**10** interprocess communication



\***It’s** **good** **to** **talk**+



**Creating processes is just half the story.**

What if you want to *control* the process once it’s running? What if you want to *send it*

*data*? Or *read its output*? **Interprocess communication** lets processes work together to *get the job done*. We’ll show you how to multiply the **power** of your code by letting it ***talk*** to other programs on your system.

**this is a new chapter 429**

***redirection***

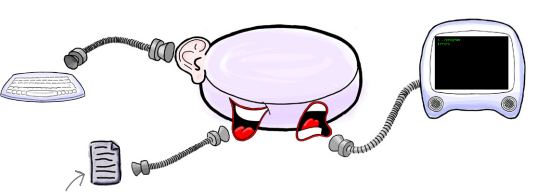
Redirecting input and output

When you run programs from the command line, you can redirect the Standard Output to a file using the > operator:

python ./rssgossip.py Snooki > stories.txt

You can redirect output using the > operator.

The Standard

Input: stdin



The Standard Output: stdout

You can redirect

the Standard

The Standard Error: stderr

Output to a file.

The Standard Output is one of the three default **data**

**streams**. A *data stream* is exactly what it sounds like: a

stream of data that goes into, or comes out of, a process.

There are data streams for the Standard Input, Output,

and Error, and there are also data streams for other things, like files or network connections. When you redirect the

output of a process, you change where the data is sent. So, instead of the Standard Output sending data to the screen, you can make itsend the data to a file.

Redirection is really useful on the command line, but is there away of making a process ***redirect itself***?

**430 *Chapter 10***

***interprocess communication***

A look inside atypical process

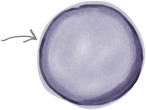
Every process will contain the program it’s running, as well as

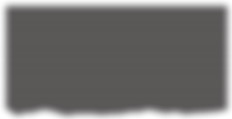
space for stack and heap data. But it will also need somewhere

to record where data streams like the Standard Output are

connected. Each data stream is represented by a **file descriptor**, which, under the surface, is just a number. The process keeps

everything straight by storing the file descriptors and their data streams in a **descriptor table**.



A typical process

Standard Input  Standard Output  Standard Error The process might also →

A file descriptor is a number that represents a data stream.

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | The keyboard |
| 1 | The screen |
| 2 | The screen |
| 3 | Database connection |

have other open streams.

The descriptor table has one column for each of the file

descriptor numbers. Eventhough these are called **file** descriptors, they might not be connected to an actual file on the hard disk.

Against every file descriptor, the table records the associated data stream. That data stream might be a connection to the keyboard or screen, a file pointer, or a connection to the network.

The first three slots in the table are always the same. Slot 0 is

the Standard Input, slot 1 is the Standard Output, and slot 2 is the Standard Error. The otherslots in the table are either empty or connected to data streams that the process has opened. For

example, every time your code opens a file for reading or writing, another slot is filled in the descriptor table.

When the process is created, the Standard Input is connected to the keyboard, and the Standard Output and Error are connected to the screen. And they will stay connected that way until

something redirects them somewhere else.

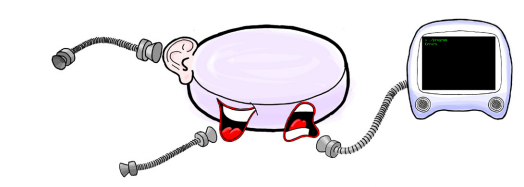
File descriptors don’t necessarily refer to files.

***you are here*** **431**

***replace the descriptors***

Redirection just replaces data streams

The Standard Input, Output, and Error are always fixed in the same places in the descriptor table. But the data streams they point to can

change.

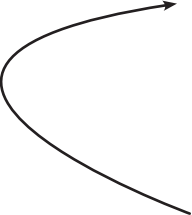


Standard Output

has been redirected



to a file.

That means if you want to redirect the Standard

Bits

Geek

Output, you just need to switch the data stream against descriptor 1 in the table.

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | The keyboard |
| 1 | The screen File stories.txt |
| 2 | The screen |
| 3 | Database connection |

All of the functions, like printf(), that send data to the Standard Output will first look in the descriptor table to see where descriptor 1 is pointing. They will then write data out to the correct data stream.

Processes can redirect themselves

Every time you’ve used redirection so far, it’sbeen from the command line using the > and < operators. But

processes can do their *own redirection* by **rewiring the descriptor table**.

|  |  |  |
| --- | --- | --- |
| So, that’swhy it’s 2 >…  You can redirect the Standard Output and  Standard Error on the command line using the > and 2> operators:  ./myprog > output.txt 2> errors.log  Now you can see why the Standard Error is  redirected with **2**>. The **2** refers to the number of the Standard Error in the descriptor table. On most operating systems, you can use **1**> as an alternative way of redirecting the Standard Output, and on Unix-based systems you can even redirect the Standard Error to the same place as the Standard Output like this:  ./myprog 2>&1 | | |
| 2> means “redirect Standard Error.” |  | 个  &1 means “to the Standard Input.” |

**432 *Chapter 10***

***interprocess communication***

fileno() tells you the descriptor

Every time you open a file, the operating system registers a

new item in the descriptor table. Let’s say you open a file with something like this:

FILE \*my\_file = fopen("guitar.mp3", "r"); The operating system will open the *guitar.mp3* file and return a

Hmmm…looks

like slot 4 is free; I’ll record the music file there.



pointer to it, but it will also skim through the descriptor table until it finds an empty slot and register the new file there.

But once you’ve got a file pointer, how do you find it in the descriptor table? The answer is by calling the **fileno()** function.

int descriptor = **fileno(my\_file)**;

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | The keyboard |
| 1 | The screen |
| 2 | The screen |
| 3 | Database connection |
| 4 | File guitar.mp3 |

This will return the value 4.

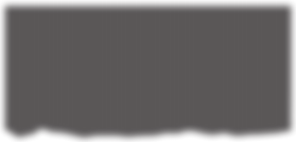
fileno() is one of the few system functions that doesn’t return –1 if it fails. As long as you pass fileno() the pointer to an

open file, it should always return the descriptor number.

dup2() duplicates data streams

Opening a file will fill aslot in the descriptor table, but what if you want to *change* the data stream already registered against

a descriptor? What if you want to change file descriptor 3 to

point to a different data stream? You can do it with the **dup2()** function. dup2() duplicates a data stream from one slot to

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | The keyboard |
| 1 | The screen |
| 2 | The screen |
| 3 | Database connection File guitar.mp3 |
| 4 | File guitar.mp3 |

another. So, if you have a file pointer to *guitar.mp3* plugged in to file descriptor 4, the following code will connect it to file descriptor 3 as well.

**dup2(4, 3)**;

There’s still just one *guitar.mp3* file, and there’s still just one data stream connected to it. But the data stream (the FILE\*) is now registered with file descriptors 3 and 4.

