**12** threads





**It’saparallel** **world**





Johnny told me he

got his heap variables locked in a mutex.



**Programs often need to do several things at the same time.**

POSIX threads can make your code more responsive by **spinning off several pieces of code to run in parallel**. But be careful! Threads are powerful tools, but you don’t want

them crashing into each other. In this chapter, you’ll learn how to put up **traffic signs** and **lane markers** that will ***prevent a code pileup***. By the end, you will knowhow to **create POSIX threads** and how to use **synchronization mechanisms** to ***protect the integrity of sensitive data***.

**this is a new chapter 501**

***working in parallel***

Tasks are sequential…or not…

Imagine you are writing something complex like a game in

C. The code will need to perform several different tasks:

**It will need to**

**calculate the latest locations of the**

**It will need to update**

**the graphics on the objects that are**



**screen.**

**moving in the game.**





**It might need**

**It will need to read control information from the games**

**to communicate with the disk**

**and the network.**

**controller or**

**keyboard.**

Not only will your code need to do all of these things, but it will need to do them ***all at the sametime***. That’s going to be true for many different programs. Chat programs

will need to read text from the network and send data to the network at the sametime. Media players will need to stream video to the display as well as watch for input from the user controls.

**How** **can** **your** **code** **perform** **several** **different** **tasks** **at** **once?**

***threads***

…and processes are not always the answer

You’ve already learned how to make the computer do

several things at once: with ***processes***. In the last chapter, you built a network server that could deal with several

different clients at once. Each time a new user connected, the server created a new process to handle the new session.

Does that mean that whenever you want to do several

things at once, you should just create a separate process? Well, not really, and here’swhy.

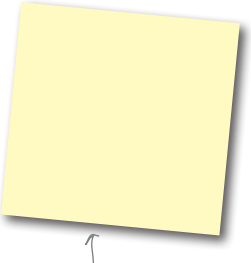
|  |
| --- |
| **Processes take time to create**  Some machines take a little while to create new processes. Not much time, but some. If the extra task you want to perform takes just a few hundredths of a second, creating a process each time won’t be very efficient.  **Processes can’t share data easily**  When you create a child process, it automatically has a complete copy of all the data from the parent process. But it’sa copy of the data. If the child needs to send data back to the parent, then you need something like a pipe to do that for you.  **Processes are just plain difficult**  You need to create a chunk of code to generate processes, and that can make your programs long and messy. |

You need something that starts a separate task quickly, can share all of your current data, and won’t need a huge

amount of code to build.

**You** **need** **threads.**

***you are here*** **503**

***single threads of execution***

Shop-n-Surf

Run the cash register.

Simple processes do one thing at a time

Say you have a task list with a set of things that you need to do:

Stock the shop.

Rewax the surfboards.

Answer the phones.

Fix the roof.



Keep the books.



***Shop-n-Surf***

just go surfing.

Alternatively,

You can’t do all of the tasks at the sametime, not by

yourself. If someone comes into the shop, you’ll need to

stop stocking the shelves. If it looks like rain, you might

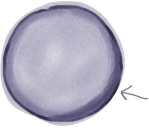
stop bookkeeping and get on the roof. If you work in a

shop alone, you’re like a simple process: you do one thing after another, but always one thing at a time. Sure, you can switch between tasks to keep everything going, but what

if there’sa **blocking operation**? What if you’reserving someone at the checkout and the phone rings?

Well, I can’t do

everything all at once. Who do you think I am?



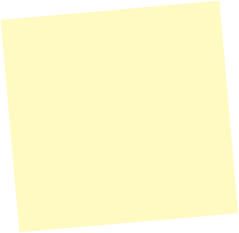
All of the programs you’ve written so far have had a **single thread of execution**. It’slike there’s only been one

Process.

person working inside the program’s process.

**504 *Chapter 12***

***threads***



Employ extra staff: use threads

Shop-n-Surf

Run the cash register.

A **multithreaded** program is like a shop with several

people working in it. If one person is running the checkout, another is filling the shelves, and someone else is waxing the surfboards, then everybody can work without interruptions. If one person answers the phone, it won’t stop the other

Stock the shop.

Re-wax the surfboards.

people in the shop.

Answer the phones.

Fix the roof.

If you employ more

Keep the books.

people, more than

one thing can be



done at once.

