Concurrent Programming with Threads*

Randal E. Bryant
David R. O'Hallaron

February 5, 2001

1 Introduction

A *thread* is a unit of execution, associated with a process, with its own thread ID, stack, stack pointer, program counter, condition codes, and general-purpose registers. Multiple threads associated with a process run concurrently in the context of that process, sharing its code, data, heap, shared libraries, signal handlers, and open files.

Programming with threads instead of conventional processes is increasingly popular because threads are less expensive than processes and because they provide a trivial mechanism for sharing data. For example, a high-performance Web server might assign a separate thread for each open connection to a Web browser, with each thread sharing a single in-memory cache of frequently requested Web pages.

Another important factor in the popularity of threads is the adoption of the standard *Pthreads* (Posix threads) interface for manipulating threads from C programs. The benefit of threads has been known for some time, but their use was hindered because each computer vendor developed its own incompatible threads package. As a result, threaded programs written for one platform would not run on other platforms. The adoption of Pthreads in 1995 has improved this situation immensely. Posix threads are now available on most systems, including Linux.

This handout is an introduction to Pthreads programming. We describe the most commonly used Pthreads functions, introduce the general concept of synchronization, and discuss some common techniques for synchronizing threads.

2 Basic thread concepts

To this point, we have worked with the traditional view of a process shown in Figure 1. In this view, a process consists of the code and data in the user's virtual memory, along with some state maintained by the kernel known as the *process context*. The code and data includes the program's text, data, runtime

heap, shared libraries, and the stack. The process context can be partitioned into of two different kinds of state: *program context* and *kernel context*. The program context resides in the processor, and includes the contents of the general-purpose registers, various condition codes registers, the stack pointer, and the program counter. The kernel context resides in the kernel, and consists of items such as the process ID, the data structures that characterize the organization of the virtual memory, and information about open files, installed signal handlers, and the extent of the heap.

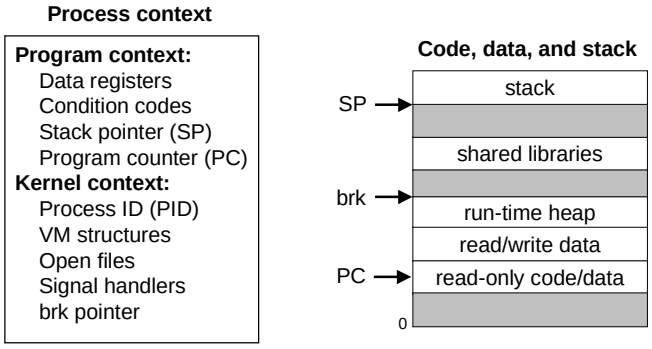

Figure 1: **Traditional view of a process.**

If we rearrange the items in Figure 1, then we arrive at the more modern view of a process shown in Figure 2. Here, a process consists of a thread, which consists of a stack and the program context (which we will call the *thread context*), plus the kernel context and the program code and data (minus the stack, of course).

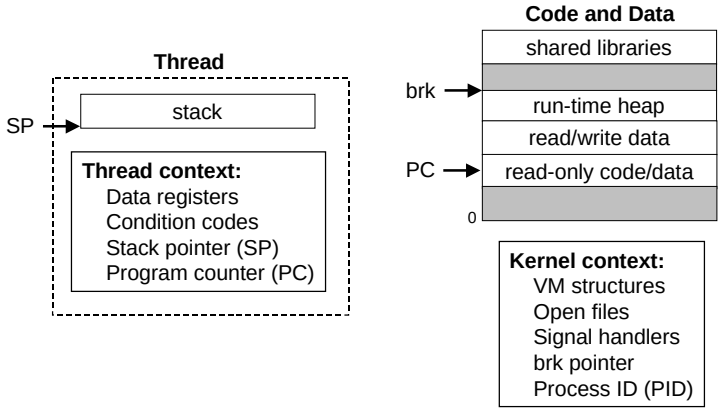

Figure 2: **Modern view of a process.**

The interesting point about Figure 2 is that it treats the process as an execution unit with a very small amount of state that runs in the context of a much larger amount of state. Given this view, we can now extend our notion of process to include multiple threads that share the same code, data, and kernel context, as shown in Figure 3. Each thread associated with a process has its own stack, registers, condition codes, stack pointer, and program counter. Since there are now multiple threads, we will also add an integer *thread ID* (TID) to each thread context.

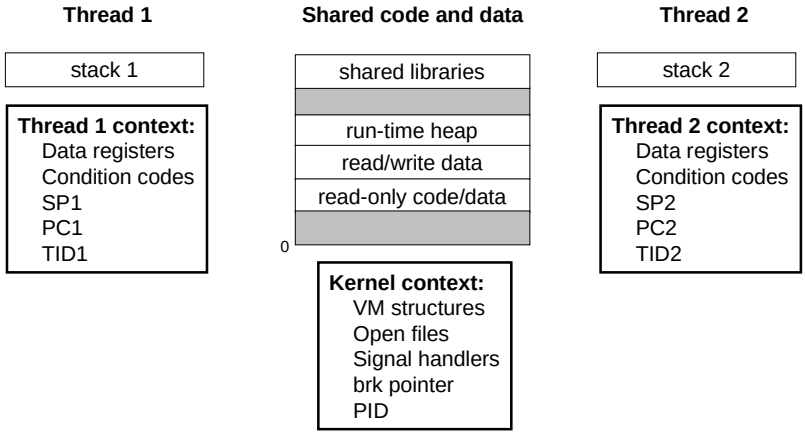

Figure 3: **Associating multiple threads with a process.**

The execution model for multiple threads is similar in some ways to the execution model for multiple processes. Consider the example in Figure 4. Each process begins life as a single thread called the *main thread*. At some point, the main thread creates a *peer thread* and from this point in time the two threads run concurrently (i.e., their logical flows overlap in time). Eventually, control passes to the peer thread via a context switch, because the main thread executes a slow system call such as `read` or `sleep`, or because it is interrupted by the system's interval timer. The peer thread executes for awhile before control passes back to the main thread, and so on.

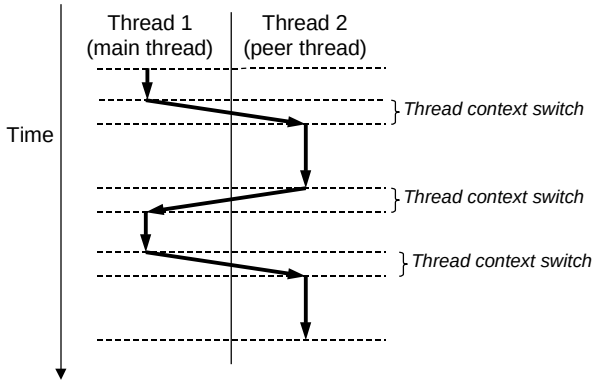

Figure 4: **Concurrent thread execution.**

Thread execution differs from processes in some important ways. Because a thread context is much smaller than a process context, a thread context switch is faster than a process context switch. Another difference is that threads, unlike processes, are not organized in a rigid parent-child hierarchy. The threads associated with a process form a pool of peers, independent of which threads were created by which other threads. The main thread is distinguished from other threads only in the sense that it is always the first thread to run in the process. The main impact of this notion of a pool of peers is that a thread can kill any of its peers, or wait for any of its peers to terminate. Further, each peer can read and write the same shared data.

3 Thread control

Pthreads defines about 60 functions that allow C programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state. However, most threaded programs use only a small subset of the functions defined in the interface.

Figure 5 shows a simple Pthreads program called `hello.c`. In this program, the main thread creates a peer thread and then waits for it to terminate. The peer thread prints `hello, world` and terminates. When the main thread detects that the peer thread has terminated, it terminates itself (and the entire process) by calling `exit`.

../code/threads/hello.c

```
1 #include "ics.h"
2
3 void *thread(void *vargp);
4
5 int main()
6 {
7     pthread_t tid;
8
9     Pthread_create(&tid, NULL, thread, NULL);
10    Pthread_join(tid, NULL);
11    exit(0);
12 }
13
14 /* thread routine */
15 void *thread(void *vargp)
16 {
17    printf("Hello, world!\n");
18    return NULL;
19 }
```

../code/threads/hello.c

Figure 5: `hello.c`**: The Pthreads "hello, world" program.**

This is the first threaded program we have seen, so let's dissect it carefully. Line 3 is the prototype for the thread routine `thread`. The Pthreads interface mandates that each thread routine has a single `(void *)` input argument and returns a single `(void *)` output value. If you want to pass multiple arguments to a thread routine, then you can put the arguments into a structure and pass a pointer to the structure. Similarly, if you want the thread routine to return multiple arguments, you can return a pointer to a structure.

Line 5 marks the beginning of the `main` routine, which runs in the context of the main thread. In line 7, the main routine declares a single local variable `tid`, which will be used to store the thread ID of the peer thread. In line 9, the main thread creates a new peer thread by calling the `pthread_create` function.[1]

[1]We are actually calling one of our error-handling wrapper functions, which were introduced in Section **??** and described in detail in Appendix **??**.

When the call to `pthread_create` returns, the main thread and the newly created thread are running concurrently, and `tid` contains the ID of the new thread. In line 10, the main thread waits for the newly created thread to terminate. Finally, in line 11, the main thread terminates itself and the entire process by calling `exit`.