**Now** **that** **you** **know** **how** **to** **find** **and** **change** **things** **in** **the** **descriptor** **table,** **you** **should**

**be** **able** **to** **redirect** **the** **Standard** **Output** **of** **a** **process** **to** **point** **to** **a** **file.**

***you are here*** **433**

***sleepless nights***

Does your error code worry you?

Do you find that you’rewriting duplicate error-handling code

everytime you make a system call? Then fear no more! Using

our patented method, we’ll show you how to make the most out of

your error code without writing the same thing over and over.

Look at these two troublesome pieces of code:

pid\_t pid = fork(); if (pid == -1) {

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

} return 1;  puli ause

if (execle(...) == -1) {

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

}

m-a-i(n),l**,**ci**!**

What’s that, you say? How do you handle that troublesome return statement? After all, you can’t move **that** into a function, can you?

There’s no need! The exit() system call is the fastest way to stop your program in its tracks.

No more worrying about returning to main(); just call exit(), and your program’shistory!

This is how it works. First, remove all of your error code into a separate function called

error() and replace that tricky return with a call to exit().  To ensure you have the exit

void error(char \*msg) system call available, you need

to include stdlib.h.

{

n(f);(stderx,it"(w:ilegy,ot(n)a)ts 1 IMMEDIATELY!

}

Now you can replace that troublesome error-checking code with something much simpler:

pid\_t pid = fork(); if (pid == -1) {

error("Can't fork process"); }

if (execle(...) == -1) {

error("Can't run script");

}

**Warning: offer limited to one exit() call per program execution. Do not operate exit() if you have a fear of sudden program termination.**

**434 *Chapter 10***

***interprocess communication***

|  |
| --- |
| This is a program that saves the output of the *rssgossip.py*  script into a file called *stories.txt*. It’s similar to the newshound  program, except it searches through a single RSS feed only. Using what you’ve learned about the descriptor table, see if you can find the missing line of code that will redirect the **Standard Output** of  The #includes and the error() function the child process to the *stories.txt* file.  have been removed to save space. |
| ↓  int main(int argc, char \*argv[]) {  char \*phrase = argv[1];  char \*vars[] = {"RSS\_FEED=http://www.cnn.com/rss/celebs.xml", NULL}; FILE \*f = fopen("stories.txt", "w");  if (!f) {  If we can’t write to stories.txt, then f will be zero.  error("Can't open stories.txt");<— We’ll report errors using the error()  } function we wrote earlier.  pid\_t pid = fork();  if (pid == -1) {  error("Can't fork process");  }  What do you think goes here?  if (!pid) {  if ( ) { error("Can't redirect Standard Output");  }  if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",  phrase, NULL, vars) == -1) { error("Can't run script");  } }  return 0; }  **newshound2.c** |

***you are here*** **435**

***standard output redirected***

3 File stories txt

|  |  |
| --- | --- |
| This is a program that saves the output of the *rssgossip.py* script  into a file called *stories.txt*. It’s similar to the newshound  program, except it searches through a single RSS feed only. Using what you’ve learned about the descriptor table, you were to find the missing line of code that will redirect the **Standard Output** of the child process to the *stories.txt* file.  int main(int argc, char \*argv[]) {  char \*phrase = argv[1];  char \*vars[] = {"RSS\_FEED=http://www.cnn.com/rss/celebs.xml", NULL}; FILE \*f = fopen("stories.txt", "w"); This opens stories.txt for writing.  if (!f) { Iff was zero, we couldn’t open the file.  error("Can't open stories.txt"); }  pid\_t pid = fork(); if (pid == -1) {  error("Can't fork process");  i}f (!pid) { ehidclero. iiii#1  if ( dup2(fileno(f), 1) == -1 ) {  error("Can't redirect Standard Output"); }  if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",  phrase, NULL, vars) == -1) { error("Can't run script");  } }  return 0; } | |
| **Did you get the right answer?** The program will change the descriptor table in the child script to look like this:  That means that when the *rssgossip.py* script sends data to the Standard Output, it should appear in the *stories.txt* file. | **newshound2.c**  # Data Stream   |  |  | | --- | --- | |  |  | |  |  | |  |  | |  |  | |  |  |   0 The keyboard  1 File stories.txt  2 The screen |

.

**436 *Chapter 10***

***interprocess communication***

 ~~Test Drive~~

This is what happens when the program is compiled and run:

This runs the program.

This displays the contents

of the stories.txt file.

If you’re on a Windows machine, you’ll need to be running Cygwin.

个

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./newshound2 'pajama death' > cat stories.txt**  **Pajama Death ex-drummer tells all.**  **New Pajama Death album due next month.** |

The stories are

saved in the

stories.txt file.

**What** **happened?**

When the program opened the *stories.txt* file with fopen(), the operating system registered the file fin the descriptor table. fileno(f) was the descriptor number it used. The dup2() function set the Standard Output descriptor (1) to point to the same file.



No data in the

file? WTF?!? 

个

Where’s The Facts?

I think there might be a

problem with the program.

See, I just tried the same thing, but on **my** machine the file was empty. So what happened?

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./newshound2 'pajama death' > cat stories.txt**  **>** |

|  |
| --- |
| Assuming you’researching for stories that exist on the feed, why was *stories.txt* empty after the program finished? |

***you are here*** **437**

***hey, wait***

Sometimes you need to wait…

The newshound2 program fires off a separate process to run

the *rssgossip.py* script. But once that child process gets created, it’s

**independent** of its parent. You could run the newshound2

program and still have an empty *stories.txt*, just because the *rssgossip.py* isn’t finished yet. That means the operating system has to give you some way of **waiting** for the child process to complete.

Can you save these stories to the file?

Might take a while …

That’s OK, I can wait.

**newshound**

The waitpid() function

The **waitpid()** function won’t return until the child process dies. That means you can add a little code to your program so that it

won’t exit until the *rssgossip.py* script has stopped running:



**child process**

You need to include the sys/wait.h

header.

at the end of the

This new code goes

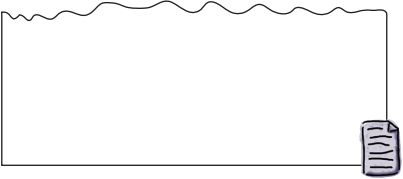
newshound2 program.

**438 *Chapter 10***

This variable is used to store information about the process.



**#include <sys/wait.h>**



This is a pointer to an int.

  You can add options here.