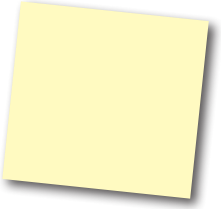
In the same way that several people can work in the same shop, you can have several threads living inside the same

process. All of the threads will have access to the same piece of heap memory. They will all be able to read and write to the same files and talk on the same network sockets. If one thread changes a global variable, all of the other threads will see the change immediately.

That means you can give each thread a separate task and they’ll all be performed at the sametime.

You can run each task

inside a separate thread.



If one thread has to

something, the other

wait for

threads

Read games controller

input.

can keep running.

Update screen.

Calculate physics of

rocket.

All of the threads can run inside a single process.

Send text message to

network.

***you are here*** **505**

***creating threads***

How do you create threads?

There area few thread libraries, and you’re going to use one of the most popular: the **POSIX thread library**, or **pthread**. You can use the pthread library on Cygwin, Linux, and the Mac.

Let’s say you want to run these two functions in separate threads:

Thread functions

need to have a 

void\* return type.

|  |
| --- |
| **void\*** does\_too(void \*a) {  int i = 0;  for (i = 0; i < 5; i++) { sleep(1);  puts("Does too!"); }  return NULL; } |

Nothing useful to return, so

just use NULL.

|  |
| --- |
| **void\*** does\_not(void \*a) {  int i = 0;  for (i = 0; i < 5; i++) { sleep(1);  puts("Does not!"); }  return NULL; } |

Did you notice that both functions return a*void pointer*?

Remember, avoid pointer can be used to point to any piece of data in memory, and you’ll need to make sure that your thread functions have a**void\*** return type.

You’re going to run each of these functions inside its own thread.



**Thread A**



**void\*** does\_not(void \*a) {

**Thread B**

**Main program**



**void\*** does\_too(void \*a) {

You’ll need to run both of these functions in parallel in separate threads. Let’s see how to do that.

**506 *Chapter 12***

***threads***

Create threads with pthread\_create

To run these functions, you’ll need a little setup code, like some headers and maybe an error() function that you can call if there’sa problem.

#include #include #include #include #include

**#include**

.. -These are the headers for the main part of the code.

This is the header for the pthread library.

**<pthread.h>**

void error(char \*msg) {

fprintf(stderr, "%s: %s\n", msg, strerror(errno)); exit(1);

}

But then you can start the code for your main function. You’re going

to create two threads, and each one needs to have its info stored in a

**pthread\_t** data structure. Then you can create and run a thread with

**pthread\_create()**. This records all the information about the thread.

This

creates the

thread.

pthread\_t t0; does\_not is the name of the function the thread will run.

pthread\_t t1; \-

 ifer(**r**(**c**'**e**a**(0**t**,**h **,**t**e**)**s**;**\_not, NULL)** == -1)  arck

if (**pthread\_create(&t1, NULL, does\_too, NULL)** == -1)

error("Can't create thread t1");

&t1 is the address of the

data structure that will store

That code will run your two functions in separate threads. But you’ve not quite finished yet. If your program just ran this and then finished, the

the thread info.

threads would be killed when the program ended. So you need to wait for

your threads to finish:

void\* result;

 The void pointer returned from each function will be stored here.

if (**pthread\_join(t0, &result)** == -1) error("Can't join thread t0");

if (**pthread\_join(t1, &result)** == -1) error("Can't join thread t1");

The pthread\_join() function

waits for a thread to finish.

The pthread\_join() also receives the return value of your thread function and stores it in avoid pointer variable. Once both threads have finished, your program can exit smoothly.

**Let’s** **see** **if** **it** **works.**

***you are here*** **507**

***test drive***

 ~~Test Drive~~

Because you’reusing the pthread library, you’ll need

to make sure you link it when you compile your program, like this:

This will link the

pthread library.

This is your program.

|  |
| --- |
| File Edit Window Help Don’tLoseTheThread |
| **> gcc argument.c -lpthread -o argument** |

When you run the code, you’ll see both functions running at the sametime:

When you run the code, the

messages might come out in a

different order than this.

|  |
| --- |
| File Edit Window Help Don’tLoseTheThread  **> ./argument**  **too! not! too! not! too! not! too! not! not! too!**  **Does Does Does Does Does Does Does Does Does Does >** |

~~btiexiom~~

**If both functions are running at the same time, why don’t the letters in the messages get mixed up? Each**

Q:

A:

**message is on its own line.**

That’s because of the way the Standard Output works. The text from puts()will all get output at once.

Q: **I removed the sleep()function, and the output showed all the output from one function and then all the output from the other function. Why is that?**

Most machines will run the code so quickly that without the sleep()call, the first function will finish before the second thread starts running.