Lines 15–19 define the thread routine, which in this case simply prints a string then terminates by executing the `return` statement in line 18.

3.1 Creating threads

Threads create other threads by calling the `pthread_create` function.

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
```
returns: 0 if OK, non-zero on error

The `pthread_create` function creates a new thread and runs the *thread routine* `f` in the context of the new thread and with an input argument of `arg`. The `attr` argument can be used to change the default attributes of the newly created thread. However, changing these attributes is beyond our scope, and in our examples, we will always call `pthread_create` with a NULL `attr` argument.

When `pthread_create` returns, argument `tid` contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>

pthread_t pthread_self(void);
```
returns: thread ID of caller

3.2 Terminating threads

A thread terminates in one of the following ways:

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the `pthread_exit` function, which returns a pointer to the return value `thread_return`. If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of `thread_return`.

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```
returns: 0 if OK, non-zero on error

- Some peer thread calls the Linux `exit` function, which terminates the process and all threads associated with the process.
- Another peer thread terminates the current thread by calling the `pthread_cancel` function with the ID of the current thread.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```
returns: 0 if OK, non-zero on error

3.3 Reaping terminated threads

Threads wait for other threads to terminate by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```
returns: 0 if OK, non-zero on error

The `pthread_join` function blocks until thread `tid` terminates, assigns the `(void *)` pointer returned by the thread routine to the location pointed to by `thread_return`, and then *reaps* any memory resources held by the terminated thread.

Notice that, unlike the Linux `wait` function, the `pthread_join` function can only wait for a specific thread to terminate. There is no way to instruct `pthread_wait` to wait for an arbitrary thread to terminate. This can complicate our code by forcing us to use other less intuitive mechanisms to detect process termination. Indeed some have argued convincingly that this represents a bug in the specification [6].

3.4 Detaching threads

At any point in time, a thread is *joinable* or *detached*. A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread. In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

By default, threads are created joinable. In order to avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to the `pthread_detach` function.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```
returns: 0 if OK, non-zero on error

The `pthread_detach` function detaches the joinable thread `tid`. Threads can detach themselves by calling `pthread_detach` with an argument of `pthread_self()`.

Even though of our examples will use joinable threads, there are good reasons to use detached threads in real programs. For example, a high-performance Web server might create a new peer thread each time it receives a connection request from a Web browser. Since each connection is handled independently by a separate thread, it is unnecessary and indeed undesirable for the server to explicitly wait for each peer thread to terminate. In this case, each peer thread should detach itself before it begins processing the request so that its memory resources can be reclaimed after it terminates.

Problem 1 [Category 1]:

A. The program in Figure 6 has a bug. The thread is supposed to sleep for one second and then print a string. However, when we run it, nothing prints. Why?

B. We can fix this bug by replacing the `exit` function in line 9 with one of two different Pthreads function calls. Which ones?

Problem 2 [Category 2]:
Write a version of `hello.c` (Figure 5) that inputs a command line argument n, and then creates and reaps n joinable peer threads.

4 Shared variables in threaded programs

From a programmer's perspective, one of the attractive aspects of threads is the ease with which multiple threads can share the same program variables. However, in order to use threads correctly, we must have a clear understanding of what we mean by sharing and how it works.

There are some basic questions to work through in order to understand whether a variable in a C program is shared or not: (1) What is the underlying memory model for threads? (2) Given this model, how are instances of the variable mapped to memory? (3) And finally, how many threads reference each of these instances? The variable is *shared* if and only if multiple threads reference some instance of the variable.

To keep our discussion of sharing concrete, we will use the program in Figure 7 as a running example. Although somewhat contrived, it is nonetheless useful to study because it illustrates a number of subtle points about sharing. The example program consists of a main thread that creates two peer threads. The main thread passes a unique id to each peer thread, which uses the id to print a personalized message, along with a count of the total number of times that the thread routine has been invoked. Here is the output when we run it on our system:

../code/threads/hellobug.c

```
1 #include "ics.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7
8     Pthread_create(&tid, NULL, thread, NULL);
9     exit(0);
10 }
11
12 /* thread routine */
13 void *thread(void *vargp)
14 {
15     Sleep(1);
16     printf("Hello, world!\n");
17     return NULL;
18 }
```

../code/threads/hellobug.c

Figure 6: **Buggy program for Problem 1.**

```
linux> sharing
[0]: Hello from foo (cnt=1)
[1]: Hello from bar (cnt=2)
```

4.1 Threads memory model

As we mentioned in Section 1, a pool of concurrent threads runs in the context of a process. Each thread has its own separate thread context, which includes a thread ID, stack, stack pointer, program counter, condition codes, and general-purpose register values. Each thread shares the rest of the process context with the other threads. This includes the entire user virtual address space, which consists of read-only text (code), read/write data, the heap, and any shared library code and data areas. The threads also share the same set of open files and the same set of installed signal handlers.

In an operational sense, it is impossible for one thread to read or write the register values of an another thread. On the other hand, any thread can access any location in the shared virtual memory. If some thread modifies a memory location, then every other thread will eventually see the change if it reads that location. Thus, registers are never shared, while virtual memory is always shared.

The memory model for the separate thread stacks is not as clean. These stacks are contained in the stack area of the virtual address space, and are *usually* accessed independently by their respective threads. We say *usually* rather than *always*, because different thread stacks are not protected from other threads. So if a thread somehow manages to acquire a pointer to another thread's stack, then it can read and write any part of that stack. Our example program shows an example of this in line 29, where the peer threads reference

../code/threads/sharing.c

```
 1 #include "ics.h"
 2 #define N 2
 3
 4 char **ptr;  /* global variable */
 5
 6 void *thread(void *vargp);
 7
 8 int main()
 9 {
10     int i;
11     pthread_t tid;
12     char *msgs[N] = {
13         "Hello from foo",
14         "Hello from bar"
15     };
16
17     ptr = msgs;
18
19     for (i = 0; i < N; i++)
20         Pthread_create(&tid, NULL, thread, (void *)i);
21     Pthread_exit(NULL);
22 }
23
24 void *thread(void *vargp)
25 {
26     int myid = (int)vargp;
27     static int cnt = 0;
28
29     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
30 }
```

../code/threads/sharing.c

Figure 7: **Example program that illustrates different aspects of sharing.**

the contents of the main thread's stack indirectly through the global `ptr` variable.

4.2 Mapping variables to memory

C variables in threaded programs are mapped to virtual memory according to their storage classes.

- *Global variables.* A *global variable* is any variable declared outside of a function At run-time, the read/write area of virtual memory contains exactly one instance of each global variable that can be referenced by any thread.

  For example, the global `ptr` variable in line 4 has one run-time instance in the read/write area of virtual memory. When there is only one instance of a variable, we will denote the instance by simply using the variable name, in this case `ptr`.

- *Local automatic variables.* A *local automatic variable* is one that is declared inside a function without the `static` attribute. At run-time, each thread's stack contains its own instances of any local automatic variables. This is true even if multiple threads execute the same thread routine.

  For example, there is one instance of the local variable `tid`, and it resides on the stack of the main thread. We will denote this instance as `tid.m`. As another example, there are two instances of the local variable `myid`, one instance on the stack of peer thread 0, and the other on the stack of peer thread 1. We will denote these instances as `myid.p0` and `myid.p1` respectively.

- *Local static variables.* A *local static variable* is one that is declared inside a function with the `static` attribute. As with global variables, the read/write area of virtual memory contains exactly one instance of each local static variable declared in a program.

  For example, even though each peer thread in our example program declares `cnt` in line 27, at run-time there is only one instance of `cnt` residing in the read/write area of virtual memory. Each peer thread reads and writes this instance.

4.3 Shared variables

A variable v is shared if and only if one of its instances is referenced by more than one threads. For example, variable `cnt` in our example program is shared because it has only one run-time instance, and this instance is referenced by both peer threads. On the other hand, `myid` is not shared because each of its two instances is referenced by exactly one thread. However, it is important to realize that local automatic variables such as `myid` can also be shared (See Problem 3).

Problem 3 [Category 1]:

A. Using the analysis from Section 4, complete the following table for the example program in Figure 7. In the first column, the notation $v.t$ denotes an instance of variable v residing on the local stack for thread t, where t is either `m` (main thread), `p0` (peer thread 0), or `p1` (peer thread 1).

Variable instance	Referenced by main thread?	Referenced by peer thread 0 ?	Referenced by peer thread 1?
`ptr`			
`cnt`			
`i.m`			
`msgs.m`			
`myid.p0`			
`myid.p1`			

B. Given the analysis in Part A, which of the variables `ptr`, `cnt`, `i`, `msgs`, and `myid` are shared?

5 Synchronizing threads with semaphores

Shared variables can be convenient, but they introduce the possibility of a new class of *synchronization errors* that we have not encountered to this point in our study of systems. Consider the `badcnt.c` program in Figure 8 that creates two threads, each of which increments a shared counter variable called `cnt`.

Since each thread increments the counter NITERS times, we might expect its final value to be $2 \times$ NITERS. However, when we run `badcnt.c` on our Linux system, we not only get wrong answers, we get different answers each time!

```
linux>  badcnt
BOOM! ctr=198841183

linux>  badcnt
BOOM! ctr=198261801

linux>  badcnt
BOOM! ctr=198269672
```

So what went wrong? To understand the problem clearly, we need to study the assembly code for the counter loop, which shown in Figure 9. We will find it helpful to partition the loop code for thread i into five parts:

- H_i: The block of instructions at the head of the loop.