The process ID

**newshound2.c**



int pid\_status;

if (**waitpid(pid, &pid\_status, 0)** == -1) {

error("Error waiting for child process");

}

return 0; }



Close

waitpid() Up

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| waitpid( | pid, | pid\_status, | options | ) |

|  |  |
| --- | --- |
| waitpid() takes three parameters: | |
| ¥  ¥  ¥ | **pid**  This is the process ID that the parent process was given when it forked the child.  **pid\_status**  This will store *exit information* about the process. waitpid() will update it,so it needs to be a pointer.  **options**  There are several options you can pass to waitpid(), and typing  man waitpid will give you more info. If you set the options to **0**, the function waits until the process finishes. |
| What’s the status?  When the waitpid() function has finished waiting, it  stores a value in pid\_status that tells you how the process did. To find the *exit status* of the child process, you’ll have  to pass the pid\_status value through a macro called **WEXITSTATUS()**:  if (**WEXITSTATUS(pid\_status)**)  If the exit status is not zero puts("Error status non-zero");  Why do you need the macro? Because the pid\_status  contains several pieces of information, and only the first 8 bits represent the exit status. The macro tells you the value of just those 8 bits. | |

***you are here*** **439**

***test drive***

 ~~Test Drive~~

Now, when you run the newshound2 program, it checks that the *rssgossip.py* script finishes before newshound2 itself ends:

The stories.txt

file now contains

the stories

as soon as

newshound2 is run.

story

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./newshound2 'pajama death' > cat stories.txt**  **Pajama Death ex-drummer tells all.**  **New Pajama Death album due next month.** |



That’s great. Now I’ll never miss

another again

.



Adding awaitpid() to the program was easy to do and it made the program more reliable. Before, you couldn’t

be sure that the subprocess had finished writing, and that meant there was noway you could use the newshound2 program as a proper tool. You couldn’t use it in scripts and you couldn’t create a GUI frontend for it.

Redirecting input and output, and making processes wait

for each other, are all simple forms of **interprocess**

**communication**. When processes are able to cooperate— by sharing data or by waiting for each other—they become much more powerful.

|  |  |
| --- | --- |
| ■ exit() is a quick way of ending a program.  ■ All open files are recorded in the descriptor table.  ■ You can redirect input and output by changing the descriptor table. | ■ fileno()will find a descriptor in the table.  ■ dup2() can be used to change the descriptor table.  ■ waitpid()will wait for processes to finish. |

**440 *Chapter 10***

***interprocess communication***

~~bt"iesxions~~

Q: **Does exit()end the program faster than just returning from**

**main()?**

No. But if you call exit(), you

A:

don’t need to structure your code to get

back to the main()function. As soon as you call exit(), your program is dead.

Q: **Should I check for –1 when I call exit(), in case it doesn’t work?**

No. exit()doesn’t return a value, because exit() never fails. exit() is the only function that is guaranteed

A:

never to return a value and never to fail.

Q: **Is the number I pass to exit() the exit status?**

A:

Yes.

**Are the Standard Input, Output, and Error always in slots 0, 1, and 2 of the descriptor table?**

Q:

A:

Yes, they are.

**So, if I open a new file, it is**

Q:

**automatically added to the descriptor table?**

A:

Yes.

Q: **Is there a rule about which slot it gets?**

New files are always added to the available slot with the lowest number. So, if slot number 4 is the first available one, that’s the one your new file will use.

A:

Q: **How big is the descriptor table?**

It has slots from 0 to 255.

A:

Q:

A:

Q:

A:

**The descriptor table seems kinda complicated. Why is it there?**

Because it allows you to rewire

the way a program works. Without the

descriptor table, redirection isn’t possible.

**Is there a way of sending data to the screen without using the Standard Output?**

On some systems. For example, on Unix-based machines, if you open

*/dev/tty*, it will send data directly to the terminal.

Q: **Can I use waitpid()to wait for any process? Or just the processes I started?**

You can use waitpid()to wait for any process.

A:

Q:**Why isn’t the pid\_status in waitpid(..., &pid\_status, ...)just an exit status?**

Because the pid\_status contains other information.

A:

Q: **Such as?**

A: For example, WIFSIGNALED (pid\_status)will be false if

a process ended naturally, or true if something killed it off.

**How can an integer variable like pid\_status contain several pieces of information?**

Q:

A:

Q:

It stores different things in different bits. The first 8 bits store the exit status. The other information is stored in the other bits.

**So, if I can extract the first 8 bits of the pid\_status value, I don’t have to use WEXITSTATUS()?**

A: It is always best to use

WEXITSTATUS(). It’s easier to read and it will work on whatever the native

int size is on the platform.

**Why is WEXITSTATUS() in uppercase?**

Q:

A:

Because it is a macro rather than a function. The compiler replaces macro statements with small pieces of code at runtime.

***you are here*** **441**

***don’t be a stranger***

Stay in touch with your child

You’ve seen how to run a separate process using exec() and fork(), and you knowhow to redirect the output of a child process into a file. But what if you want to listen to a child

process directly? Is that possible? Rather than waiting for a

child process to send all of its data into a file and then reading the file afterward, is there some way to start a process running and read the data it generates ***in realtime***?

Reading story links from rssgossip

As an example, there’san option on the *rssgossip.py* script that allows you to display the URLs for any stories that it finds:

-u tells the script to include story links.

The URL line

begins with a

tab character.

This is the URL for the story.

|  |
| --- |
| File Edit Window Help |
| **> python rssgossip.py -u 'pajama death'**  **Pajama Death ex-drummer tells all.**  [**http://www.rock-news.com/exclusive/24.html**](http://www.rock-news.com/exclusive/24.html) **New Pajama Death album due next month.**  [**http://www.rolling-stone.com/pdalbum.html**](http://www.rolling-stone.com/pdalbum.html) |

Now, you *could* run the script and save its output to a file, but that would be slow. It would be much better if the parent and child process could talk to each other while the child process is still running.



Since I created

you, you never write, you never phone …

Parent process

**442 *Chapter 10***

Whatever.



process

Child

***interprocess communication***

Connect your processes with pipes

You’ve already used something that makes live connections between processes: pipes.

grep filters the

The two processes are connected with a pipe.

output of the script.

rssgossip.py sends its  output into the pipe.

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **python rssgossip.py -u 'pajama death' | grep 'http'** [**http://www.rock-news.com/exclusive/24.html**](http://www.rock-news.com/exclusive/24.html)  [**http://www.rolling-stone.com/pdalbum.html**](http://www.rolling-stone.com/pdalbum.html) |

Pipes are used on the command line to connect the **output** of one process with the **input** of another process. In the example here, you’rerunning the *rssgossip.py* script manually and then

passing its output through a command called **grep**. The grep command finds all the lines containing **http**.

Piped commands are parents and children

Whenever you *pipe* commands together on the command line, you are actually connecting them together as parent and child processes. So, in the above example, the grep command is

the **parent** of the *rssgossip.py* script.

**1 The command line creates the parent process.**

**2 The parent process forks the rssgossip.py**

**script in a child process.**

**3 The parent connects the output of the child**

**with the input of the parent using a pipe.**

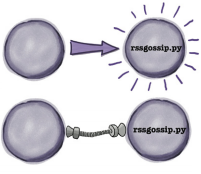
**4 The parent process execs the grep command.**

Pipes are used a lot on the command line to allow users to

connect processes together. But what if you’re just using C

code? How do you connect a pipe to your child process so that you can read its output as soon as it’s generated?







