Chapter 5: Achieving Good Performance

Typically, it is fairly straightforward to reason about the performance of sequential computations. For most programs, it suffices simply to count the number of instructions that are executed. In some cases, we realize that memory system performance is the bottleneck, so we find ways to reduce memory usage or to improve memory locality. In general, programmers are encouraged to avoid premature optimization by remembering the 90/10 rule, which states that 90% of the time is spent in 10% of the code. Thus, a prudent strategy is to write a program in a clean manner, and if its performance needs improving, to identify the 10% of the code that dominates the execution time. This 10% can then be rewritten, perhaps even rewritten in some alternative language, such as assembly code or C.

Unfortunately, the situation is much more complex with parallel programs. As we will see, the factors that determine performance are not just instruction times, but also communication time, waiting time, dependences, etc. Dynamic effects, such as contention, are time-dependent and vary from problem to problem and from machine to machine. Furthermore, controlling the costs is much more complicated. But before considering the complications, consider a fundamental principle of parallel computation.

Amdahl's Law

Amdahl's Law observes that if $1/S$ of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of S. The reasoning is that the parallel execution time, T_P, of a computation with sequential execution time, T_S, will be the sum of the time for the sequential component and the parallel component. For P processors we have

$$T_P = 1/S \cdot T_S + (1-1/S) \cdot T_S / P$$

Imagining a value for P so large that the parallel portion takes negligible time, the maximum performance improvement is a factor of S. That is, the proportion of sequential code in a computation determines its potential for improvement using parallelism.

Given Amdahl's Law, we can see that the 90/10 rule does not work, even if the 90% of the execution time goes to 0. By leaving the 10% of the code unchanged, our execution time is at best 1/10 of the original, and when we use many more than 10 processors, a 10x speedup is likely to be unsatisfactory.

The situation is actually somewhat worse than Amdahl's Law implies. One obvious problem is that the parallelizable portion of the computation might not be improved to an

Amdahl's Law. The "law" was enunciated in a 1967 paper by Gene Amdahl, an IBM mainframe architect [Amdahl, G.M., Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, AFIPS Press 30:483-485, 1967]. It is a "law" in the same sense that the Law of Supply and Demand is a law: It describes a relationship between two components of program execution time, as expressed by the equation given in the text. Both laws are powerful tools to explain the behavior of important phenomena, and both laws assume as constant other quantities that affect the behavior. Amdahl's Law applies to a program instance.

unlimited extent—that is, there is probably an upper limit on the number of processors that can be used and still improve the performance—so the parallel execution time is unlikely to vanish. Furthermore, a parallel implementation often executes more total instruction than the sequential solution, making the (1-1/S) · T_S an under estimate.

Many, including Amdahl, have interpreted the law as proof that applying large numbers of processors to a problem will have limited success, but this seems to contradict news reports in which huge parallel computers improve computations by huge factors. What gives? Amdahl's law describes a key fact that applies to an *instance* of a computation. Portions of a computation that are sequential will, as parallelism is applied, dominate the execution time. The law fixes an *instance*, and considers the effect of increasing parallelism. Most parallel computations, such as those in the news, fix the parallelism and expand the instances. In such cases the proportion of sequential code diminishes relative to the overall problem as larger instances are considered. So, doubling the problem size may increase the sequential portion negligibly, making a greater fraction of the problem available for parallel execution.

In summary, Amdahl's law does not deny the value of parallel computing. Rather, it reminds us that to achieve parallel performance we must be concerned with the entire program.

Measuring Performance

As mentioned repeatedly, the main point of parallel computing is to run computations faster. *Faster* obviously means "in less time," but we immediately wonder, "How much less?" To understand both what is possible and what we can expect to achieve, we use several metrics to measure parallel performance, each with its own strengths and weaknesses.

Execution Time

Perhaps the most intuitive metric is *execution time*. Most of us think of the so called "wall clock" time as synonymous with execution time, and for programs that run for hours and hours, that equivalence is accurate enough. But the elapsed wall clock time includes operating system time for loading and initiating the program, I/O time for reading data, paging time for the compulsory page misses, check-pointing time, etc. For short computations—the kind that we often use when we are analyzing program behavior—these items can be significant contributors to execution time. One argument says that because they are not affected by the user programming, they should be factored out of performance analysis that is directed at understanding the behavior of a parallel solution; the other view says that some services provided by the OS are needed, and the time should be charged. It is a complicated matter that we take up again at the end of the chapter.