../code/threads/badcnt.c

```
1  #include "ics.h"
2
3  #define NITERS 100000000
4
5  void *count(void *arg);
6
7  /* shared variable */
8  unsigned int cnt = 0;
9
10 int main()
11 {
12     pthread_t tid1, tid2;
13
14     Pthread_create(&tid1, NULL, count, NULL);
15     Pthread_create(&tid2, NULL, count, NULL);
16
17     Pthread_join(tid1, NULL);
18     Pthread_join(tid2, NULL);
19
20     if (cnt != (unsigned)NITERS*2)
21         printf("BOOM! cnt=%d\n", cnt);
22     else
23         printf("OK cnt=%d\n", cnt);
24     exit(0);
25 }
26
27 /* thread routine */
28 void *count(void *arg)
29 {
30     int i;
31
32     for (i=0; i<NITERS; i++)
33         cnt++;
34     return NULL;
35 }
```

../code/threads/badcnt.c

Figure 8: `badcnt.c`: **An improperly synchronized counter program.**

- L_i: The instruction that loads the shared variable `cnt` into register $\%eax_i$, where $\%eax_i$ denotes the value of register $\%eax$ in thread i.
- U_i: The instruction that updates (increments) $\%eax_i$.
- S_i: The instruction that stores the updated value of $\%eax_i$ back to the shared variable `cnt`.
- T_i: The block of instructions at the tail of the loop.

Notice that the head and tail manipulate only local stack variables, while L_i, U_i, and S_i manipulate the contents of the shared counter variable.

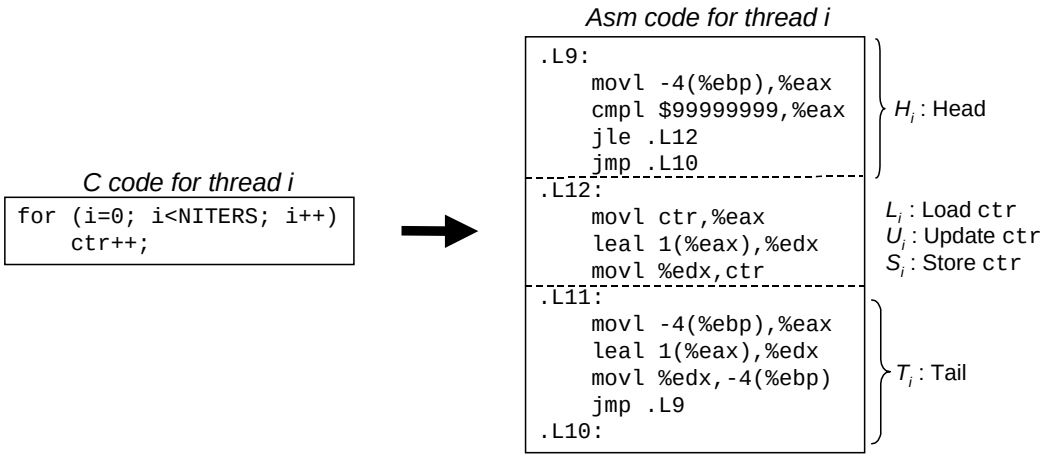

Figure 9: **IA32 assembly code for the counter loop in `badcnt.c`.**

5.1 Sequential consistency

When the two peer threads in `badcnt.c` run concurrently on a single CPU, the instructions are completed one after the other in some order. Thus, each concurrent execution defines some total ordering (or interleaving) of the instructions in the two threads. When we reason about concurrent execution, the *only* assumption we can make about the total ordering of instructions is that it is *sequentially consistent.* That is, instructions can be interleaved in any order, so long as the instructions for each thread execute in program order. For example, the ordering

$$H_1, H_2, L_1, L_2, U_1, U_2, S_1, S_2, T_1, T_2$$

is sequentially consistent, while the ordering

$$H_1, H_2, U_1, L_2, L_1, U_2, S_1, S_2, T_1, T_2$$

is not sequentially consistent because U_1 executes L_1. Unfortunately not all sequentially consistent orderings are created equal. Some will produce correct results, but others will not, and there is no way for us to predict whether the operating system will choose a correct ordering for our threads. For example, Figure 10(a) shows the step-by-step operation of a correct instruction ordering. After each thread has updated the shared variable `cnt`, its value in memory is 2, which is the expected result.

Step	Thread	Instr	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	1	S_1	1	–	1
5	2	H_2	–	–	1
6	2	L_2	–	1	1
7	2	U_2	–	2	1
8	2	S_2	–	2	2
9	2	T_2	–	2	2
10	1	T_1	1	–	2

(a) Correct ordering

Step	Thread	Instr	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	–	–	0
2	1	L_1	0	–	0
3	1	U_1	1	–	0
4	2	H_2	–	–	0
5	2	L_2	–	0	0
6	1	S_1	1	–	1
7	1	T_1	1	–	1
8	2	U_2	–	1	1
9	2	S_2	–	1	1
10	2	T_2	–	1	1

(b) Incorrect ordering

Figure 10: **Sequentially-consistent orderings for the first loop iteration in `badcnt.c`.**

On the other hand, the ordering in Figure 10(b) produces an incorrect value for `cnt`. The problem occurs because thread 2 loads `cnt` in step 5, after thread 1 loads `cnt` in step 2, but before thread 1 stores its updated value in step 6. Thus each thread ends up storing an updated counter value of 1.

We can clarify this idea of correct and incorrect instruction orderings with the help of a formalism known as a *progress graph*, which we introduce in the next section.

Problem 4 [Category 1]:
Which of the following instruction orderings for `badcnt.c` are sequentially consistent?

A. $H_1, H_2, L_1, L_2, U_1, U_2, S_2, S_1, T_2, T_1$.

B. $H_1, H_2, L_2, U_2, S_2, U_1, T_2, L_1, S_1, T_1$.

C. $H_2, L_2, H_1, L_1, U_1, S_1, U_2, S_2, T_2, T_1$.

D. $H_2, H_1, L_2, L_1, S_2, U_1, U_2, S_1, T_2, T_1$.

Problem 5 [Category 1]:
Complete the table for the following sequentially consistent ordering of `badcnt.c`.

Step	Thread	Instr	$\%eax_1$	$\%eax_2$	`cnt`
1	1	H_1	–	–	0
2	1	L_1			
3	2	H_2			
4	2	L_2			
5	2	U_2			
6	2	S_2			
7	1	U_1			
8	1	S_1			
9	1	T_1			
10	2	T_2			

Does this ordering result in a correct value for `cnt`?

5.2 Progress graphs

A *progress graph* models the execution of n concurrent threads as a trajectory through an n-dimensional Cartesian space. Each axis k corresponds to the progress of thread k. Each point $(I_1, I_2, \ldots, I_n)$ represents the state where thread k, $(k = 1, \ldots, n)$ has completed instruction I_k. The origin of the graph corresponds to the *initial state* where none of the threads has yet completed an instruction.

Figure 11 shows the 2-dimensional progress graph for the first loop iteration of the `badcnt.c` program. The horizontal axis corresponds to thread 1, the vertical axis to thread 2. Point (L_1, S_2) corresponds to the state where thread 1 has completed L_1 and thread 2 has completed S_2.

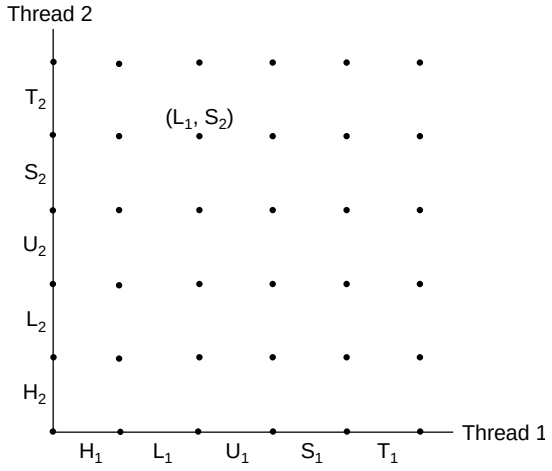

Figure 11: **Progress graph for the first loop iteration of** `badcnt.c`.

A progress graph models instruction execution as a *transition* from one state to another. A transition is represented as a directed edge from one point to an adjacent point. Figure 12 shows the legal transitions

in a 2-dimensional progress graph. For threads running on a single processor, where instructions complete one at a time in sequentially-consistent order, legal transitions move to the right (an instruction in thread 1 completes) or up (an instruction in thread 2 completes).

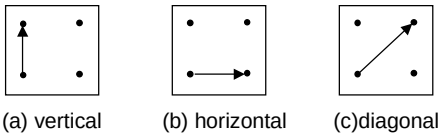

Figure 12: **Legal transitions in a progress graph.**

Programs never run backwards, so transitions that move down or to the left are not legal. However, if the underlying system is a multiprocessor and each thread runs on its own processor, then transitions that move diagonally up and to the right are also legal. A concurrent program should run correctly regardless of the number of processors, so in we general we will allow for diagonal transitions.

