible output:**

**coffee with donuts**

**cream with donuts**

**char \*my\_env[] = {"FOOD=donuts", NULL};**

**if(execle("./coffee", "./coffee", "cream", NULL, my\_env) == -1){ fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

**donuts with coffee**

**if(execl("./coffee", "coffee", NULL) == -1){**

**fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

**coffee with coffee**

**char \*my\_env[] = {"FOOD=donuts", NULL};**

**if(execle("./coffee", "coffee", NULL, my\_env) == -1){**

**fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

***you are here*** **413**

***messages unmixed***



**Mixed Messages Solution**

The guys over at Starbuzz have come up with a new order-generation program that they call **coffee**:

#include <stdio.h> #include <stdlib.h>

int main(int argc, char \*argv[]) {

char \*w = getenv("EXTRA"); if (!w)

w = getenv("FOOD"); if (!w)

w = argv[argc - 1];

char \*c = getenv("EXTRA"); if (!c)

c = argv[argc - 1];

printf("%s with %s\n", c, w); return 0;

}

To try it out, they’ve created this test program. Can you match up these code fragments to the output they produce?

|  |  |  |  |
| --- | --- | --- | --- |
| #include <string.h> #include <stdio.h> #include <errno.h>  int main(int argc, char \*argv[]){ |  | Candidate code goes | here. |
| |  | | --- | |  |   fprintf(stderr,"Can't create order: %s\n", strerror(errno)); return 1;  }  return 0; } | | | |

**414 *Chapter 9***

***processes and system calls***

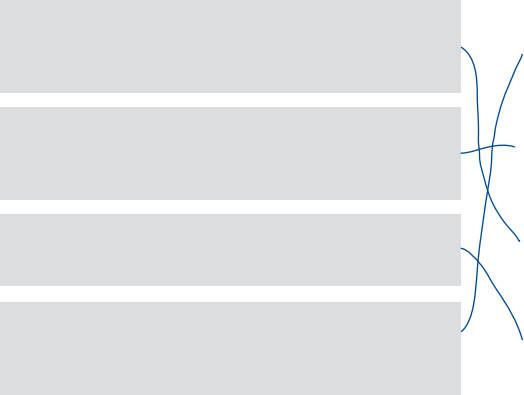
**Candidates: Possible output:**

**coffee with donuts**

**cream with donuts**

**donuts with coffee**

**coffee with coffee**



**char \*my\_env[] = {"FOOD=coffee", NULL};**

**if(execle("./coffee", "./coffee", "donuts", NULL, my\_env) == -1){ fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));**

**return 1;**

**}**

**char \*my\_env[] = {"FOOD=donuts", NULL};**

**if(execle("./coffee", "./coffee", "cream", NULL, my\_env) == -1){ fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

**if(execl("./coffee", "coffee", NULL) == -1){**

**fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

**char \*my\_env[] = {"FOOD=donuts", NULL};**

**if(execle("./coffee", "coffee", NULL, my\_env) == -1){**

**fprintf(stderr,"Can't run process 0: %s\n", strerror(errno)); return 1;**

**}**

***you are here*** **415**

***rss gossip***

Read the news with RSS

RSS feeds area common way for websites to publish their latest news stories. Each RSS feed is just an

XML file containing a summary of stories and links. Of course, it’s possible to write a C program that will read RSS files straight off the Web, but it involves

a few programming ideas that you haven’t seen yet. But that’s not a problem if you can find another

program that will handle the RSS processing for you.



Do this!

\*

c \*

|  |
| --- |
| Download RSS Gossip from  [*https://github.com/dogriffiths/rssgossip/zipball/master*](https://github.com/dogriffiths/rssgossip/zipball/master).  Also, if you don’t have Python installed, you can get it here: [*http://www.python.org/*](http://www.python.org/). |

I want all the latest

stories on Pajama Death.

**RSS Gossip** is a small **Python script** that can search RSS feeds for stories containing a piece of text. To run the script, you will need Python installed. Once you

have Python and *rssgossip.py*, you can search for stories like this:

This is running in a Unix environment.