In this book we use execution time to refer to the net execution time of a parallel program exclusive of initial OS, I/O, etc. charges. The problem of compulsory page misses is usually handled by running the computation twice and measuring only the second one. When we intend to include all of the components contributing to execution time, we will refer to *wall clock time*.

Notice that execution times (and wall clock times for that matter) cannot be compared if they come from different computers. And, in most cases it is not possible to compare the execution times of programs running different inputs even for the same computer.

FLOPS

Another common metric is FLOPS, short for floating point operations per second, which is often used in scientific computations that are dominated by floating point arithmetic. Because double precision floating point arithmetic is usually significantly more expensive than single precision, it is common when reporting FLOPS to state which type of arithmetic is being measured. An obvious downside to using FLOPS is that it ignores other costs such as integer computations, which may also be a significant component of computation time. Perhaps more significant is that FLOPS rates can often be affected by extremely low-level program modifications that allow the programs to exploit a special feature of the hardware, e.g. a combined multiply/add operation. Such “improvements” typically have little generality, either to other computations or to other computers.

A limitation of both of the above metrics is that they distill all performance into a single number without providing an indication of the parallel behavior of the computation. Instead, we often wish to understand how the performance of the program scales as we change the amount of parallelism.

Speedup

Speedup is defined as the execution time of a sequential program divided by the execution time of a parallel program that computes the same result. In particular, $Speedup = T_S / T_P$, where T_S is the sequential time and T_P is the parallel time running on P processors. Speedup is often plotted on the *y*-axis and the number of processors on the *x*-axis, as shown in Figure 5.1.

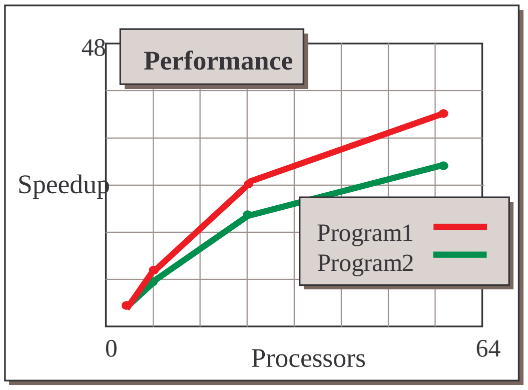


Figure 5.1. A typical speedup graph showing performance for two programs.

The speedup graph shows a characteristic typical of many parallel programs, namely, that the speedup curves level off as we increase the number of processors. This feature is the result of keeping the problem size constant while increasing the number of processors, which causes the amount of work per processor to decrease; with less work per processor costs such as overhead—or sequential computation, as Amdahl predicted—become more significant, causing the total execution not to scale so well.

Efficiency

Efficiency is a normalized measure of speedup: *Efficiency* = *Speedup/P*. Ideally, speedup should scale linearly with *P*, implying that efficiency should have a constant value of 1. Of course, because of various sources of performance loss, efficiency is more typically below 1, and it diminishes as we increase the number of processors. Efficiency greater than 1 represents superlinear speedup.

Superlinear Speedup

The upper curve in the Figure 5.1 graph indicates superlinear speedup, which occurs when speedup grows faster than the number of processors. How is this possible? Surely the sequential program, which is the basis for the speedup computation, could just simulate the *P* processes of the parallel program to achieve an execution time that is no more than *P* times the parallel execution time. Shouldn't superlinear speedup be impossible? There are two reasons why superlinear speedup occurs.

The most common reason is that the computation's working set—that is, the set of pages needed for the computationally intensive part of the program—does not fit in the cache when executed on a single processor, but it does fit into the caches of the multiple processors when the problem is divided amongst them for parallel execution. In such cases the superlinear speedup derives from improved execution time due to the more efficient memory system behavior of the multi-processor execution.

The second case of superlinear speedup occurs when performing a search that is terminated as soon as the desired element is found. When performed in parallel, the search is effectively performed in a different order, implying that the total amount of data searched can actually be less than in the sequential case. Thus, the parallel execution actually performs less work.

Issues with Speedup and Efficiency

Since speedup is a ratio of two execution times, it is a unitless metric that would seem to factor out technological details such as processor speed. Instead, such details insidiously affect speedup, so we must be careful in interpreting speedup figures. There are several concerns.