The execution history of a program is modeled as a *trajectory*, or sequence of transitions, through the state space. Figure 13 shows the trajectory that corresponds to the instruction ordering

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2.$$

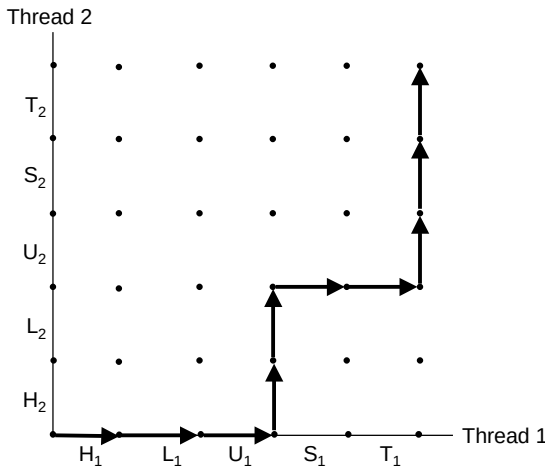

Figure 13: **An example trajectory.**

For thread i, the instructions (L_i, U_i, S_i) that manipulate the contents of the shared variable `cnt` constitute a *critical section* (with respect to shared variable `cnt`) that should not be interleaved with the critical section of the other thread. The intersection of the two critical sections defines a region of the state space known as an *unsafe region*. Figure 14 shows the unsafe region for the variable `cnt`. Notice that the unsafe region abuts, but does not include, the states along its perimeter. For example, states (H_1, H_2) and (S_1, U_2) abut the unsafe region, but are not a part of it.

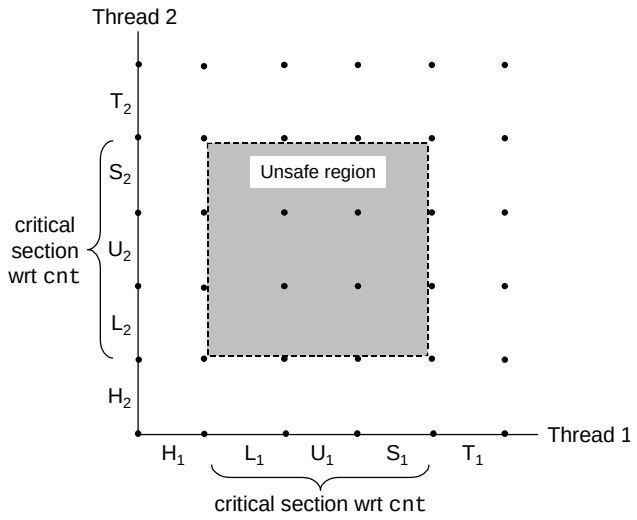

Figure 14: **Critical sections and unsafe regions.**

A trajectory that skirts the unsafe region is known as a *safe trajectory*. Conversely, a trajectory that touches any part of the unsafe region is an *unsafe trajectory*. Figure 15 shows examples of safe and unsafe trajectories through the state space of our example `badcnt.c` program. The upper trajectory skirts the unsafe region along its left and top sides, and thus is safe. The lower trajectory crosses the unsafe region with one of its diagonal transitions, and thus is unsafe.

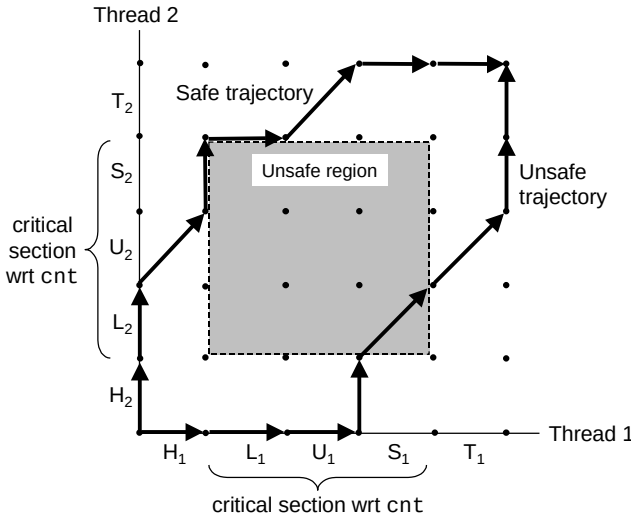

Figure 15: **Safe and unsafe trajectories.**

Any safe trajectory will correctly update the shared counter. Conversely, any unsafe trajectory will produce either a predictably wrong result or a result that cannot be predicted. The latter case arises when the trajectory crosses the lower right-hand corner of the region with a diagonal transition from state (U_1, H_2) to (S_1, L_2). If thread 1 stores its updated value of the counter variable before thread 2 loads it, then the result

will be correct, otherwise it will be incorrect. Since we cannot predict the ordering of load and store operations, we consider this case, as well as the symmetric case where the trajectory crosses the upper left-hand corner of the unsafe region, to be incorrect.

The bottom line is that in order to guarantee correct execution of our example threaded program — and indeed any concurrent program that shares global data structures – we must somehow *synchronize* the threads so that they always have a safe trajectory. Dijkstra proposed the first solution to this problem in a classic paper that introduced the fundamental idea of a *semaphore*.

5.3 Protecting shared variables with semaphores

A *semaphore*, s, is a global variable with a nonnegative integer value that can only be manipulated by two special operations, called P and V:

- $P(s)$: `while (s <= 0); s--;`
- $V(s)$: `s++;`

The names P and V come from the Dutch *Proberen* (to test) and *Verhogen* (to increment). The P operation waits for the semaphore s to become nonzero, and then decrements it. The V operation increments s. The test and decrement operations in P occur indivisibly, in the sense that once the predicate $s <= 0$ becomes false, the decrement occurs without interruption. The increment operation in V also occurs indivisibly, in that it loads, increments, and stores the semaphore without interruption.

The definitions of P and V ensure that a running program can never enter a state where a properly initialized semaphore has a negative value. This property, known as the *semaphore invariant*, provides a powerful tool for controlling the trajectories of concurrent programs so that they avoid unsafe regions.

The basic idea is to associate a semaphore s, initially 1, with each shared variable (or related set of shared variables) and then surround the corresponding critical section with $P(s)$ and $V(s)$ operations.[2]

For example, the progress graph in Figure 16 shows how we would use semaphores to properly synchronize our example counter program. In the figure, each state is labeled with the value of semaphore s in that state. The crucial idea is that this combination of P and V operations creates a collection of states, called a *forbidden region*, where $s < 0$. Because of the semaphore invariant, no feasible trajectory can include one of the states in the forbidden region. And since the forbidden region completely encloses the unsafe region, no feasible trajectory can touch any part of the unsafe region. Thus, every feasible trajectory is safe, and regardless of the ordering of the instructions at runtime, the program correctly increments the counter.

5.4 Posix semaphores

The Posix standard defines a number of functions for manipulating semaphores. This section describes the three basic functions, `sem_init`, `sem_wait` (P), and `sem_post` (V).

A program initializes a semaphore by calling the `sem_init` function.

[2] A semaphore that is used in this way to protect shared variables is called a *binary semaphore* because its value is always 0 or 1.

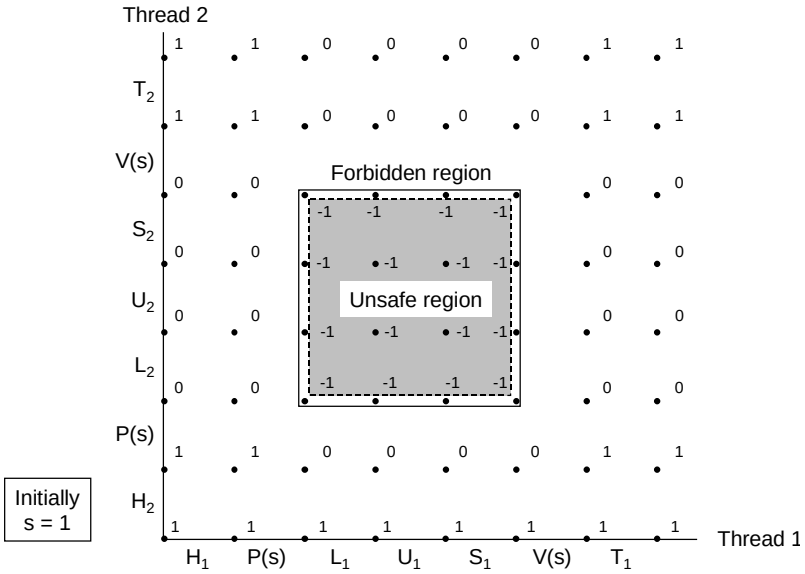

Figure 16: **Safe sharing with semaphores.** The states where $s < 0$ define a *forbidden region* that surrounds the unsafe region.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```
returns: 0 if OK, -1 on error

The `sem_init` function initializes semaphore `sem` to `value`. Each semaphore must be initialized before it can be used. If `sem` is being used to synchronize concurrent threads associated with the same process, then `pshared` is zero. If `sem` is being used to synchronize concurrent processes (not discussed here), then `pshared` is nonzero. We use Posix semaphores only in the context of concurrent threads, so `pshared` is 0 in all of our examples.