You need to create

an environment

variable containing

This isn’t a

|  |
| --- |
| File Edit Window Help ReadAllAboutIt  **> export RSS\_FEED=http://www.cnn.com/rss/celebs.xml > python rssgossip.py 'pajama death'**  **Pajama Death launch own range of kitchen appliances. Lead singer of Pajama Death has new love interest.**  **"I never ate the bat" says Pajama Death's Hancock.** |

real feed.

You should

replace it

with one you

the address of an

RSS feed.

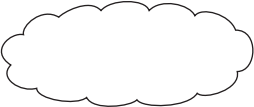
This runs the

find online.

rssgossip script with a search string.



 Editor



Ooh, I just had a great idea. Why not write a program that can search **a lot of** RSS feeds



c c all at once! Can you do that?

**416 *Chapter 9***

***processes and system calls***

|  |  |
| --- | --- |
|  | The editor wants a program on his machine that can search a lot of RSS feeds all at the  same time. You could do that if you ran the *rssgossip.py* several times for different RSS feeds. Fortunately, the **out-of-work actors** have made a start on the program for you. Trouble is,  they’re having problems creating the call to exec() the *rssgossip.py* script. Think carefully about what you need to do to run the script, and then complete the **newshound** code. |
| This is an  environment  array.  You need  to insert  the  function  name  here.   |  |  | | --- | --- | | To save space, this listing doesn’t    int main(int argc, char \*argv[]) { | These are RSS feeds the editor wants (you might want to choose your own).  匕 |   char \*feeds[] = {"http://www.cnn.com/rss/celebs.xml",  "<http://www.rollingstone.com/rock.xml>", "<http://eonline.com/gossip.xml>"};  We’ll pass the search terms in as an argument. Loop through each of the feeds.  for (i = 0; i < times; i++) {  char var[255];  sprintf(var, "RSS\_FEED=%s", feeds[i]);  char \*vars[] = {var, NULL};  On the editor’s Mac, Python is installed here. if ( ("/usr/bin/python", "/usr/bin/python",  ) == -1) {  fprintf(stderr, "Can't run script: %s\n", strerror(errno)); return 1; You need to insert the other  } parameters to the function here. }  return 0; }  **newshound.c**  int times = 3;  char \*phrase = argv[1]; int i;  show the #include lines.  **And for extra bonus points …**  What will the program do when it runs? | |

***you are here*** **417**

***newshound hounded***

|  |  |
| --- | --- |
|  | The editor wants a program on his machine that can search a lot of RSS feeds all at the  same time. You could do that if you ran the *rssgossip.py* several times for different RSS feeds. Fortunately, the **out-of-work actors** have made a start on the program for you. Trouble is,  they’re having problems creating the call to exec() the *rssgossip.py* script. You were to think carefully about what you need to do to run the script, and then complete the **newshound** code. |
| · execle ·  You’re using  a LIST of  args and an ENVIRON-  · “./rssgossip.py”, phrase, NULL, vars ·  This is the  search phrase, as a command- line argument.  Pass the  environment as an extra parameter.  MENT, so  it’s execLE.  This is the  name of the Python script.  **newshound.c**  int main(int argc, char \*argv[]) {  char \*feeds[] = {"http://www.cnn.com/rss/celebs.xml",  "<http://www.rollingstone.com/rock.xml>", "<http://eonline.com/gossip.xml>"};  int times = 3;  char \*phrase = argv[1]; int i;  for (i = 0; i < times; i++) { char var[255];  sprintf(var, "RSS\_FEED=%s", feeds[i]); char \*vars[] = {var, NULL};  if ( ("/usr/bin/python", "/usr/bin/python",  ) == -1) {  fprintf(stderr, "Can't run script: %s\n", strerror(errno)); return 1;  } }  return 0; } | |
| **But what will the program do when you run it?** | |

**418 *Chapter 9***

***processes and system calls***

 ~~Test Drive~~

When you compile and run the program, it looks like it works:

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./newshound 'pajama death'**  **Pajama Death ex-drummer tells all.**  **New Pajama Death album due next month.** |

The newshound program has the *rssgossip.py* script using data from the array of RSS feeds.