First, recognize that it is difficult to compare speedup from machines of different generations, even if they have the same architecture. The problem is that different components of a parallel machine are generally improved by different amounts, changing their relative importance. So, for example, processor performance has increased over time, but communication latency has not fallen proportionately. Thus, the time spent communicating will not have diminished as much as the time spent computing. As a result, speedup values have generally decreased over time. Stated another way, the parallel components of a computation have become relatively more expensive compared to the processing components.

The second issue concerns T_S, speedup's numerator, which should be the time for the fastest sequential solution for the given processor and problem size. If T_S is artificially inflated, speedup will be greater. A subtle way to increase T_S is to turn off scalar compiler optimizations for both the sequential and parallel programs, which might seem fair since it is using the same compiler for both programs. However, such a change effectively slows the processors and improves—relatively speaking—communication latency. When reporting speedup, the sequential program should be provided and the compiler optimization settings detailed.

Another common way to increase T_S is to measure the one-processor performance of the *parallel* program. Speedup computed on this basis is called *relative speedup* and should be reported as such. True speedup includes the likely possibility that the sequential algorithm is different than the parallel algorithm. Relative speedup, which simply compares different runs of the same algorithm, takes as the base case an algorithm optimized for concurrent execution but with no parallelism; it will likely run slower because of parallel overheads, causing the speedup to look better. Notice that it can happen that a well-written parallel program on one processor *is* faster than any known sequential program, making it the best sequential program. In such cases we have true speedup, not relative speedup. The situation should be explicitly identified.

Relative speed up cannot always be avoided. For example, for large computations it may be impossible to measure a sequential program on a given problem size, because the data structures do not fit in memory. In such cases relative speedup is all that can be reported. The base case will be a parallel computation on a small number of processors, and the *y*-

axis of the speedup plot should be scaled by that amount. So, for example, if the smallest possible run has P=4, then dividing by the runtime for P=64, will show perfect speedup at y=16.

Another way to inadvertently affect T_S is the "cold start" problem. An easy way to accidentally get a large T_S value is to run the sequential program once and include all of the paging behavior and compulsory cache misses in its timing. As noted earlier it is good practice to run a parallel computation a few times, measuring only the later runs. This allows the caches to "warm up," so that compulsory cache miss times are not unnecessarily included in the performance measure, thereby complicating our understanding of the program's speedup. (Of course, if the program has conflict misses, they should and will be counted.) Properly, most analysts "warm" their programs. But the sequential program should be "warmed," too, so that the paging and compulsory misses do not figure into its execution time. Though easily overlooked, cold starts are also easily corrected.

More worrisome are computations that involve considerable off-processor activity, e.g. disk I/O. One-time I/O bursts, say to read in problem data, are fine because timing measurements can by-pass them; the problem is continual off-processor operations. Not only are they slow relative to the processors, but they greatly complicate the speedup analysis of a computation. For example, if both the sequential and parallel solutions have to perform the same off-processor operations from a single source, huge times for these operations can completely obscure the parallelism because they will dominate the measurements. In such cases it is not necessary to parallelize the program at all. If processors can independently perform the off-processor operations, then this parallelism alone dominates the speedup computation, which will likely look perfect. Any measurements of a computation involving off-processor charges must control their effects carefully.

Performance Trade-Offs

We know that communication time, idle time, wait time, and many other quantities can affect the performance of a parallel computation. The complicating factor is that attempts to lower one cost can increase others. In this section we consider such complications.

Communication vs. computation

Communication costs are a direct expense for using parallelism because they do not arise in sequential computing. Accordingly, it is almost always smart to attempt to reduce them.

Overlap Communication and Computation. One way to reduce communication costs is to overlap communication with computation. Because communication can be performed concurrently with computation, and because the computation must be performed anyway, a perfect overlap—that is, the data is available when it is needed—hides the communication cost perfectly. Partial overlap will diminish waiting time and give partial improvement. The key, of course, is to identify computation that is independent of the communication. From a performance perspective, overlapping is generally a win without costs. From a programming perspective, overlapping communication and computation can complicate the program's structure.

Redundant Computation. Another way to reduce communication costs is to perform redundant computations. We observed in Chapter 2, for example, that the local generation of a random number, *r*, by all processes was superior to generating the value in one process and requiring all others to reference it. Unlike overlapping, redundant computation incurs a cost because there is no parallelism when all processors must execute the random number generator code. Stated another way, we have increased the total number of instructions to be executed in order to remove the communication cost. Whenever the cost of the redundant computation is less than the communication cost, redundant computation is a win.