Programs perform P and V operations on semaphore `sem` by calling the `sem_wait` and `sem_post` functions, respectively.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```
returns: 0 if OK, -1 on error

Figure 17 shows a version of the threaded counter program from Figure 8, called `goodcnt.c`, that uses semaphore operations to properly synchronize access to the shared counter variable. The code follows directly from Figure 16, so there are just a few aspects of it to point out:

- First, in a convention dating back to Dijkstra's original semaphore paper, a binary semaphore used for

../code/threads/goodcnt.c

```
 1 #include "ics.h"
 2
 3 #define NITERS 10000000
 4
 5 void *count(void *arg);
 6
 7 /* shared variables */
 8 unsigned int cnt; /* counter */
 9 sem_t sem;        /* semaphore */
10
11 int main()
12 {
13     pthread_t tid1, tid2;
14
15     Sem_init(&sem, 0, 1);
16
17     Pthread_create(&tid1, NULL, count, NULL);
18     Pthread_create(&tid2, NULL, count, NULL);
19
20     Pthread_join(tid1, NULL);
21     Pthread_join(tid2, NULL);
22
23     if (cnt != (unsigned)NITERS*2)
24         printf("BOOM! cnt=%d\n", cnt);
25     else
26         printf("OK cnt=%d\n", cnt);
27     exit(0);
28 }
29
30 /* thread routine */
31 void *count(void *arg)
32 {
33     int i;
34
35     for (i=0; i<NITERS; i++) {
36         P(&sem);
37         cnt++;
38         V(&sem);
39     }
40     return NULL;
41 }
```

../code/threads/goodcnt.c

Figure 17: `goodcnt.c`: **A properly synchronized version of** `badcnt.c`**.**

safe sharing is often called a *mutex* because it provides each thread with *mutually exclusive access* to the shared data. We have followed this convention in our code.

- Second, the `Sem_init`, `P`, and `V` functions are Unix-style error-handling wrappers for the `sem_init`, `sem_wait`, and `sem_post` functions, respectively.

5.5 Signaling with semaphores

We saw in the previous section how semaphores can be used for sharing. But semaphores can also be used for *signaling*. In this scenario, a thread uses a semaphore operation to notify another thread when some condition in the program state becomes true. A classic example is the `producer-consumer` interaction shown in Figure 18. A producer thread and a consumer thread share a buffer with n slots. The producer thread repeatedly produces new items and inserts them in the buffer. The consumer thread repeatedly removes items from the buffer and then consumes them. Other variants allow different combinations of single and multiple producers and consumers.

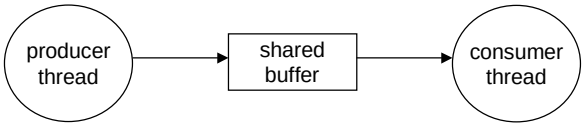

Figure 18: **Producer-consumer model.**

Producer-consumer interactions are common in real systems. For example, in a multimedia system, the producer might encode video frames while the consumer decodes and renders them on the screen. The purpose of the buffer is to reduce jitter in the video stream caused by data-dependent differences in the encoding and decoding times for individual frames. The buffer provides a reservoir of slots to the producer and a reservoir of encoded frames to the consumer. Another common example is the design of graphical user interfaces. The producer detects mouse and keyboard events and inserts them in the buffer. The consumer removes the events from the buffer in some priority-based manner and paints the screen.

Figure 19 outlines how we would use Posix semaphores to synchronize the producer and consumer threads in a simple producer-consumer program where the buffer can hold at most one item.

We use two semaphores, `empty` and `full` to characterize the state of the buffer. The `empty` semaphore indicates that the buffer contains no valid items. It is initialized to the initial number of available empty buffer slots (1). The `full` semaphore indicates that the buffer contains a valid item. It is initialized to the initial number of valid items (0).

The producer thread produces an item (in this case a simple integer), then waits for the buffer to be empty with a P operation on semaphore `empty`. After the producer writes the item to the buffer, it informs the consumer that there is now a valid item with a V operation on `full`. Conversely, the consumer thread waits for a valid item with a P operation on `full`. After reading the item, it signals that the buffer is empty with a V operation on `empty`.

The impact at run-time is that the producer and consumer ping-pong back and forth, as shown in Figure 21.

../code/threads/prodcons.c

```
1 #include "ics.h"
2
3 #define NITERS 5
4
5 void *producer(void *arg), *consumer(void *arg);
6
7 struct {
8   int buf;           /* shared variable */
9   sem_t full, empty; /* semaphores */
10 } shared;
11
12
13 int main()
14 {
15   pthread_t tid_producer, tid_consumer;
16
17   /* initialize the semaphores */
18   Sem_init(&shared.empty, 0, 1);
19   Sem_init(&shared.full,  0, 0);
20
21   /* create threads and wait for them to finish */
22   Pthread_create(&tid_producer, NULL, producer, NULL);
23   Pthread_create(&tid_consumer, NULL, consumer, NULL);
24   Pthread_join(tid_producer, NULL);
25   Pthread_join(tid_consumer, NULL);
26
27   exit(0);
28 }
```

../code/threads/prodcons.c

Figure 19: **Producer-consumer program: Main routine.** One producer thread and one consumer thread manipulate a 1-item buffer. Initially, `empty == 1` and `full == 0`.

../code/threads/prodcons.c

```
1  /* producer thread */
2  void *producer(void *arg)
3  {
4    int i, item;
5
6    for (i=0; i<NITERS; i++) {
7      /* produce item */
8      item = i;
9      printf("produced %d\n", item);
10
11     /* write item to buf */
12     P(&shared.empty);
13     shared.buf = item;
14     V(&shared.full);
15   }
16   return NULL;
17 }
18
19 /* consumer thread */
20 void *consumer(void *arg)
21 {
22   int i, item;
23
24   for (i=0; i<NITERS; i++) {
25     /* read item from buf */
26     P(&shared.full);
27     item = shared.buf;
28     V(&shared.empty);
29
30     /* consume item */
31     printf("consumed %d\n", item);
32   }
33   return NULL;
34 }
```

../code/threads/prodcons.c

Figure 20: **Producer-consumer program: Producer and consumer threads.**

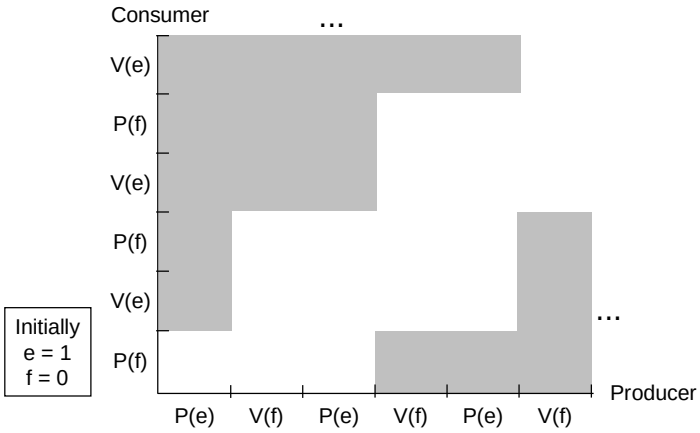

Figure 21: **Progress graph for** `prodcons.c`.

Problem 6 [Category 3]:
Generalize the producer-consumer program in Figure 19 to manipulate a circular buffer with a capacity of BUFSIZE integer items. The producer generates NITEMS integer items: $0, 1, \ldots,$ NITEMS $- 1$. For each item, it works for a while (i.e., `Sleep(rand()%MAXRAND)`), produces the item by printing a message, and then inserts the item at the rear of the buffer. The consumer repeatedly removes an item from the front of the buffer, works for a while, and then consumes the item by printing a message. For example:

```
linux> prodconsn
produced 0
produced 1
consumed 0
produced 2
consumed 1
consumed 2
produced 3
produced 4
consumed 3
consumed 4
```

6 Synchronizing threads with mutex and condition variables

As an alternative to P and V operations on semaphores, Pthreads provides a family of synchronization operations on *mutex* and *condition* variables. In general, we prefer semaphores over their Pthreads counterparts because they are more elegant and simpler to reason about. However, there are some useful synchronization patterns, such as timeout waiting, that are impossible to implement with semaphores. Thus, it is worthwhile to have some facility with the Pthreads operations.

In the previous section, we learned that semaphores can be used for both sharing *and* signaling. Pthreads, on the other hand, provides one set of functions (based on mutex variables) for sharing, and another set (based on condition variables) for signaling.

6.1 Mutex variables

A *mutex* is synchronization variable that is used like a binary semaphore to protect the access to shared variables. There are three basic operations defined on a mutex.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

return: 0 if OK, nonzero on error

A mutex must be initialized before it can be used, either by at run-time by calling the `pthread_init` function, or at compile-time:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

For our purposes, the `attr` argument in `pthread_init` will always be NULL and can be safely ignored.

The `pthread_mutex_lock` function performs a P operation and the `pthread_mutex_unlock` function performs a V operation. Completing the call to `pthread_mutex_lock` is referred to as *acquiring the mutex*, and completing the call to `pthread_mutex_unlock` is referred to as *releasing the mutex*. At any point in time, at most one thread can hold the mutex.

6.2 Condition variables

Condition variables are synchronization variables that are used for signaling. Pthreads defines three basic operations on condition variables.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

return: 0 if OK, nonzero on error

A condition variable `cond` must be initialized before it is used, either by calling `pthread_cond_init` or at compile-time:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

For our purposes the `attr` argument will always be NULL and can be safely ignored.