A:

**508 *Chapter 12***

***threads***



Beer Magnets

It’s time for a really BIG party. This code runs 20 threads that count the number of beers down from 2,000,000. See if you can spot the missing code, and if you get the answer right, celebrate by cracking open a

couple of cold ones yourself.

int beers = 2000000;  Begin with 2 million beers.

void\* drink\_lots(void \*a)

{  Each thread will run this function. int i;

for (i = 0; i < 100000; i++) {

beers = beers - 1; The function will reduce the } beers variable by 100,000. return NULL;

}

int main() {

pthread\_t threads[20]; int t;

printf("%i bottles of beer on the wall\n%i bottles of beer\n", beers, beers);

for (t = 0; t < 20; t++) {  ’ eeninads si poarc,r, lniu do

( , NULL, · that! · , NULL);  }

void\* result;

for (t = 0; t < 20; t++) {

This code waits for all the

extra threads to finish.

(threads[t], &result);  }

printf("There are now %i bottles of beer on the wall\n", beers); return 0;

}



**pthread\_create**

**threads[t]**

**&threads[t]**

**threads**

|  |
| --- |
| **pthread\_join** |

|  |
| --- |
| **drink\_lots** |

***you are here*** **509**

***beer solved***



Beer Magnets Solution

It’s time for a really BIG party. This code runs 20 threads that count the number of beers down from 2,000,000. You were to spot the missing code.

int beers = 2000000; void\* drink\_lots(void {

\*a)

int

i;

for (i = 0; i < 100000; i++) { beers = beers - 1;

}

return NULL;

}

main()

int {

pthread\_t threads[20]; int t;

printf("%i bottles of beer on for (t = 0; t < 20; t++) {

the

wall\n%i

bottles of beer\n", beers, beers);

rs—cued’niddei!



**&threads[t]**

**pthread\_create**

(

|  |
| --- |
| **drink\_lots** |

NULL);

NULL,

,

,

}

void\* result;

for (t = 0; t < 20; t++) {

(threads[t], &result);

|  |
| --- |
| **pthread\_join** |

}

printf("There are now %i bottles of beer on the wall\n", beers); return 0;

}

**threads[t]**

|  |
| --- |
| **threads** |

**510 *Chapter 12***

***threads***

 ~~Test Drive~~

Let’stake a closer look at that last program. If you compile and run the code a few times, this happens:

The 20 threads have reduced the beers variable to 0.

|  |
| --- |
| File Edit Window Help Don’tLoseTheThread |
| **> ./beer**  **2000000 bottles of beer on the wall**  **2000000 bottles of beer**  **There are now 0 bottles of beer on the wall > ./beer**  **2000000 bottles of beer on the wall**  **2000000 bottles of beer**  **There are now 883988 bottles of beer on the wall > ./beer**  **2000000 bottles of beer on the wall**  **2000000 bottles of beer**  **There are now 945170 bottles of beer on the wall >** |

Hey, wait…

WTF?????

个

Where’s The Froth?

**The** **code** **usually** **doesn’t** **reduce** **the** **beers** **variable** **to** **zero.**

That’s really odd. The beers variable begins with a value of 2 million. Then 20 threads each try to reduce the value by 100,000. Shouldn’t that mean that the beers variable *always* goes to zero?

|  |
| --- |
| Look carefully at the code again, and try to imagine what will happen if several threads are running it at the same time. Why is the result unpredictable? Why doesn’t the beers variable get set to zero when all the threads have run? Write your answer below. |

***you are here*** **511**

***not thread-safe***

The code is not thread-safe

The great thing about threads is that lots of different tasks can run at the sametime and have access to the same data. The downside is that all those different threads have access to the same data …

Unlike the first program, the threads in the second program are all

reading and changing a shared piece of memory: the beers variable.

To understand what’s going on, let’s see what happens if two threads try to reduce the value of beers using this line of code:

**beers = beers - 1;** 

Imagine two threads are running this

line of code at the same time.



**1**

**First of all, both threads will need to read the current value of the beers variable.**

Thread 2

Thread 1



beers = 37



**2**



beers = 37

**Then, each thread will subtract 1 from**

**the number.**

Thread 2

 Both threads are getting

Thread 1

the same value. Can you see where this is going?

beers-1 = 36



**3**



beers-1 = 36

**Finally, each thread stores the value for beers–1 back into the beers variable.**



beers = 36



<- Thread 2

Thread 1 



beers = 36

Eventhough both of the threads were trying to reduce the value of beers by 1,they didn’t succeed. Instead of reducing the value by 2, they only decreased it by 1. That’s why the beers variable didn’t get reduced to zero—the threads kept getting in the way of each

|  |
| --- |
| **Be** **careful** **to** **look** **out** **for** **code** **that** **isn’t** **thread-safe.**  *How will you know?*  *Usually, if two threads read and write to the same variable, it’s not.* |

other.