Notice that redundant computation *also removes a dependence* from the original program between the generating process and the others that will need the value. It is useful to remove dependences even if the cost of the added computation exactly matches the communication cost. In the case of the random number generation, redundant computation removes the possibility that a client process will have to wait for the server process to produce it. If the client can generate its own random number, it does not have to wait. Such cases complicate the assessing the trade-off.

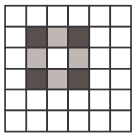
Memory vs. parallelism

Memory usage and parallelism interact in many ways. Perhaps the most favorable is the "cache effect" that leads to superlinear parallel performance, noted above. With all processors having caches, there is more fast memory in a parallel computer. But there are other cases where memory and parallelism interact.

Privatization. For example, parallelism can be increased by using additional memory to break false dependences. One memorable example is the use of `private_count` variables in the Count 3s program, which removed the need for threads to interact each time they recorded the next 3. The effect was to increase the number of count variables from 1 to `t`, the number of threads. It is a tiny memory cost for a big savings in reduced dependences

Batching. One way to reduce the number of dependences is to increase the granularity of interaction. Batching is a programming technique in which work or transmissions are performed as a group. For example, rather than transmitting elements of an array, transmit a whole row or column; rather than grabbing one task from the task queue, get several. Batching effectively raises the granularity (see below) of fine-grain interactions to reduce their frequency. The added memory is simply required to record the items of the batch, and like privatization, is almost always worth the memory costs.

Memoization. Memoization stores a computed value to avoid re-computing later. An example is a stencil optimization: A value is computed based on some combination of the scaled values of its neighbors, shown schematically below,



where color indicates the scaling coefficient; elements such as the corner elements are multiplied by the scale factor four times as the stencil "moves through the array," and memoizing this value can reduce the number of multiplies and memory references. [DETAILED EXAMPLE HERE.] It is a sensible program optimization that removes instruction executions that, strictly speaking, may not result in parallelism improvements. However, in many cases memoization will result in better parallelism, as when the computation is redundant or involves non-local data values.

Padding. Finally, we note that false sharing—references to independent variables that become dependent because they are allocated to the same cache line—can be eliminated by padding data structures to push the values onto different cache lines.

Overhead vs. parallelism

Parallelism and overhead are sometimes at odds. At one extreme, all parallel overhead, such as lock contention, can be avoided by using just one process. As we increase the number of threads the contention will likely increase. If the problem size remains fixed each processor has less work to perform between synchronizations, causing synchronization to become a larger portion of the overall computation. And a smaller problem size implies that there is less computation available to overlap with communication, which will typically increase the wait times for data.

It is the overhead of parallelism that is usually the reason why P cannot increase without bound. Indeed, even computations that could conceptually be solved with a processor devoted to each data point will be buried by overhead before $P=n$. Thus, we find that most programs have an upper limit for each data size at which the marginal value of an additional processor is negative, that is, adding a processor causes the execution time to increase.

Parallelize Overhead. Recall that in Chapter 4, when lock contention became a serious concern, we adopted a combining tree to solve it. In essence, the threads split up the task of accumulating intermediate values into several independent parallel activities.

[THIS SECTION CONTINUES WITH THESE TOPICS]

Load balance vs. parallelism. Increased parallelism can also improve load balance, as it's often easier to distribute evenly a large number of fine-grained units of work than a smaller number of coarse-grained units of work.

Granularity tradeoffs. Many of the above tradeoffs are related to the granularity of parallelism. The best granularity often depends on both algorithmic characteristics, such as the amount of parallelism and the types of dependences, and hardware characteristics, such as the cache size, the cache line size, and the latency and bandwidth of the machine's communication substrate.

Latency vs. bandwidth. As discussed in Chapter 3, there are many instances where bandwidth can be used to reduce latency.

Scaled speedup vs. Fixed-Size speedup
Choosing a problem size can be difficult**.**

What should we measure?

The kernel or the entire program?
Amdahl’s law says that everything is important!

Operating System Costs
Because operating systems are so integral to computation, it is complicated to assess their effects on performance.

Initialization.
How is memory laid out in the parallel computer?