A thread waits for some program condition associated with the condition variable `cond` to become true by calling `pthread_cond_wait`. In order to guarantee that the call to `pthread_cond_wait` is indivisible with respect to other instances of `pthread_cond_wait` and `pthread_cond_signal`, Pthreads requires that a mutex variable `mutex` be associated with the condition variable `cond`, and that a call to `pthread_mutex_wait` must always be protected by that mutex:

```
Pthread_mutex_lock(&mutex);
Pthread_cond_wait(&cond, &mutex);
Pthread_mutex_unlock(&mutex);
```

The call to `pthread_cond_wait` puts the thread to sleep, waiting for `cond` to become true, and then releases the mutex.

A thread indicates that the condition associated with `cond` has become true by making a call to the `pthread_cond_signal` function. As before, the call must be protected by the mutex associated with `cond`:

```
Pthread_mutex_lock(&mutex);
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&mutex);
```

If there are any threads waiting on condition `cond`, then the call to `pthread_cond_signal` wakes up exactly one of them. The thread that wakes up as a result of the signal reacquires the mutex and then returns from the call to `pthread_cond_wait`.

If no threads are currently waiting on condition `cond`, then nothing happens. Thus, like Unix signals, and unlike Posix semaphores, *Pthreads signals are not queued*, which makes them much harder to reason about than semaphore operations.

6.3 Barrier synchronization

In general, we find that synchronizing with mutex and condition variables is more difficult and cumbersome than with semaphores. However, a barrier is an example of a synchronization pattern that can be expressed quite elegantly with operations on mutex and condition variables.

A *barrier* is a function, `void barrier(void)`, that returns to the caller only when every other thread has also called `barrier`. Barriers are essential in concurrent programs whose threads must progress at roughly the same rate. For example, we use threads in our research to implement parallel simulations of earthquakes. The duration of the earthquake (say 60 seconds) is broken up into thousands of tiny timesteps. Each thread runs on a separate processor and models the propagation of seismic waves through some chunk of the earth, first for timestep 1, then for timestep 2, and so on. In order to get consistent results, each thread must finish simulating timestep k before the others can begin simulating timestep $k + 1$. We guarantee this by placing a barrier between the execution of each timestep.

Our barrier implementation uses a signaling variant called `pthread_cond_broadcast` that wakes up *every* thread currently waiting on condition variable `cond`.

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
                                                    returns: 0 if OK, nonzero on error
```

Figure 22 shows the code for a simple barrier package based on mutex and condition variables.

../code/threads/barrier.c

```
 1 #include "ics.h"
 2
 3 static pthread_mutex_t mutex;
 4 static pthread_cond_t cond;
 5
 6 static int nthreads;
 7 static int barriercnt = 0;
 8
 9 void barrier_init(int n)
10 {
11     nthreads = n;
12     Pthread_mutex_init(&mutex, NULL);
13     Pthread_cond_init(&cond, NULL);
14 }
15
16 void barrier()
17 {
18     Pthread_mutex_lock(&mutex);
19     if (++barriercnt == nthreads) {
20         barriercnt = 0;
21         Pthread_cond_broadcast(&cond);
22     }
23     else
24         Pthread_cond_wait(&cond, &mutex);
25     Pthread_mutex_unlock(&mutex);
26 }
```

../code/threads/barrier.c

Figure 22: `barrier.c`**: A simple barrier synchronization package.**

The barrier package uses 4 global variables that are defined in lines 2–7. The variables are declared with the `static` attribute so that they are not visible to other object modules. `Cond` is a condition variable and `mutex` is its associated mutex variable. `Nthreads` records the number of threads involved in the barrier, and `barriercnt` keeps track of the number of threads that have called the `barrier` function.

The `barrier_init` function in lines 9–14 is called once by the main thread, before it creates any other threads.

The mutex that surrounds the body of the `barrier` function guarantees that it is executed indivisibly and

in some total order by each thread. Thus, once the current thread has acquired the mutex in line 18, there are only two possibilities.

1. If the current thread is the last to enter the barrier, then it clears the barrier count for the next time (line 20), and wakes up all of the other threads (line 21). We know that all of the other threads are asleep, waiting on a signal, because the current thread is the last to enter the barrier.

2. If the current thread is not the last thread, then it goes to sleep and releases the mutex (line 24) so that other threads can enter the barrier.

Problem 7 [Category 3]:
Write a driver program `barriermain.c` that tests the barrier package in Figure 22. The driver accepts a command line argument n, calls the `barrier_init` function, and then creates n threads that repeatedly synchronize by printing a message and calling the `barrier`. For example,

```
bass> barrier 2
1026: hello from barrier 0
2051: hello from barrier 0
1026: hello from barrier 1
2051: hello from barrier 1
1026: hello from barrier 2
2051: hello from barrier 2
1026: hello from barrier 3
2051: hello from barrier 3
1026: hello from barrier 4
2051: hello from barrier 4
```

Problem 8 [Category 3]:
Write a barrier synchronization package, with the same interface as the package in Figure 22, that uses semaphores instead of Pthreads mutex and condition variables.

6.4 Timeout waiting

Sometimes when we write a concurrent program, we are only willing to wait a finite amount of time for a particular condition to become true. Since a P operation can block indefinitely, this kind of *timeout waiting* is not possible to implement with semaphore operations. However, Pthreads provides this capability, in the form of the `pthread_cond_timedwait` function.

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           struct timespec *abstime);
                                                  returns: 0 if OK, ETIMEDOUT if timeout
```

The `pthread_cond_timedwait` function behaves like the `pthread_cond_wait` function, except that it returns with an error code of ETIMEDOUT once the value of the system clock exceeds the absolute time value in `abstime`. Figure 23 shows a handy routine that a thread can use to build the `abstime` argument each time it calls `pthread_cond_timedwait`:

../code/threads/maketimeout.c

```
 1 #include "ics.h"
 2
 3 struct timespec *maketimeout(struct timespec *tp, int secs)
 4 {
 5     struct timeval now;
 6
 7     gettimeofday(&now, NULL);
 8     tp->tv_sec = now.tv_sec + secs;
 9     tp->tv_nsec = now.tv_usec * 1000;
10     return tp;
11 }
```

../code/threads/maketimeout.c

Figure 23: `maketimeout`**: Builds a timeout structure for** `pthread_cond_timedwait`**.**

For a simple example of timeout waiting in a threaded program, suppose we want to write a *beeping timebomb* that waits at most 5 seconds for the user to hit the return key, printing out "BEEP" every second. If the user doesn't hit the return key in time, then the program explodes by printing "BOOM!". Otherwise, it prints "Whew!" and exits. Figure 24 shows a threaded timebomb that is based on the `pthread_cond_timedwait` function.

The main timebomb thread locks the mutex and then creates a peer thread that calls `getchar`, which blocks the thread until the user hits the return key. When `getchar` returns, the peer thread signals the main thread that the user has hit the return key, and then terminates. Notice that since the main thread locked the mutex before creating the peer thread, the peer thread cannot acquire the mutex and signal the main thread until the main thread releases the mutex by calling `pthread_cond_timedwait`.

Meanwhile, after the main thread creates the peer thread, it waits up to one second for the peer thread to terminate. If `pthread_cond_timedwait` does not time out, then the main thread knows that the peer thread has terminated, so it prints "Whew!" and exits. Otherwise, it beeps and waits for another second. This continues until it has waited a total of 5 seconds, at which point the loop terminates, the main thread explodes by printing "Boom!", and then exits.

../code/threads/timebomb.c

```
 1 #include "ics.h"
 2
 3 #define TIMEOUT 5
 4
 5 void *thread(void *vargp);
 6 struct timespec *maketimeout(struct timespec *tp, int secs);
 7
 8 pthread_cond_t cond;
 9 pthread_mutex_t mutex;
10 pthread_t tid;
11
12 int main()
13 {
14     int i, rc;
15     struct timespec timeout;
16
17     Pthread_cond_init(&cond, NULL);
18     Pthread_mutex_init(&mutex, NULL);
19
20     Pthread_mutex_lock(&mutex);
21     Pthread_create(&tid, NULL, thread, NULL);
22     for (i=0; i<TIMEOUT; i++) {
23         printf("BEEP\n");
24         rc = pthread_cond_timedwait(&cond, &mutex,
25                                     maketimeout(&timeout, 1));
26         if (rc != ETIMEDOUT) {
27             printf("WHEW!\n");
28             exit(0);
29         }
30     }
31     printf("BOOM!\n");
32     exit(0);
33 }
34
35 /* thread routine */
36 void *thread(void *vargp)
37 {
38     getchar();
39     Pthread_mutex_lock(&mutex);
40     Pthread_cond_signal(&cond);
41     Pthread_mutex_unlock(&mutex);
42     return NULL;
43 }
```

../code/threads/timebomb.c

Figure 24: `timebomb.c`: **A beeping timebomb that explodes unless the user hits a key within 5 seconds.**

Problem 9 [Category 3]:
Implement a threaded version of the C `fgets` function, called `tfgets`, that times out and returns a NULL pointer if it does not receive an input line on stdin within 5 seconds.