***you are here*** **443**

***pipe()***

Case study: opening stories in a browser

Let’s say you want to run the *rssgossip.py* script and then

open the stories it finds in a web browser. Your program

I want a program

that opens stories in my browser as soon as they’re found.

will run in the parent process and *rssgossip.py* will run in the child. You need to create a pipe that connects the output of *rssgossip.py* to the input of your program.

**But** **how** **do** **you** **create** **a** **pipe?**

pipe() opens two data streams

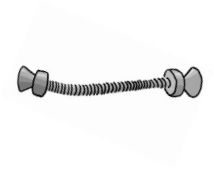
Because the child is going to send data to the parent, you

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | Standard input |
| 1 | Standard output |
| 2 | Standard error |
| 3 | Read-end of the pipe |
| 4 | Write-end of the pipe |

need a pipe that’s connected to the Standard Output of the child and the Standard Input of the parent. You’ll create

the pipe using the **pipe()** function. Remember how we said that every time you open a data stream to something like a file, it gets added to the descriptor table? Well, that’s exactly what the pipe() functions does: it creates two

This is fd[0].  This is fd[1]. 

connected streams and adds them to the table. Whatever is written into one stream can be read from the other.

Calling pipe() creates these two descriptors.

…can be read from here.

Whatever is written here…

When pipe() creates the two lines in the descriptor table, it will store their file descriptors in a two-element array:

The descriptors will be

|  |  |
| --- | --- |
| the pipe() function.  You pass the name  of the array to | stored in this array.  int fd[2];  if (pipe(fd) == -1) {  error("Can't create the  } |

The pipe() command creates a pipe and tells you two descriptors: fd[1] is the descriptor that **writes** to the pipe, and fd[0] is the descriptor that **reads** from the pipe. Once you’ve got the descriptors, you’ll need to use them in the parent and child processes.

pipe");

fd[1] writes to the pipe; fd[0] reads from it.

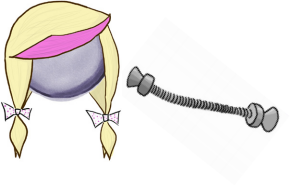
**444 *Chapter 10***

***interprocess communication***

In the child

In the child process, you need to **close** the fd[0] end of the pipe and then change the child process’s Standard Output to point to the same stream as descriptor fd[1].

This will close the read end of the pipe.

read from the pipe.  **close(fd[0]);**The child then connects the write

This is fd[0],

the read end

of the pipe. 

The child won’t 

**dup2(fd[1], 1);** end to the Standard Output.

The child won’t read

from the pipe… …but will

write.

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | Standard input |
| 1 | Standard output Write-end of the pipe |
| 2 | Standard error |
| 3 | Read-end of the pipe |
| 4 | Write-end of the pipe |

This is fd[1], the write end

of the pipe.

That means that everything the child sends to the Standard Output will be written to the pipe.

In the parent

In the parent process, you need to close the fd[1] end

of the pipe (because you won’t be writing to it) and then

redirect the parent process’s Standard Input to read its data

from the same place as descriptor fd[0]:  fd[0] is the read end of the pipe.

The parent connects  **dup2(fd[0], 0);**

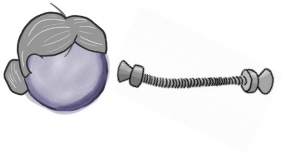
ettthe **close(fd[1]);** This will close the write end of the pipe.

The parent will read

from the  pipe…

…but won’t write

|  |  |
| --- | --- |
| # | Data Stream |
| 0 | Standard input Read-end of the pipe |
| 1 | Standard output |
| 2 | Standard error |
| 3 | Read-end of the pipe |
| 4 | Write-end of the pipe |

Everything that the child writes to the pipe will be read through the Standard Input of the parent process.

***you are here*** **445**

***ready-bake code***

Opening a web page in a browser

Your program will need to open up a web page using the

machine’s browser. That’s kind of hard to do, because

different operating systems have different ways of talking to programs like web browsers.

Fortunately, the out-of-work actors have hacked together

some code that will open web pages on most systems. It looks like they were in a rush to go do something else, so they’ve

put together something pretty simple using system():

|  |  |  |
| --- | --- | --- |
|  | Ready-Bake Code | void open\_url(char \*url) {  char launch[255];  sprintf(launch, "cmd /c |



This will open a web page on Windows.

↓

start %s", url);

This will open a web page on Linux.

}

system(launch);

sprintf(launch, "x-www-browser '%s' &", url); system(launch);

sprintf(launch, "open '%s'", url); system(launch); 

This will open a web page on the Mac.

The code runs **three separate commands** to open a

URL: that’s one command each for the Mac, Windows, and Linux. Two of the commands will always fail, but as long as the third command works, that’ll be fine.

Piste

Go Off

|  |
| --- |
| Think you can write better code than the out-of-work actors?  Then why not rewrite the code to use fork() and exec() for your favorite operating system? |

**446 *Chapter 10***

***interprocess communication***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| It looks like most of the program is already written. All you need to do is complete the code that connects the *parent* and *child* processes to a pipe. To save space, the #include lines and the error() and open\_url() functions have been removed. Remember, in this program the *child* is going to talk to the *parent*, so make sure that pipe’s connected the right way!  int main(int argc, char \*argv[]) You might want to replace this  { with another RSS newsfeed.  char \*phrase = argv[1];  char \*vars[] = {"RSS\_FEED=http://www.cnn.com/rss/celebs.xml", NULL}; int fd[2]; This array will store the descriptors for your pipe. | | | | |
| pid\_t pid = fork();  if (pid == -1) {  error("Can't fork  process");  Are you parent or child? What code goes in these lines?  }  if (!pid) { | | | | Create your pipe here. |
| if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py", "-u", phrase, NULL, vars) == -1) { | | | | |
| error("Can't run script");  }Are you in the parent or the child here? } What do you need to do to the pipe? | | “-u” tells the script to  URLs for the stories. | display | |
| char line[255];  while (fgets(line, 255, )) {  If the line starts with a tab… …then it’s a URL.  if (line[0] == '\t') open\_url(line + 1);  } 个 | | | | What needs  to go here?  What will you  read from? |
| return 0; } | “line + 1” is the string starting  after the tab character. | | | |
| **news\_opener.c** | | | | |