Summary

Exercises

Chapter 6: Programming with Threads

Recall in Chapter 1 that we used threads to implement the count 3's program. In this chapter we'll explore thread-based programming in more detail using the standard POSIX Threads interface. We'll first explain the basic concepts needed to create threads and to let them interact with one another. We'll then discuss issues of safety and performance before we step back and evaluate the overall approach.

Thread Creation and Destruction

Consider the following standard code:

```
 1 #include <pthread.h>
 2 int err;
 3
 4 void main ()
 5 {
 6    pthread_t tid[MAX];   /* An array of Thread ID's, one for each */
 7                          /* thread that is created */
 8
 9    for (i=0; i<t; i++)
10    {
11       err = pthread_create (&tid[i], NULL, count3s_thread, i);
12    }
13
14    for (i=0; i<t; i++)
15    {
16       err = pthread_join_(tid[i], &status[i])
17    }
18 }
```

The above code shows a `main()` function, which then creates—and launches—`t` threads in the first loop, and then waits for the `t` threads to complete in the second loop. We often refer to the creating thread as the ***parent*** and the created threads as ***children***.

The above code differs from the pseudocode in Chapter 1 in a few details. Line 1 includes the pthreads header file, which declares the various pthreads routines and datatypes. Each thread that is created needs its own thread ID, so these thread ID's are declared on line 6. To create a thread, we invoke the `pthread_create()` routine with four parameters. The first parameter is a pointer to a thread ID, which will point to a valid thread ID when this thread successfully returns. The second argument provides the thread's attributes; in this case, the NULL value specifies default attributes. The third parameter is a pointer to the start function, which the thread will execute once it's created. The fourth argument is passed to the start routine, in this case, it represents a unique integer between `0` and `t-1` that is associated with each thread. The loop on line 16 then calls `pthread_join()` to wait for each of the child threads to terminate. If instead of waiting for the child threads to complete, the `main()` routine finishes and exits using `pthread_exit()`, the child threads will continue to execute. Otherwise, the child threads will automatically terminate when `main()` finishes, since the entire process will have terminated. See Code Specs 1 and 2.

pthread_create()

```
int pthread_create (                // create a new thread
  pthread_t *tid,                   // thread ID
  const pthread_attr_t *attr,       // thread attributes
  void *(*start_routine) (void *),// pointer to function to execute
  void *arg                         // argument to function
  );
```

Arguments:

- The thread ID of the successfully created thread.
- The thread's attributes, explained below; the `NULL` value specifies default attributes.
- The function that the new thread will execute once it is created.
- An argument passed to the `start_routine(),` in this case, it represents a unique integer between `0` and `t-1` that is associated with each thread.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- Use a structure to pass multiple arguments to the start routine.

Code Spec 1. pthread_create(). The POSIX Threads thread creation function.

pthread_join()

```
int pthread_join (                   // wait for a thread to terminate
  pthread_t tid,                     // thread IT to wait for
  void **status                      // exit status
);
```

Arguments:

- The ID of the thread to wait for.
- The completion status of the exiting thread will be copied into `*status` unless status is `NULL`, in which case the completion status is not copied.

Return value:

- 0 for success. Error code from <errno.h> otherwise.

Notes:

- Once a thread is joined, the thread no longer exists, its thread ID is no longer valid, and it cannot be joined with any other thread.

Code Spec 2. pthread_join(). The POSIX Threads rendezvous function pthread_join().

Thread ID's

Each thread has a unique ID of type `pthread_t`. As with all pthread data types, a thread ID should be treated as an ***opaque type***, meaning that individual fields of the structure should never be accessed directly. Because child threads do not know their thread ID, the two routines allow a thread to determine its thread ID, pthread_self(), and to compare two thread ID's, pthread_equal(), see Code Specs 3 and 4.

pthread_self()

```
pthread_t pthread_self ();         // Get my thread ID
```

Return value:

- The ID of the thread that called this function.

Code Spec 3. pthread_self(). The POSIX Threads function to fetch a thread's ID.

pthread_equal()

```
int pthread_equal (                  // Test for equality
  pthread_t t1,                      // First operand thread ID
  pthread_t t2                       // Second operand thread ID
);
```

Arguments:

- Two thread ID's

Return value:

- Non-zero if the two thread ID's are the same (following the C convention).
- 0 if the two threads are different.

Code Spec 4. pthread_equal(). The POSIX Threads function to compare two thread IDs for equality.

Destroying Threads

There are three ways that threads can terminate.