- Your function should be implemented in a package called `tfgets-thread.c`.
- Your solution may use any Pthreads function.
- Your solution may not use the Linux `sleep` or `alarm` functions.
- Test your solution using the following driver program:

../code/threads/tfgets-main.c

```
1 #include "ics.h"
2
3 char *tfgets(char *s, int size, FILE *stream);
4
5 int main()
6 {
7     char buf[MAXLINE];
8
9     if (tfgets(buf, MAXLINE, stdin) == NULL)
10        printf("BOOM!\n");
11    else
12        printf("%s", buf);
13
14    exit(0);
15 }
```

../code/threads/tfgets-main.c

Problem 10 [Category 3]:
For an interesting contrast in concurrency models, implement `tfgets` using processes, signals, and nonlocal jumps instead of threads.

- Your function should be implemented in a package called `tfgets-proc.c`.
- Your solution may not use the Linux `alarm` function.
- Test your solution using the driver program from Problem 9.

7 Thread-safe and reentrant functions

When we program with threads, we must be careful to write functions that are thread-safe. A function is *thread-safe* if and only if it will always produce correct results when called repeatedly within multiple

concurrent threads. If a function is not thread-safe, then it is said to be *thread-unsafe.* We can identify four (non-disjoint) classes of thread-unsafe functions:

1. *Failing to protect shared variables.* We have already encountered this problem with the `count` function in Figure 8 that increments an unprotected global counter variable.

   This class of thread-unsafe function is relatively easy to make thread-safe: protect the shared variables with the appropriate synchronization operations (e.g., P and V functions or Pthreads lock and unlock functions). An advantage is that it does not require any changes in the calling program. A disadvantage is that the synchronization operations will slow down the function.

2. *Relying on state across multiple function invocations.* A pseudo-random number generator is a good example of this class of thread-unsafe function. Consider the `rand` package from [5]:

   ```
   unsigned int next = 1;

   /* rand - return pseudo-random integer on 0..32767 */
   int rand(void)
   {
       next = next * 1103515245 + 12345;
       return (unsigned int)(next/65536) % 32768;
   }

   /* srand - set seed for rand() */
   void srand(unsigned int seed)
   {
       next = seed;
   }
   ```

   The `rand` function is thread-unsafe because the result of the current invocation depends on an intermediate result from the previous iteration. When we call `rand` repeatedly from a single thread after seeding it with a call to `srand`, we can expect a repeatable sequence of numbers. However, this assumption no longer holds if multiple threads are calling `rand`.

   The only way to make a function such as `rand` thread-safe is to rewrite it so that it does not use any static data, relying instead on the caller to pass the state information in arguments. The disadvantage is that the programmer is now forced to change the code in the calling routine as well. In a large program where there are potentially hundreds of different call sites, making such modifications could be non-trivial and error-prone.

3. *Returning a pointer to a static variable.* Some functions, such as `gethostbyname`, compute a result in a `static` structure and then return a pointer to that structure. If we call such functions from concurrent threads, then disaster is likely as results being used by one thread are suddenly overwritten by another thread.

   There are two ways to deal with this class of thread-unsafe function. One option is to rewrite the function so that the caller passes the address of the structure to store the results in. This eliminates all shared data, but requires the programmer to change the code in the caller as well.

If the thread-unsafe function is difficult or impossible to modify (e.g., it is linked from a library), then another option is to use what we call the *lock-and-copy* approach. The idea is to associate a mutex with the thread-unsafe function. At each call site, lock the mutex, call the thread-unsafe function, dynamically allocate memory for the result, copy the result returned by the function to this memory, and then unlock the mutex. An attractive variant is to define a thread-safe wrapper function that performs the lock-and-copy, and then replace all calls to the thread-unsafe function with calls to the wrapper.

4. *Calling thread-unsafe functions.* If a function f calls a thread-unsafe function, then f is thread-unsafe, too.

Thread-safety can be a confusing issue because there is no simple comprehensive rule for distinguishing thread-safe functions from thread-unsafe ones. Although very thread-unsafe function references shared variables (or calls other functions that are thread-unsafe), not every function that references shared data is thread-unsafe. As we have seen, it all depends on how the function uses the shared variables.

7.1 Reentrant functions

There is an important class of thread-safe functions, known as *reentrant functions*, that are characterized by the property that they do not reference any shared data when they are called by multiple threads. Although the terms *thread-safe* and *reentrant* are sometimes incorrectly used as synonyms, there is a clear technical distinction that is worth preserving. Reentrant functions are typically more efficient than non-reentrant thread-safe functions because they require no synchronization operations. Furthermore, as we have seen, sometimes the only way to convert a thread-unsafe function into a thread-safe one is to rewrite it so that it is reentrant.

Figure 25 shows the set relationships between reentrant, thread-safe, and thread-unsafe functions. The set of all functions is partitioned into the disjoint sets of thread-safe and thread-unsafe functions. The set of reentrant functions is a proper subset of the thread-safe functions.

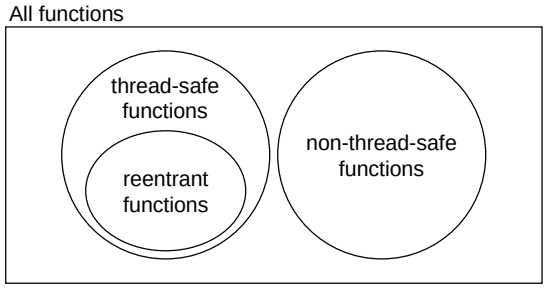

Figure 25: **Relationships between the sets of reentrant, thread-safe, and non-thread-safe functions.**

Is it possible to inspect the code of some function and declare *a priori* that it is reentrant? Unfortunately, it depends. If all function arguments are passed by value (i.e., no pointers) and all data references are to local automatic stack variables (i.e., no references to static or global variables), then the function is *explicitly reentrant*, in the sense that we can assert its reentrancy regardless of how it is called.

However, if we loosen our assumptions a bit and allow some parameters in our otherwise explicitly reentrant function to be passed by reference, then we have an *implicitly reentrant* function, in the sense that it is only reentrant if the calling threads are careful to pass pointers to non-shared data. In the rest of the book, we will use the term *reentrant* to include both explicit and implicit reentrant functions, but it is important to realize that reentrancy is sometimes a property of both the caller and the callee.

To understand the distinctions between thread-unsafe, thread-safe, and reentrant functions more clearly, let's consider different versions of our `maketimeout` function from Figure 23. We will start with the function in Figure 26, a thread-unsafe function that returns a pointer to a static variable.

../code/threads/maketimeout_u.c

```
1 #include "ics.h"
2
3 struct timespec *maketimeout_u(int secs)
4 {
5     static struct timespec timespec;
6     struct timeval now;
7
8     gettimeofday(&now, NULL);
9     timespec.tv_sec = now.tv_sec + secs;
10    timespec.tv_nsec = now.tv_usec * 1000;
11    return &timespec;
12 }
```

../code/threads/maketimeout_u.c

Figure 26: `maketimeout_u`**: A version of** `maketimeout` **that is not thread-safe.**

Figure 27 shows how we might use the lock-and-copy approach to create a thread-safe (but not reentrant) wrapper that the calling program can invoke instead of the original thread-unsafe function.

Finally, we can go all out and rewrite the unsafe function so that it is reentrant, as shown in Figure 28. Notice that the calling thread now has the responsibility of passing an address that points to unshared data.

7.2 Thread-safe library functions

Most Linux functions and the functions defined in the standard C library (such as `malloc`, `free`, `realloc`, `printf`, and `scanf`) are thread-safe, with only a few exceptions. Figure 29 lists the common exceptions. (See [6] for a complete list.)

The `asctime`, `ctime`, and `localtime` functions are commonly used functions for converting back and forth between different time and data formats. The `gethostbyname`, `gethostbyaddr`, and `inet_nota` functions are commonly used network programming functions that we will encounter in the next chapter.

With the exception of `rand`, all of these thread-unsafe functions are of the class-2 variety that return a pointer to a static variable. If we need to call one of these functions in a threaded program, the simplest approach is to lock-and-copy as in Figure 27.

The disadvantage is that the additional synchronization will slow down the program. Further, this approach

../code/threads/maketimeout_t.c

```
1 #include "ics.h"
2 struct timespec *maketimeout_u(int secs);
3
4 static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
5
6 struct timespec *maketimeout_t(int secs)
7 {
8     struct timespec *sp; /* shared */
9     struct timespec *up = Malloc(sizeof(struct timespec)); /* unshared */
10
11    Pthread_mutex_lock(&mutex);
12    sp = maketimeout_u(secs);
13    *up = *sp;    /* copy the shared struct to the unshared one */
14    Pthread_mutex_unlock(&mutex);
15    return up;
16 }
```

../code/threads/maketimeout_t.c

Figure 27: `maketimeout_t`**: A version of** `maketimeout` **that is thread-safe but not reentrant.**

../code/threads/maketimeout_r.c

```
1 #include "ics.h"
2
3 struct timespec *maketimeout_r(struct timespec *tp, int secs)
4 {
5     struct timeval now;
6
7     gettimeofday(&now, NULL);
8     tp->tv_sec = now.tv_sec + secs;
9     tp->tv_nsec = now.tv_usec * 1000;
10    return tp;
11 }
```

../code/threads/maketimeout_r.c

Figure 28: `maketimeout_r`**: A version of** `maketimeout` **that is reentrant and thread-safe.**

Thread-unsafe function	Thread-unsafe class	Linux thread-safe version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

Figure 29: **Common thread-unsafe library functions.**

will not work for a class-2 thread-unsafe function such as `rand` that relies on static state across calls. Therefore, Linux and other Unix systems provide reentrant versions of most thread-unsafe functions that end with the "`_r`" suffix. Unfortunately, these functions are poorly documented and the interfaces differ from system to system. Thus, the "`_r`" interface should not be used unless absolutely necessary.