***you are here*** **447**

***pipe connected***

|  |  |
| --- | --- |
|  | It looks like most of the program is already written. You were to complete the code that connects  the *parent* and *child* processes to a pipe. To save space, the #include lines and the error() and open\_url() functions have been removed. |
| {  }  char \*phrase = argv[1];  char \*vars[] = {"RSS\_FEED=http://www.cnn.com/rss/celebs.xml", NULL}; int fd[2]; This will create the pipe and store its descriptors in fd[0] and fd[1].  if (pipe(fd) == -1) {  Need to check that return in case we can’t create the  error(“Can’t create the pipe”);  code, pipe.  · } ·  pid\_t pid = fork(); if (pid == -1) {  error("Can't fork process"); }  You’re in the child process here.  if (d!(f)d[{1], 1); This will set the Standard Output to the write end of the pipe.  · close(fd[0]);  The child won’t read from the pipe, so we’ll close the read end.  if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py", "-u", phrase, NULL, vars) == -1) {  error("Can't run script");  } You’re in the parent process down here.  d}up2(fd[0], 0); · This will redirect the Standard Input to the read end of the pipe.  close(fd[1]); · This will close the write end of the pipe, char line[255]; because the parent won’t write to it.  while (fgets(line, 255, stdin · )) {  if (line[0] == '\t')  open\_url(line + 1); You’re reading from the You could also  } Standard Input, because  have put fd[0].  return 0; that’s connected to the  pipe.  int main(int argc, char \*argv[]) | |
| **news\_opener.c** | |

***interprocess communication***

 ~~Test Drive~~

When you compile and run the code, this happens:

|  |
| --- |
| File Edit Window Help ReadAllAboutIt |
| **> ./news\_opener 'pajama death'** |

**That’s** **great.** **It** **worked.**

The news\_opener program ran the *rssgossip.py* in a

separate process and told it to display URLs for each story it found. All of the output of the screen was redirected through a **pipe** that was connected to the news\_opener parent

process. That meant the news\_opener process could search for any URLs and then open them in the browser.

Pipes area great way of connecting processes together.

Now, you have the ability to not only **run** processes and

**control** their environments, but you also have away of

个

**capturing their output**. That opens up a huge amount of functionality to you. Your C code can now use and control ***any program*** that you can use from the command line.

The program opens all

the news stories it can

find in the browser.

Go OffPiste

|  |
| --- |
| Now that you knowhow to control *rssgossip.py*, why not try controlling some of these programs? You can get all of them for Unix-style machines or any Windows machine using Cygwin:  **curl/wget**  These programs let you talk to web servers. If you call them from C code, you can write  programs that can talk to the Web.  **mail/mutt**  These programs let you send email from the command line. If they’re on your machine, it means your C programs can send mail too.  **convert**  This command can convert one image format to another image format. Why not create a C program that outputs SVG charts in text format, and then use the convert command to create PNG images from them? |

***you are here*** **449**

***no dumb questions***

~~b"iesioms~~

**Is a pipe a file?**

Q:

A: It’s up to the operating system how it creates pipes, but pipes created with the pipe()function are not normally files.

Q: **So pipes *might* be files?**

It is possible to create pipes based on files, which are normally called *named pipes* or *FIFO* (first-in/first-out) files.

A:

Q:

A:

**Why would anyone want a pipe that uses a file?**

Pipes based on files have names.

That means they are useful if two

processes need to talk to each other and they are not parent and child processes. As long as both processes know the name of the pipe, they can talk with it.

Q: **Great! So how do I use named pipes?**

Using the mkfifo()system call.

A:

Q:

A:

Q:

A:

For more information, see [*http://tinyurl.com/cdf6ve5*](http://tinyurl.com/cdf6ve5).

**If most pipes are not files, what are they?**

Usually, they are just pieces of

memory. Data is written at one point and read at another.

**What happens if I try to read from a pipe and there’s nothing in there?**

Your program will wait until something is there.

Q: **How does the parent know when the child is finished?**

When the child process dies, the pipe is closed and the fgets()command

A:

receives an end-of-file, which means the fgets()function returns 0, and the loop ends.

**Can parents speak to children?**

Q:

A:Absolutely. There is no reason why you can’t connect your pipes the other way around, so that the parent sends data to

the child process.

**Can you have a pipe that works in both directions at once? That way, my parent and child processes could have a two-way conversation.**

Q:

A: No, you can’t do that. Pipes always work in only one direction. But you can

create two pipes: one from the parent to the child, and one from the child to the parent.

|  |
| --- |
| ■ Parent and child processes can ■ You can redirect Standard Input and communicate using pipes. Output to the pipe.  ■ The pipe()function creates a ■ The parent and child processes use pipe and two descriptors. different ends of the pipe.  ■ The descriptors are for the read and write ends of the pipe. |

**450 *Chapter 10***

***interprocess communication***

The death of a process

You’ve seen how processes are created, how their

#include <stdio.h>

int main() {

environments are configured, and even how processes talk to each other. But what about how processes die? For example, if your program is reading data from the keyboard and the user hits Ctrl-C, the program stops running.

How does that happen? You can tell from the output

char name[30];

printf("Enter your name: "); fgets(name, 30, stdin);

printf("Hello %s\n", name); return 0;

that the program never got as far as running the second printf(), so the Ctrl-C didn’t just stop the fgets()

command. Instead, the whole program just stopped in its

tracks. Did the operating system just unload the program? Did the fgets() function call exit()? What happened?

}

File Edit Window Help

|  |
| --- |
| **> ./greetings**  **Enter your name: ^C >** |

The O/S controls your program with signals

The magic all happens in the operating system. When you call the fgets() function, the operating system reads the data from the keyboard, and when it sees the user hit Ctrl-C, sends an interrupt signal to the program.

If you press Ctrl-C, the program stops running. But why?

Hey! He hit Ctrl-C. Run your interrupt handler.

Someone hits Ctrl-C.





**Keyboard**

|  |  |
| --- | --- |
| **Ctrl-C** ▲  The operating  system sends an interrupt signal.  **operating system** | **Interrupt**  **signal**  The process runs its  default interrupt  handler and calls exit().  **process** |

A signal is just a short message: a single integer value. When the signal arrives, the process has to stop whatever it’s doing and go deal with the signal. The process looks at a table of

*signal mappings* that link each signal with a function called the **signal handler**. The default signal handler for the interrupt signal just calls the exit() function.

So, why doesn’t the operating system just kill the program? Because the signal table lets you run your ***own code*** when your process receives a signal.

Signal mappings



This is the

interrupt signal.

SIGINT has the value 2.



|  |  |
| --- | --- |
| Signal | Handler |
| SIGURG | Do nothing |
| SIGINT | Call exit() |

The default handler calls exit().

***you are here*** **451**

***sigaction()***

Catching signals and running your own code

Sometimes you’ll want to run your own code if someone interrupts your program. For example, if your process has files or network

connections open, it might want to close things down and tidy up before exiting. But how do you tell the computer to run your code when it sends you a signal? You can do it with **sigaction**s.