Worked!? Worked?!? It didn’t work! What about the announcement of the surprise concert? That was on every other news site! I coulda sent my photographers down there. As it is, I was beaten to the story by everyone else in town!

**Actually** **there** **is** **a** **problem.**

Although the newshound program managed

to run the *rssgossip.py* script, it looks like it didn’t manage to run the script for *all of the feeds*. In fact, the only news it displayed came from the **first**

**feed on the list**. That meant the other news stories matching the search terms were missed.





|  |
| --- |
| Look at the code of the newshound program again and think about how it works. Why do you think it failed to run the *rssgossip.py* script for any of the other  newsfeeds? |



***you are here*** **419**

***fork()***

exec() is the end of the line for your program

The exec() functions *replace* the current

Once the newshound program hands over the process to the rssgossip.py program, newshound quits.

function by running a new program. But what

happens to the original program? It terminates, and it terminates **immediately**. That’swhy the program only ran the *rssgossip.py* script for the

newshound

rssgossip.py

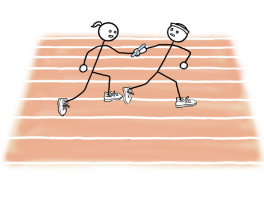
first newsfeed. After it had called execle() the first time, the newshound program terminated.

The loop will run only once.



for (i = 0; i < times; i++) {

...

if (execle("/usr/bin/python", "/usr/bin/python",

"./rssgossip.py", phrase, NULL, vars) == -1) {

Once execle() is called, the whole program quits.

}

...

}

But if you want to start *another* process and keep your original process running, how do you do it?

fork() will clone your process

You’re going to get around this problem by using a system call named **fork()**.

fork() makes a complete **copy** of the current process. The brand-new copy will be running

the same program, on the same line number. It will have exactly the same variables that contain exactly the same values. The only difference is

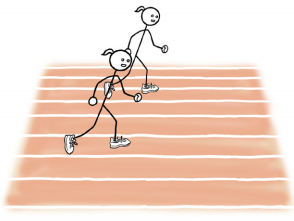
that the copy process will have a different process identifier from the original.

The original process is called the **parent**

**process**, and the newly created copy is called the **child process**.

But how can cloning the current process fix the problems with exec()? Let’ssee.

|  |
| --- |
| **Unlike** **Linux** **and** **the** **Mac,**  **Windows**  **doesn’t** **support** **fork() natively.**  *To use* fork() *on a Windows machine, you should first install Cygwin.* |

The fork() system call will

clone the current process. 



The original

process is

called the

parent process.

The new process  is called the

child process.

**420 *Chapter 9***

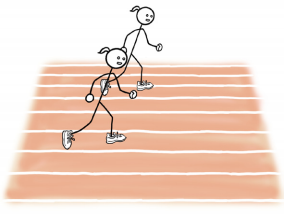
***processes and system calls***

Running a child process with fork() + exec()

The trick is to only call an exec() function on a *child*

*process*. That way, your original parent process will be able to continue running. Let’s look at the process step by step.

k-New process 1234

The original

process

1. Make a copy

Begin by making a copy of your current process by calling the fork() system call.

The processes need some way of telling which of them is the parent process and which is the child, so the fork() function returns 0 to the child process, and it will return a **nonzero** value to the parent process.

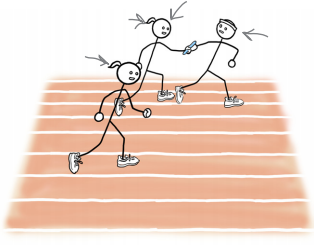
2. If you’re the child process, call exec()

At this point, you have two identical processes running,

both of them using identical code. But the child process

(the one that received a 0 from the fork() call) now needs to replace itself by calling exec():

The child process calls exec().