And why was the result so unpredictable? Because the threads didn’t always run the line of code at exactly the sametime. Sometimes the threads didn’t crash into each other, and sometimes they did.

**512 *Chapter 12***

***threads***



You need to add traffic signals

Multithreaded programs can be powerful, but they can also behave in unpredictable ways, unless you put some controls in place.

Imagine two cars want to pass down the same narrow

stretch of road. To prevent an accident, you can add traffic signals. Those traffic signals prevent the cars from getting access to a shared resource (the road) at the sametime.

It’s the same thing when you want two or more threads

to access a shared data resource: you need to add traffic

signals so that no two threads can read the data and write it back at the sametime.



|  |
| --- |
| **Shared variable** |



**A**

The two cars represent

two threads. They both

want to access the same

shared variable.

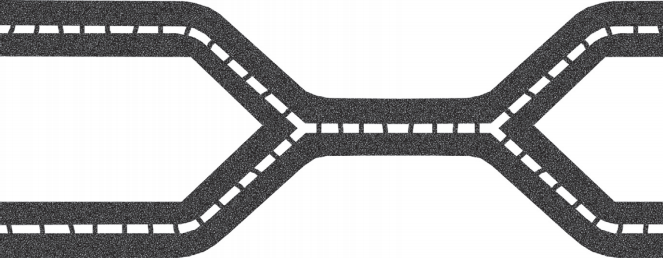


个

The traffic signals prevent the two

threads from accessing the same

shared variable at the same time.

**B**

The traffic signals that prevent threads from crashing into each other are called **mutexes**, and they are one of the

simplest ways of making your code thread-safe.

Mutexes are sometimes just called locks.

MUT-EX = MUTually EXclusive.

***you are here*** **513**

***mutex***

Use a mutexas a traffic signal

To protect a section of code, you will need to create amutex lock like this:

**pthread\_mutex\_t a\_lock = PTHREAD\_MUTEX\_INITIALIZER;**

The mutex needs to be visible to all of the threads that might crash into each other, so that means you’ll probably want to create it as a **global variable**.

PTHREAD\_MUTEX\_INITIALIZER is actually a macro.

When the compiler sees that, it will insert all of the code your program needs to create the mutex lock properly.



**1**

**Red means stop.**

At the beginning of your sensitive code section, you need to place your first traffic signal. The pthread\_mutex\_lock() will let only **one thread** get past. All the other threads will have to wait when they get to it.





**A**



**C**



**B**

**pthread\_mutex\_lock(&a\_lock);** Only one thread at a time will get past this. /\* Sensitive code starts here... \*/



**2**

**Green means go.**

When the thread gets to the end of the sensitive code, it makes a call to pthread\_mutex\_unlock(). That sets the traffic signal back to green, and another thread is allowed onto the sensitive code:

**A B**  **C**

/\* ...End of sensitive code \*/

**pthread\_mutex\_unlock(&a\_lock);**

Now that you knowhow to create locks in your code, you have a lot of control over exactly how your threads will work.

**514 *Chapter 12***

***threads***

Passing Long Values to Thread Functions Up Close

|  |  |  |
| --- | --- | --- |
| Thread functions can accept a single void pointer parameter and return a single void pointer value. Quite often, you will want to pass and  return integer values to a thread, and one trick is to use **long** values. longs can be stored in void pointers because they are the same size.   |  |  | | --- | --- | | void\* do\_stuff(void\* param) A thread function can accept a single  long before using it.  { void pointer parameter.  long thread\_no = (long)param; Convert it back to a long.  printf("Thread number %ld\n", thread\_no);  return (void\*)(thread\_no + 1); }  Cast it back to a void pointer when it’s returned.  int main() {  pthread\_t threads[20]; long t;  Convert the long t value to a void pointer.  for (t = 0; t < 3; t++) {  pthread\_create(&threads[t], NULL, do\_stuff, (void\*)t);  }  void\* result;  for (t = 0; t < 3; t++) {  Convert the return value to a  pthread\_join(threads[t], &result);  printf("Thread %ld returned %ld\n", t, (long)result); }  return 0; }  File Edit Window Help Don’tLoseTheThread   |  | | --- | | **> ./param\_test**  **Thread number 0**  **Thread 0 returned 1 Thread number 1**  **Thread number 2**  **Thread 1 returned 2 Thread 2 returned 3 >** | |   Each thread receives its thread number.  Each thread returns its thread number + 1. |