A sigaction is a function wrapper

A sigaction is a struct that contains a pointer to a function. sigactions are used to tell the operating system which function it should call when a signal is sent to a process. So, if you have a

function called diediedie() that you want the operating system to call if someone sends an *interrupt* signal to your process, you’ll

need to wrap the diediedie() function up as a sigaction.

This is how you create a sigaction: Create a new action.

**struct sigaction action;**

These are some

additional flags.

You can just set

This is the name of the function

you want the computer to call.

The function that the sigaction

**action.sa\_handler = diediedie;**

**sigemptyset(&action.sa\_mask);**  **action.sa\_flags = 0;**

them to zero.

The mask is a way of filtering the

signals that the sigaction will handle.

wraps is called a handler.

The function wrapped by a sigaction is called the **handler**, because it will be used to deal with (or *handle*) asignal that’s sent to it. If you want to create a handler, it will need to be written in a certain way.

You’ll usually want to use an empty mask, like here.

All handlers take signal arguments

|  |
| --- |
| **Be** **careful** **when** **writing** **to**  **Standard** **Output** **and** **Error** **in**  **handler** **functions.**  *Even though the example code you’ll use will display text on*  *the Standard Output, be careful about doing that in more complex programs. Signals can arrive*  *because something bad has*  *happened to the program. That might mean that Standard Output isn’t available, so be careful.* |

Signals are just integer values, and if you create a custom handler function, it will need to accept an int argument, like this:

void diediedie(**int sig**) This is the signal number

{  the handler has caught. puts ("Goodbye cruel world....\n");

exit(1);

}

Because the handler is passed the number of the signal, you can *reuse* the same handler for several signals. Or, you can have a

separate handler for each signal. How you choose to program it is up to you.

Handlers are intended to be short, fast pieces of code. They should do *just enough* to deal with the signal that’s been received.

***interprocess communication***

This is a function that will make it a little easier to

register functions as signal handlers:

sigactions are registered with sigaction()

Once you’ve create a sigaction, you’ll need to tell the operating system about it. You do that with the **sigaction()** function:

**sigaction(signal\_no, &new\_action, &old\_action);**

sigaction() takes three parameters:

¥

¥

¥

**The signal number.**

The integer value of the signal you want to handle. Usually, you’ll pass one of the standard signal symbols, like SIGINT or SIGQUIT.

**The new action.**

This is the **address** of the new sigaction you want to register. **The old action.**

If you pass a pointer to another sigaction, it will be filled with

details of the *current* handler that you’re about to replace. If you don’t care about the existing signal handler, you can set this to NULL.

You’ll find out more

about the standard

signals in a while.

The sigaction() function will return –1 if it fails and will also set the errno function. To keep the code short, some of the code you’ll see in this book will skip checking for errors, but you should ***always*** check for errors in your own code.

The signal number

Ready-Bake Code

**int catch\_signal(int sig, void**

**{**

**struct sigaction action;** 

A pointer to the handler function

↓

**(\*handler)(int))**

Create an action.

Use an empty mask.

**action.sa\_handler = handler;** <— **sigemptyset(&action.sa\_mask);**

**action.sa\_flags = 0;**

Set the action’s handler to

the handler function that

was passed in.

**return sigaction (sig, &action, NULL);**

**}**

Return the value of sigaction(),

so you can check for errors.

This function will allow you to set a signal handler by calling

catch\_signal() with a signal number and a function name:

**catch\_signal(SIGINT, diedieie)**

***you are here*** **453**

***catching signals***

Rewriting the code to use a signal handler

You now have all the code to make your program do something if someone hits the Ctrl-C key:

Handlers

have void

return types.

|  |  |
| --- | --- |
| #include <stdio.h>  The operating system passes  the signal to the handler.  #include <signal.h><— You need to include the signal.h header. #include <stdlib.h>  This our new signal handler.  void diediedie(int sig)k一 {  puts ("Goodbye cruel world....\n"); exit(1);  } This is the function to register a handler.  int catch\_signal(int sig, void (\*handler)(int)) {  struct sigaction action;  action.sa\_handler = handler; sigemptyset(&action.sa\_mask);  action.sa\_flags = 0;  return sigaction (sig, &action, NULL); }  SIGINT means we are capturing This sets the interrupt  int main() the interrupt signal. the handle\_interrupt()  {  if (catch\_signal(SIGINT, handle\_interrupt) == -1) {  fprintf(stderr, "Can't map the handler"); exit(2);  }  char name[30];  printf("Enter your name: "); fgets(name, 30, stdin);  printf("Hello %s\n", name); return 0;  } | handler to function. |

The program will ask for the user’s name and then wait for her to type. But if instead of typing her name, the user just hits the Ctrl-C key, the operating system will automatically send the process an *interrupt signal* (SIGINT). That interrupt signal will be handled by the sigaction

that was registered in the catch\_signal() function. The

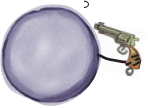
sigaction contains a pointer to the diediedie() function. This

will then be called, and the program will display a message and exit().

**454 *Chapter 10***

***interprocess communication***

 ~~Test Drive~~

When you run the new version of the program and press Ctrl-C, this happens:

Goodbye,

cruel world …

|  |
| --- |
| File Edit Window Help |
| **> ./greetings**  **Enter your name: ^CGoodbye cruel world.... >** |

The operating system received the Ctrl-C and sent a SIGINT signal to the process, which then ran***Dour*** handle\_interrupt() function.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| There area bunch of different signals the operating system can send to your process. Match each signal to its cause.   |  |  | | --- | --- | | SIGINT | The process was interrupted. | | SIGQUIT | The terminal window changed size.  The process tried to access illegal memory. | | SIGFPE |  | |  | Someone just asked the kernel to kill the | | SIGTRAP | process. | | SIGSEGV | The process wrote to a pipe that nothing’s reading. | | SIGWINCH | Floating-point error. | | SIGTERM | Someone asked the process to stop and dump the memory in a core dump file. | | SIGPIPE | The debugger asks where the process is. | |

***you are here*** **455**

***purpose found***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ≤△l“T、△n  There area bunch of different signals the operating system can send to your process. You were to match each signal to its cause.  SIGINT ——— The process was interrupted.   |  |  | | --- | --- | | SIGQUIT | The terminal window changed size.  The process tried to access illegal memory. | | SIGFPE |  | |  | Someone just asked the kernel to kill the | | SIGTRAP | process. | | SIGSEGV | The process wrote to a pipe that nothing’s reading. | | SIGWINCH | Floating-point error. | | SIGTERM | Someone asked the process to stop and dump the memory in a core dump file. | | SIGPIPE | The debugger asks where the process is. | |

~~btiexiom~~

Q: **If the interrupt handler didn’t call exit(), would the program still have ended?**

A:

No.

Q: **So, I could write a program that completely ignores interrupts?**

You could, but it’s not a good idea. In general, if your

A:

program receives an error signal, it’s best to exit with an error, even if you run some of your own code first.