This is the child This is the parent process.

process.

The child process is

replaced by rssgossip.py.

Now you have two separate processes: the child process is running the *rssgossip.py* script, and the original parent process is free to continue doing something else.

***you are here*** **421**

***code magnets***



Code Magnets

It’s time to update the newshound program. The code needs to run the *rssgossip.py* script in a separate process for each of the RSS feeds. The code is reduced, so you only have to worry about the main loop. Be careful to check for errors, and don’t get the parent and child processes mixed!

for (i = 0; i < times; i++) { char var[255];

Put your magnets in this

space.

sprintf(var, "RSS\_FEED=%s", feeds[i]); char \*vars[] = {var, NULL};

}

**422 *Chapter 9***

***processes and system calls***

|  |
| --- |
| the  What fork()?  You call fork() like this:  **pid\_t pid = fork();**  fork() will actually return an integer value that is 0 for the child process and positive for the parent process. The parent process will receive the process identifier of the child process.  But what is **pid\_t**? Different operating systems use  different kinds of integers to store process IDs: some  might use shorts and some might use ints. So pid\_t is always set to the type that the operating system uses. |



|  |
| --- |
| **fprintf(stderr, "Can't fork process: %s\n", strerror(errno));** |

**fprintf(stderr, "Can't run script: %s\n", strerror(errno));**

**if (execle("/usr/bin/python", "/usr/bin/python",**

**phrase, NULL, vars) == -1) {**

**"./rssgossip.py",**

**pid\_t pid = fork();**

|  |
| --- |
| **return 1;** |



**}**

**if (pid == -1) {**

**if (!pid) {**

|  |
| --- |
|  |



|  |
| --- |
|  |

|  |
| --- |
| **}** |

|  |
| --- |
| **}** |



|  |
| --- |
| **return 1;** |

***you are here*** **423**

***magnets unmuddled***



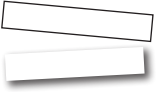
Code Magnets Solution

It’s time to update the newshound program. The code needs to run the *rssgossip.py* script in a separate process for each of the RSS feeds. The code is reduced, so you only had to worry about the main loop. Be careful to

check for errors, and don’t get the parent and child processes mixed!

for (i = 0; i < times; i++) { char var[255];

sprintf(var, "RSS\_FEED=%s", feeds[i]); char \*vars[] = {var, NULL};

First, call fork() to clone the process.

**pid\_t pid = fork();**

If fork() returned -1, there was a problem cloning the process.

**if (pid == -1) {**



**return 1;**



|  |
| --- |
| **fprintf(stderr, "Can't fork process: %s\n", strerror(errno));** |

If fork() returned a 0, the code is

This is the same as if (pid == 0).  running in the child process.

**}**

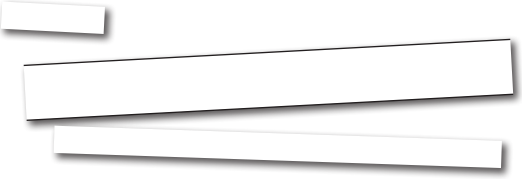


**if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",**

**phrase, NULL, vars) == -1) {**

**if (!pid) {**

|  |
| --- |
| **return 1;** |



If you get here, you’re the child process,

so we should exec() the script.



**fprintf(stderr, "Can't run script: %s\n", strerror(errno));**

|  |
| --- |
| **}** |

|  |
| --- |
| **}** |

}

***processes and system calls***

 ~~Test Drive~~

Now, if you compile and run the code, this happens:



Hey! That’s

great! I’ll send my photographers down to the premiere.

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./newshound 'pajama death'**  **Pajama Death ex-drummer tells all.**  **New Pajama Death album due next month.**  **Photos from the surprise Pajama Death concert. Official Pajama Death pajamas go on sale.**  **"When Pajama Death jumped the shark" by HenryW. Breaking News: Pajama Death attend premiere.** |

By fork-inga copy of itself and then exec-ing the

Python script in a separate process, the newshound

program is able to run a separate process for each of the RSS feeds. And the great thing is that these processes will all run **at the sametime**.