8 Other synchronization errors

Even if we have managed to make our functions thread-safe, our programs might can still suffer from subtle synchronization errors such as races and deadlocks.

8.1 Races

A *race* occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y. Races usually occur because programmers assume that threads will take some particular trajectory through the execution state space, forgetting the golden rule that threaded programs must work correctly for any feasible trajectory.

An example is the easiest way to understand the nature of races. Consider the simple program in Figure 30. The main thread creates four peer threads and passes a pointer to a unique integer ID to each one. Each peer thread copies the ID passed in its argument to a local variable (line 22), and then prints a message containing the ID.

It looks simple enough, but when we run this program on our system, we get the following incorrect result:

```
linux> race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

The problem is caused by a race between each peer thread and the main thread. Can you spot the race?

Here is what happens. When the main thread creates a peer thread in line 13, it passes a pointer to the local stack variable i. At this point the race is on between the next call to `pthread_create` in line 13 and the

../code/threads/race.c

```
1 #include "ics.h"
2
3 #define N 4
4
5 void *thread(void *vargp);
6
7 int main()
8 {
9     pthread_t tid[N];
10    int i;
11
12    for (i = 0; i < N; i++)
13        Pthread_create(&tid[i], NULL, thread, &i);
14    for (i = 0; i < N; i++)
15        Pthread_join(tid[i], NULL);
16    exit(0);
17 }
18
19 /* thread routine */
20 void *thread(void *vargp)
21 {
22    int myid = *((int *)vargp);
23
24    printf("Hello from thread %d\n", myid);
25    return NULL;
26 }
```

../code/threads/race.c

Figure 30: **A program with a race.**

dereferencing and assignment of the argument in line 22. If the peer thread executes line 22 before the main thread executes line 13, then the `myid` variable gets the correct ID. Otherwise it will contain the ID of some other thread. The scary thing is that whether we get the correct answer depends on how the kernel schedules the execution of the threads. On our system it fails, but on other systems it might work correctly, leaving the programmer blissfully unaware of a serious bug.

To eliminate the race, we can dynamically allocate a separate block for each integer ID, and pass the thread routine a pointer to this block, as shown in Figure 31 (lines 13-15). Notice that the thread routine must free the block in order to avoid a memory leak.

../code/threads/norace.c

```
 1 #include "ics.h"
 2
 3 #define N 4
 4
 5 void *thread(void *vargp);
 6
 7 int main()
 8 {
 9     pthread_t tid[N];
10     int i, *ptr;
11
12     for (i = 0; i < N; i++) {
13         ptr = Malloc(sizeof(int));
14         *ptr = i;
15         Pthread_create(&tid[i], NULL, thread, ptr);
16     }
17     for (i = 0; i < N; i++)
18         Pthread_join(tid[i], NULL);
19     exit(0);
20 }
21
22 /* thread routine */
23 void *thread(void *vargp)
24 {
25     int myid = *((int *)vargp);
26
27     Free(vargp);
28     printf("Hello from thread %d\n", myid);
29     return NULL;
30 }
```

../code/threads/norace.c

Figure 31: **A correct version the program in Figure 30 without a race.**

When we run this program on our system, we now get the correct result:

```
linux> norace
```

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

We will use a similar technique in Chapter **??** when we discuss the design of threaded network servers.

Problem 11 [Category 1]:
In Figure 31, we might be tempted to free the allocated memory block immediately after line 15 in the main thread, instead of freeing it in the peer thread. But this would be a bad idea. Why?

Problem 12 [Category 1]:

A. In Figure 31, we eliminated the race by allocating a separate block for each integer ID. Outline a different approach that does not call the `malloc` or `free` functions.

B. What are the advantages and disadvantages of this approach?

8.2 Deadlocks

Semaphores introduce the potential for a nasty kind of runtime error, called *deadlock*, where a collection of threads are blocked, waiting for a condition that will never be true. The progress graph is an invaluable tool for understanding deadlock. For example, Figure 32 shows the progress graph for a pair of threads that use two semaphores for sharing. From this graph, we can glean some important insights about deadlock:

- The programmer has incorrectly ordered the P and V operations such that the forbidden regions for the two semaphores overlap. If some execution trajectory happens to reach the `deadlock state` d, then no further progress is possible because the overlapping forbidden regions block progress in every legal direction. In other words, the program is because each thread is waiting for the other to do a V operation that will never occur.

- The overlapping forbidden regions induce a set of states, called the *deadlock region*. If a trajectory happens to touch a state in the deadlock region, then deadlock is inevitable. Trajectories can enter deadlock regions, but they can never leave.

- Deadlock is an especially difficult problem because it is not always predictable. Some lucky execution trajectories will skirt the deadlock region, while others will be trapped by it. Figure 32 shows an example of each. The implications for a programmer are somewhat scary. You might run the same program 1000 times without any problem, but then the next time it deadlocks. Or the program might work fine on one machine but deadlock on another. Worst of all, the error is often not repeatable because different executions have different trajectories.

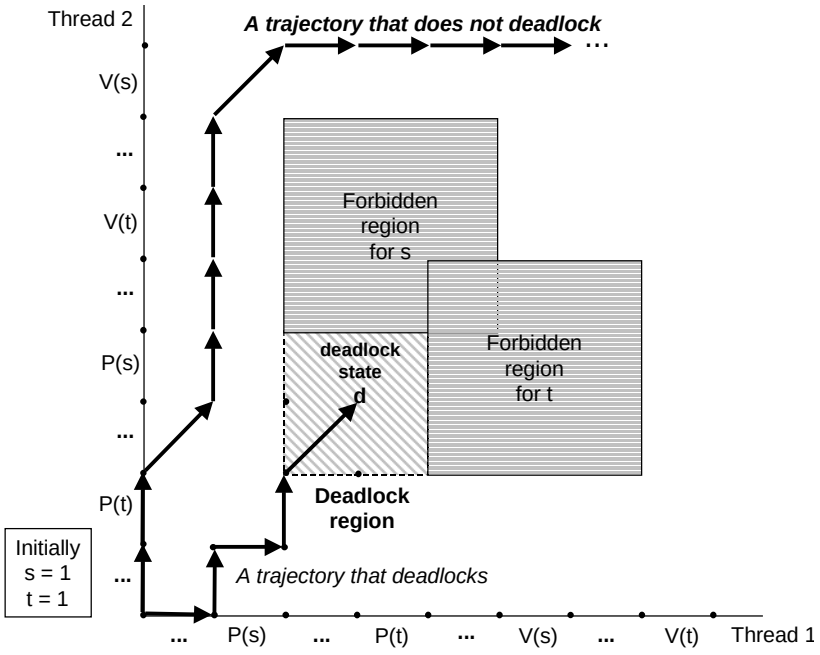

Figure 32: **Progress graph for a program that can deadlock.**

A number of formal techniques have been proposed in the literature for detecting and avoiding deadlocks, but none are much help for the practicing Pthreads programmer. The best advice we can offer is to be alert to potential deadlocks when you use semaphores.

Problem 13 [Category 1]:
Consider the following program, which uses a pair of semaphores for sharing.

```
Initially: s = 1, t = 0.

Thread 1:       Thread 2:
   P(s);           P(s);
   V(s);           V(s);
   P(t);           P(t);
   V(t);           V(t);
```

A. Sketch the progress graph and forbidden regions for this program.

B. Which four states form the corners of the deadlock region?

C. Does the program always deadlock?

D. What simple change to the initial semaphore values will fix the deadlock?

E. Sketch the progress graph of this new version.

9 Summary

Threads are a popular and useful tool for introducing concurrency in programs. Threads are typically more efficient than processes, and it is much easier to share data between threads than between processes. However, the ease of sharing introduces the possibility of synchronization errors that are difficult to diagnose.

Programmers writing threaded programs must be careful to protect shared data with the appropriate synchronization mechanisms. Functions called by threads must be thread-safe. Races and deadlocks must be avoided. In sum, the wise programmer approaches the design of threaded programs with great care and not a little trepidation.

Bibliographic notes

Semaphore operations were proposed by Dijkstra [4]. The progress graph concept was introduced by Coffman [3] and later formalized by Carson and Reynolds [2]. The book by Butenhof [1] contains a comprehensive description of the Posix threads interface.

References

[1] D. Butenhof. *Programming with Posix Threads.* Addison-Wesley, 1997.

[2] S. Carson and P. Reynolds. The geometry of semaphore programs. *ACM Transactions on Programming Languages and Systems*, 9(1):25–53, 1987.

[3] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[4] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[5] B. Kernighan and D. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[6] W. Richard Stevens. *Unix Network Programming: Networking APIs (Second Edition)*, volume 1. Prentice Hall, 1998.