***you are here*** **515**

***exercise***

pthread\_mutex\_t beers\_lock = PTHREAD\_MUTEX\_INITIALIZER; void\* drink\_lots(void \*a)

{

int i;

pthread\_mutex\_lock(&beers\_lock); for (i = 0; i < 100000; i++) {

beers = beers - 1; }

pthread\_mutex\_unlock(&beers\_lock); printf("beers = %i\n", beers);

return NULL;

}

pthread\_mutex\_t beers\_lock = PTHREAD\_MUTEX\_INITIALIZER; void\* drink\_lots(void \*a)

{

int i;

for (i = 0; i < 100000; i++) {

pthread\_mutex\_lock(&beers\_lock);

beers = beers - 1;

pthread\_mutex\_unlock(&beers\_lock);

}

printf("beers = %i\n", beers); return NULL;

}

|  |
| --- |
| There’s no simple way to decide where to put the locks in your code. Where you put them will change the way the code performs. Here are two versions of the drink\_lots() function  that lock the code in different ways.  Version A  Version B |

**516 *Chapter 12***

***threads***

Match the code to

the output.

**> ./beer\_fixed\_strategy\_2**

**2000000 bottles of beer on the wall**

**2000000 bottles of beer**

**beers = 63082 beers = 123**

**beers = 104 beers = 102 beers = 96 beers = 75 beers = 67 beers = 66 beers = 65 beers = 62 beers = 58 beers = 56 beers = 51 beers = 41 beers = 36 beers = 30 beers = 28 beers = 15 beers = 14 beers = 0**

**There are now 0 bottles of beer on the wall >**

File Edit Window Help Don’tLoseTheThread

|  |
| --- |
| Both pieces of code use a mutex to protect the beers variable, and each now displays the value of beers before they exit, but because they are locking the code in different places, they generate different output on the screen.  Can you figure out which version produced each of these two runs? |
| File Edit Window Help Don’tLoseTheThread  **> ./beer**  **2000000 bottles of beer 2000000 bottles of beer beers = 1900000**  **on the wall**  **beers = 1800000 beers = 1700000 beers = 1600000 beers = 1500000 beers = 1400000 beers = 1300000 beers = 1200000 beers = 1100000 beers = 1000000 beers = 900000 beers = 800000 beers = 700000 beers = 600000 beers = 500000 beers = 400000 beers = 300000 beers = 200000 beers = 100000 beers = 0**  **There are now 0 bottles >**  **of beer on the wall** |

**517**

***exercise solved***

pthread\_mutex\_t beers\_lock = PTHREAD\_MUTEX\_INITIALIZER; void\* drink\_lots(void \*a)

{

int i;

pthread\_mutex\_lock(&beers\_lock); for (i = 0; i < 100000; i++) {

beers = beers - 1; }

pthread\_mutex\_unlock(&beers\_lock); printf("beers = %i\n", beers);

return NULL;

}



pthread\_mutex\_t beers\_lock = PTHREAD\_MUTEX\_INITIALIZER; void\* drink\_lots(void \*a)

{

int i;

for (i = 0; i < 100000; i++) {

pthread\_mutex\_lock(&beers\_lock);

beers = beers - 1;

pthread\_mutex\_unlock(&beers\_lock);

}

printf("beers = %i\n", beers); return NULL;

}

|  |
| --- |
| There’s no simple way to decide where to put the locks in your code. Where you put them will change the way the code performs. Here are two versions of the drink\_lots() function that lock the code in different ways.  Version A    Version B |

**518 *Chapter 12***

***threads***

Match the code to

the output.