**456 *Chapter 10***

***interprocess communication***

Use kill to send signals

If you’ve written some signal-handling code, how do you test

it? Fortunately, on Unix-style systems, there’sa command  called **kill**. It’s called kill because it’s normally used to kill

off processes, but in fact, kill just sends a signal to a process.

By default,the commandsends a SIGTERM signal to the process, but you can use it to send any signal you like.

To try it out, open *two terminals*. In one terminal, you can run your program. Then, in the second terminal, you can send signals to your program with the kill command:

Including Cygwin on Windows

ps displays your

current processes. 

This sends SIGTERM

to the program. 

This sends SIGINT刁

to the program.

This sends SIGSEGV to the program.

This is the program we want to

send signals to. 78222 is the

process ID.

|  |
| --- |
| File Edit Window Help |
| **> ps**  **77868 ttys003 0:00.02 bash**  **78222 ttys003 0:00.01 ./testprog**  **> kill 78222**  **> kill -INT 78222 > kill -SEGV 78222 > kill -KILL 78222** |

This sends SIGKILL, which can’t be ignored.

Each of these kill commands will send signals to the process

and run whatever handler the process has configured. The

exception is the **SIGKILL** signal. The SIGKILL signal can’t be caught by code, and it can’t be ignored. That means if you

have a bug in your code and it is ignoring every signal, you can

SIGSTOP can’t be ignored either. It’s used to pause your process.

**always** stop the process with kill -KILL.

Send signals with raise()

Sometimes you might want a process to send a signal to itself, which you can do with the raise() command.

kill -KILL <pid> will always kill your program.

**raise(SIGTERM);**

Normally, the raise() command is used inside your own custom signal handlers. It means your code can receive a

signal for something minor and then choose to raise a more serious signal.

This is called **signal escalation**.

***you are here*** **457**

***smell the coffee***

Sending your code awake-up call

The operating system sends signals to a process when something has happened that the process needs to know about. It might be that the user has tried to interrupt the process, or someone has tried to kill it, or even that the process has tried to do something it shouldn’t have, like trying to access a restricted piece of memory.

But signals are not just used when things go wrong. Sometimes a

process might actually want to generate its own signals. One example of that is the **alarm signal**, **SIGALRM**. The alarm signal is usually created by the process’s **interval timer**. The interval timer is like

Tick, tick, tick, just a couple of minutes …

an alarm clock: you set it for sometime in the future, and in the meantime your program can go and do something else:

i iilimer →

**alarm(120);**

do\_important\_busy\_work();

 Calling alarm(120) sets

the alarm for 120

Meanwhile, your code  does something else.

do\_more\_busy\_work();

seconds in the future.

But eventhough your program is busy doing other things, the timer is still running in the background. That means that when the 120

seconds are up …

…the timer fires a SIGALRM signal

When a process receives a signal, it **stops doing everything else** and handles the signal. But what does a process do with an alarm

signal by default? It ***stops the process***. It’s really unlikely that you would ever want atimer to kill your program for you, so most of the time you will set the handler to do something else:

|  |
| --- |
| **Don’t** **use**  **alarm()** **and** **sleep()** **at**  **the** **same** **time.**  *The* sleep() *function puts your program to sleep for a few seconds, but it works by using the same interval timer as the* alarm()  *function, so if you try to use the two functions at the same time, they will interfere with each other.* |

catch\_signal(SIGALRM, pour\_coffee); alarm(120);

This will catch the 

liieated

earlier.

Ah, sweet,

sweet coffee …

Brrriiiiiiinnnng!



Alarm signals let you **multitask**. If you need to run a particular job

every few seconds, or if you want to limit the amount of time you spend doing a job, then alarm signals area great way of getting a program to ***interrupt itself***.

**458 *Chapter 10***

***interprocess communication***

|  |  |  |
| --- | --- | --- |
| Resetting Ignoring Signals Up | | |
| You’ve seen how to set custom signal handlers, but  what if you want to restore the default signal handler? Fortunately, the *signal.h* header has a special symbol  **SIG\_DFL**, which means ***handle it the default way***.  **catch\_signal(SIGTERM, SIG\_DFL);**  Also, there’s another symbol, **SIG\_IGN**, that tells the process to completely **ignore** a signal.  **catch\_signal(SIGINT, SIG\_IGN);**  But you should be *very careful* before you choose to ignore a signal. Signals are an important way of  controlling—and stopping—processes. If you ignore them, your program will be harder to stop. |  | Ctrl-C? Talk  to the hand; I’m doing nothing. |
|  |

~~btiexioms~~

Q: **Can I set an alarm for less than a second?**

A:

Yes, but it’s a little more complicated. You need to use a

different function called setitimer(). It lets you set the

process’s interval timer directly in either seconds or fractions of a second.

Q: **How do I do that?**

A:

Go to [*http://tinyurl.com/3o7hzbm* for](http://tinyurl.com/3o7hzbmfor) more details.

**Why is there only one timer for a process?**

Q:

A: The timers have to be managed by the operating system kernel, and if processes had lots of timers, the kernel would go slower and slower. To prevent this from happening, the operating system limits each process to one timer.

Q: **Timers let me multitask?! Great, so I can use them to do lots of things at once?**

No. Remember, your process will always stop whatever it’s doing when it handles a signal. That means it is still only doing one thing at a time. You’ll see later how you can really make your code do more than one thing at a time.

A:

Q:

**What happens if I set one timer and it had already been set?**

A: Whenever you call the alarm()function, you reset the timer. That means if you set the alarm for 10 seconds, then a

moment later you set it for 10 minutes, the alarm won’tfire until 10 minutes are up. The original 10-second timer will be lost.



and Close

OK, so if I receive TERM signal, I

should just exit() like before …

***you are here*** **459**

***exercise***

What should

happen once

the score is

displayed?

|  |  |
| --- | --- |
| This is the source code for a program that tests the user’s math skills. It asks the user to work the answer to a simple multiplication problem and keeps track of how many answers he got  right.The program will keep running forever, unless:  1. The user presses Ctrl-C, or  2. The user takes more than **five seconds** to answer the question.  When the program ends, it will display the final score and set the exit status to 0.  #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <time.h>  #include <string.h> #include <errno.h> #include <signal.h>  int score = 0;  void end\_game(int sig) { | |
|  | printf("\nFinal score: %i\n", score); |
| }  int catch\_signal(int sig, void (\*handler)(int)) {  struct sigaction action;  action.sa\_handler = handler; sigemptyset(&action.sa\_mask); action.sa\_flags = 0;  return sigaction (sig, &action, NULL); } | |