This is your

newshound

process.



It runs separate



processes for each of

the three newsfeeds.

**newshound**





The child processes all

run at the same time.

That’sa lot faster than reading the newsfeeds one at a time. By learning how to create and run separate processes with fork() and exec(), not only can you make the most

of your existing software, but you can also improve the performance of your code.

***you are here*** **425**

***no dumb questions***

~~btiexion~~

Q: **Does system() run programs in a separate process?**

Yes. But system()gives you less control over exactly how the program runs.

A:

Q: **Isn’t fork-ing processes really inefficient?I mean, it copies an entire process, and then a moment later we replace the child process by doing an exec()?**

Operating systems use lots of tricks to make fork-ing processes really quick. For example, the operating system cheats and avoids making an actual copy of the

A:

parent process’s data. Instead, the child

and parent processes share the same data.

**But what if one of the processes changes some data in memory? Won’t that screw things up?**

Q:

A:

It would, but the operating system

will catch that a piece of memory is going to change, and then it will make a separate copy of that piece of memory for the child process.

Q: **That technique sounds quite cool. Does it have a name?**

Yes; it’s called “copy-on-write.” Q: **Is a pid\_tjust an int?**

A:

A: It depends on the platform. The only thing you know is that it will be some integer type.

Q: **I stored the result of a fork() call in an int, and it worked just fine.**

It’s best to always use pid\_t to store process IDs. If you don’t, you might cause problems with other system calls or if your code is compiled on another

A:

machine.

**Why doesn’tWindows support the fork()system call?**

Q:

A:

Windows manages processes very differently from other operating systems, and the kinds of tricks fork() needs to do in order to work efficiently are very hard to do on Windows. This may be why there isn’t a version of fork() built in.

Q: **But Cygwin lets me do fork()s on Windows, right?**

Yes. The gurus who work on

A:

Cygwin did a lot of work to make Windows processes look like processes that are

used on Unix, Linux, and the Mac. But

because they still need to rely on Windows to create the underlying processes,

fork()on Cygwin can be a little slower than fork()on other platforms.

Q: **So, if I’m just interested in writing code to work on Windows, is there**

**something else I should use instead?**

Yes. There’s a function called

A:

CreateProcess()that’s like an

enhanced version of system(). To find out more, go to [*http://msdn.microsoft.com*](http://msdn.microsoft.com)and search for “CreateProcess.”

Q:**Won’t the output of the various feeds get mixed up?**

A: The operating system will make sure that each string is printed completely.

|  |  |
| --- | --- |
| ■ System calls are functions that live in the kernel.  ■ The exec()functions give you more control than system().  ■ The exec()functions replace the current process. | ■ The fork()function duplicates the current process.  ■ System calls usually return –1 if they fail.  ■ Failed system calls set the errno variable to the error number. |

**426 *Chapter 9***

***processes and system calls***

**CHAPTER 9**

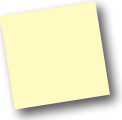
Your C Toolbox

**You’ve** **got** **Chapter** **9** **under** **your** **belt,** **and** **now** **you’ve**

**added** **processes** **and** **system**

**calls** **to** **your** **toolbox.** **For** **a** **complete**

**list** **of** **tooltips** **in** **the** **book,** **see** **Appendix** **ii.**

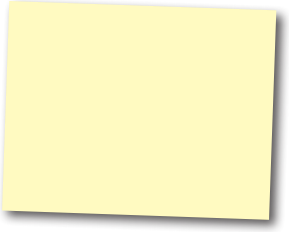


system() will

run a string

like a console

command.



execl() = list of args.

execle() = list of args + environment. execlp() = list of args + search on path. execv() = array of args.

execve() = array of args + environment. execvp() = array of args + search on path.

fork()

duplicates

the current

process.

fork() +

exec()

creates a

child process.

***you are here*** **427**