**> ./beer\_fixed\_strategy\_2**

**2000000 bottles of beer on the wall**

**2000000 bottles of beer**

**beers = 63082 beers = 123**

**beers = 104 beers = 102 beers = 96 beers = 75 beers = 67 beers = 66 beers = 65 beers = 62 beers = 58 beers = 56 beers = 51 beers = 41 beers = 36 beers = 30 beers = 28 beers = 15 beers = 14 beers = 0**

**There are now 0 bottles of beer on the wall >**

File Edit Window Help Don’tLoseTheThread

|  |
| --- |
| Both pieces of code use a mutex to protect the beers variable, and each now displays the value of beers before they exit, but because they are locking the code in different places, they generate different output on the screen.  You were to figure out which version produced each of these two runs. |
| File Edit Window Help Don’tLoseTheThread  **> ./beer**  **2000000 bottles of beer 2000000 bottles of beer beers = 1900000**  **on the wall**  **beers = 1800000 beers = 1700000 beers = 1600000 beers = 1500000 beers = 1400000 beers = 1300000 beers = 1200000 beers = 1100000 beers = 1000000 beers = 900000 beers = 800000 beers = 700000 beers = 600000 beers = 500000 beers = 400000 beers = 300000 beers = 200000 beers = 100000 beers = 0**  **There are now 0 bottles >**  **of beer on the wall** |

**519**

***congratulations!***

**Congratulations! You’ve (almost)**

**reached the end of the book. Now it’s time to crack open one of those**

**2,000,000 bottles of beer and celebrate!**

You’re now in a great position to decide what *kind* of C coder you want to be. Do you want to be a **Linux hacker** using pure C? Or a **maker** writing embedded C in small devices like the Arduino?

Maybe you want to go on to be a **games developer** in C++? Or a **Mac and iOS programmer** in Objective-C?

Whatever you choose to do, you’re now part of the community that uses and loves

the language that has created more software than any other. The language behind the Internet and almost every operating system. The language that’s used to *write almost all the otherlanguages*. And the language that can write for almost every processor in

existence, from watches and phones to planes and satellites.

**New** **C** **Hacker,** **we** **salute** **you!**

~~btiexioms~~

Q: **Does my machine have to**

**have multiple processors to support threads?**

No. Most machines have processors with multiple **cores**, which means that

A:

their CPUs contain miniprocessors that can do several things at once. But even if your code is running on a single core/ single processor, you will still be able to run threads.

Q: **How?**

The operating system will switch rapidly between the threads and make it appear that it is running several things at once.

A:

Q:**Will threads make my programs faster?**

Not necessarily. While threads can help you use more of the processors and cores on your machine, you need to be careful about the amount of locking your code needs to do. If your threads are

A:

locked too often, your code may run as slowly as single-threaded code.

**How can I design my thread code to be fast?**

Q:

A:

Try to reduce the amount of data

that threads need to access. If threads

don’t access a lot of shared data, they

won’t need to lock each other out so often and will be much more efficient.

Q:**Are threads faster than separate processes?**

They usually are, simply because it takes a little more time to create

A:

Q:

A:

processes than it does to create extra threads.

**I’ve heard that mutexes can lead to “deadlocks.” What are they?**

Say you have two threads, and they both want to get mutexes A and B. If the first thread already has A, and the second thread already has B, then the threads will be deadlocked. This is because the first thread can’t get mutex B and the second thread can’t get mutex A. They both come to a standstill.

**520 *Chapter 12***

***threads***

**CHAPTER 12**

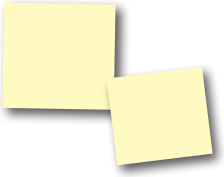
Your C Toolbox

**You’ve** **got** **Chapter** **12** **under** **your** **belt,** **and** **now** **you’ve**

**added** **threads** **to** **your** **toolbox.**

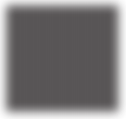
**For** **a** **complete** **list** **of** **tooltips** **in** **the**

**book,** **see** **Appendix** **ii.**

Threads allow a process to do more than one thing at the

Simple

Threads are

same time.

processes do

one thing at a

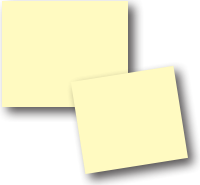
“lightweight

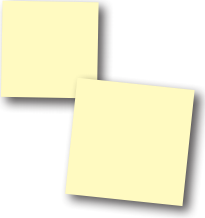
processes.”

time.

POSIX threads (pthread) is

a threading library.



Threads

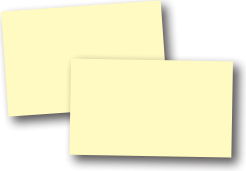
pthread\_create() creates a thread to run a function.

share the same global variables.

r()

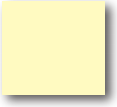
a thread to finish.

If two threads read and update the same variable, your code will be unpredictable.



pthread\_mutex\_lock()

creates a mutex on code.

Mutexes are

locks that

pthread\_mutex\_unlock() releases the mutex.

protect shared data.

***you are here*** **521**