**460 *Chapter 10***

***interprocess communication***

|  |  |
| --- | --- |
| void times\_up(int sig) | |
| {  } | puts("\nTIME'S UP!");  raise( );  个  Raise what? |
| void error(char \*msg) {  functions do?  you get different random numbers each time.  fprintf(stderr, "%s: %s\n", msg, strerror(errno)); exit(1);  }  int main() {  What will  the signal()  catch\_signal(SIGALRM, ); catch\_signal(SIGINT, ); srandom (time (0));  This makes sure  while(1) {  int a = random() % 11; a and b will be random numbers from 0 to 10.  int b = random() % 11;  char txt[4];  Hmmm…what line is missing? Need to check the spec… | |
| printf("\nWhat is %i times %i? ", a, b); fgets(txt, 4, stdin);  int answer = atoi(txt); if (answer == a \* b)  score++; else  printf("\nWrong! Score: %i\n", score); }  return 0; } | |

***you are here*** **461**

***exercise solved***

You need to set

the exit status

to 0 and stop.

|  |  |
| --- | --- |
| This is the source code for a program that tests the user’s math skills. It asks the user to work the answer to a simple multiplication problem and keeps track of how many answers he got  right.The program will keep running forever, unless:  1. The user presses Ctrl-C, or  2. The user takes more than **five seconds** to answer the question.  When the program ends, it will display the final score and set the exit status to 0.  #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <time.h>  #include <string.h> #include <errno.h>  #include <signal.h> int score = 0;  void end\_game(int sig) | |
| {  } | printf("\nFinal score: %i\n", score);  exit(0); · |
| int catch\_signal(int sig, void (\*handler)(int)) {  struct sigaction action;  action.sa\_handler = handler; sigemptyset(&action.sa\_mask); action.sa\_flags = 0;  return sigaction (sig, &action, NULL); } | |

**462 *Chapter 10***

***interprocess communication***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| void times\_up(int sig) {  puts("\nTIME'S UP!");  raise( · SIGINT · ); } 个  i Iilwo \_pg  void error(char \*msg) {  fprintf(stderr, "%s: %s\n", msg, strerror(errno)); exit(1);  } | | | | |
| int main() { | | |  | the handlers.  The signal()  functions set |
| This makes sure  you get different random numbers  each time. | | catch\_signal(SIGALRM, times\_up · ); catch\_signal(SIGINT, end\_game · ); srandom (time (0));  while(1) {  int a = random() % 11; int b = random() % 11; char txt[4]; |
| Set the alarm to fire in 5 seconds.  个  As long as you go through  the loop in less than 5 seconds, the timer will be reset and it will never fire.  } | alarm(5); ·  printf("\nWhat is %i times %i? ", a, b); fgets(txt, 4, stdin);  int answer = atoi(txt); if (answer == a \* b)  score++; else  printf("\nWrong! Score: %i\n", score); }  return 0; | | | |

***you are here*** **463**

***test drive***

 ~~Test Drive~~

To see if the program works, you need to run it a couple of times.

Test 1: hit Ctrl-C

|  |
| --- |
| File Edit Window Help |
| **> ./math\_master**  **What is 0 times 1? 0**  **What is 6 times 1? 6**  **What is 4 times 10? 40 What is 2 times 3? 6**  **What is 7 times 4? 28 What is 4 times 10? ^C Final score: 5**  **>** |

The first time, you’ll answer a few questions and then hit Ctrl-C.

Ctrl-C sends the process an interrupt signal (SIGINT) that makes the program display the final score and then exit().

The user hit Ctrl-C here. The program displayed the final score before ending.

|  |
| --- |
| File Edit Window Help |
| **> ./math\_master**  **What is 5 times 9? 45 What is 2 times 8? 16 What is 9 times 1? 9 What is 9 times 3?**  **TIME'S UP!**  **Final score: 3**  **>** |

The second time, instead of hitting Ctrl-C, wait for at least five seconds on one of the answers and see what happens.

Test 2: wait five seconds

The alarm signal (SIGALRM) fires. The program was waiting for the user to enter an answer, but because he took so long, the timer signal was sent; the process immediately switches to the times\_up()

handler function. The handler displays the “TIME’S UP!” message and then escalates the signal to a SIGINT that causes the program to

display the final score.

Uh, oh…looks like someone

was a little slow.

Signals area little complex, but incredibly useful. They allow

your programs to end gracefully, and the interval timer can help you deal with tasks that are taking too long.

**464 *Chapter 10***

***interprocess communication***

~~b"iesxions~~

Q:**Are signals always received in the same order they are sent?**

Not if they are sent very close

A:

together. The operating system might choose to reorder the signals if it thinks one is more important than the others.

Q: **Is that always true?**

A:

It depends on the platform. On

most versions of Cygwin, for example, the signals will always be sent and received in the same order. But in general, you

shouldn’t rely on it.

**If I send the same signal twice,**

Q:

A:

**will it be received twice by the process?**

Again, it depends. On Linux and the Mac, if the same signal is repeated very

quickly, the kernel might choose to only

send the signal once to the process. On

Cygwin, it will always send both signals. But again, you should not assume that just because you sent the same signal twice, it will be received twice.

|  |  |
| --- | --- |
| ■ The operating system talks to processes using signals.  ■ Programs are normally stopped using signals.  ■ When a process receives a signal, it runs a handler.  ■ For most error signals, the default handler stops the program.  ■ Handlers can be replaced with the signal()function.  ■ You can send signals to yourself with raise(). | ■ The interval timersends SIGALRM signals.  ■ The alarm()function sets the interval timer.  ■ There is one timer per process.  ■ Don’t use sleep() and alarm() at the same time.  ■ kill sends signals to a process.  ■ kill -KILL will always kill a process. |

***you are here*** **465**

***c toolbox***

Your C Toolbox

**You’ve** **got** **Chapter** **10**

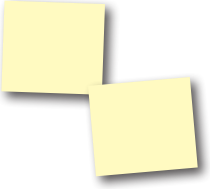
**under** **your** **belt,** **and** **now**

**you’ve** **added** **interprocess**

**communication** **to** **your** **toolbox.** **For**

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



fileno()

exit() stops

the program

immediately.

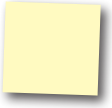
finds the

descriptor.

dup2()

duplicates a

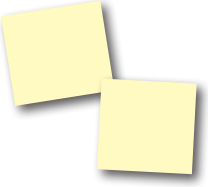
data stream.

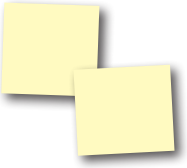
waitpid()

waits for a

process to

finish.



Signals are

pipe()

se

creates a

communication

O/S.

pipe.

sigaction()

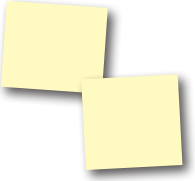
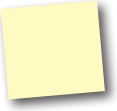
lets you

Processes can

communicate

handle signals.

using pipes.



alarm() sends

a SIGALRM

after a few

seconds.

The kill

command

sends a signal.

A program can

send signals

to itself with

raise().