1. A thread can return from the start routine.
2. A thread can call `pthread_exit()`.
3. A thread can be ***cancelled*** by another thread.

In each case, the thread is destroyed and its resources become unavailable.

void pthread_exit()

```
void pthread_exit (                    // terminate a thread
  void *status                         // completion status
);
```

Arguments:

- The completion status of the thread that has exited. This pointer value is available to other threads.

Return value:

- None

Notes:

- When a thread exits by simply returning from the start routine, the thread's completion status is set to the start routine's return value.

Code Spec 5. pthread_exit(). The POSIX Threads thread termination function pthread_exit().

Thread Attributes

Each thread maintains its own properties, known as attributes, which are stored in a structure of type `pthread_attr_t`. For example, threads can be either ***detached*** or ***joinable***. Detached threads cannot be joined with other threads, so they have slightly lower overhead in some implementations of POSIX Threads. For parallel computing, we will rarely need detached threads. Threads can also be either ***bound*** or ***unbound***. Bound threads are scheduled by the operating system, whereas unbound threads are scheduled by the Pthreads library. For parallel computing, we typically use bound threads so that each thread provides physical concurrency.

POSIX Threads provides routines to initialize thread attributes, set their attributes, and destroy attributes, as shown in Code Spec 6.

Code Spec 6. pthread attributes. An example of how thread attributes are set in the POSIX Threads

Thread Attributes

```
pthread_attr_t  attr;              // Declare a thread attribute
pthread_t       tid;

pthread_attr_init(&attr);          // Initialize a thread attribute
pthread_attr_setdetachstate(&attr, // Set the thread attribute
                            PTHREAD_CREATE_UNDETACHED);

pthread_create (&tid, &attr, start_func, NULL); // Use the attribute
                                                // to create a thread
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);       // Destroy the thread attribute
```

interface.

Example

The following example illustrates a potential pitfall that can occur because of the interaction between parent and child threads. The parent thread simply creates a child thread and waits for the child to exit. The child thread does some useful work and then exits, returning an error code. Do you see what's wrong with this code?

```
 1 #include <pthread.h>
 2
 3 void main ()
 4 {
 5    pthread_t tid;
 6    int *status;
 7
 8    pthread_create (&tid, NULL, start, NULL);
 9    pthread_join_(tid, &status);
10 }
11
12 void start()
13 {
14    int errorcode;
15    /* do something useful. . . */
16
17    if (. . . )
18       errorcode = something;
19    pthread_exit(&errorcode);
20 }
```

The problem occurs in the call to `pthread_exit()` on line 17, where the child is attempting to return an error code to the parent. Unfortunately, because `errorcode` is declared to be local to the `start()` function, the memory for `errorcode` is allocated on the child thread's stack. When the child exits, its call stack is de-allocated, and the parent has a dangling pointer to `errorcode`. At some point in the future, when a new procedure is invoked, it will over-write the stack location where `errorcode` resides, and the value of `errorcode` will change.

Mutual Exclusion

We can now create and destroy threads, but to allow threads to interact constructively, we need methods of coordinating their interaction. In particular, when two threads share access to memory, it is often useful to employ a lock, called a ***mutex***, to provide *mutual exclusion* or mutually exclusive access to the variable. As we saw in Chapter 1, without mutual exclusion, race conditions can lead to unpredictable results, because when multiple threads execute the following code, the `count` variable, which is shared among all threads, will not be atomically updated.

```
for (i=start; i<start+length_per_thread; i++)
{
  if (array[i] == 3)
  {
      count++;
      }
    }
```

The solution, of course, is to protect the update of count using a mutex, as shown below:

```
 1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
 2
 3 void count3s_thread (int id)
 4 {
 5    /* Compute portion of array that this thread should work on */
 6    int length_per_thread = length/t;
 7    int start = id * length_per_thread;
 8
 9    for (i=start; i<start+length_per_thread; i++)
10    {
11       if (array[i] == 3)
12       {
13          pthread_mutex_lock(&lock);
14          count++;
15          pthread_mutex_unlock(&lock);
16       }
17    }
18 }
```

Line 1 shows how a mutex can be statically declared. Like threads, mutexes have attributes, and by initializing the mutex to `PTHREAD_MUTEX_INITIALIZER`, the mutex is assigned default attributes. To use this mutex, its address is passed to the lock and unlock routines on lines 13 and 15, respectively. The appropriate discipline, of course, is to bracket all ***critical sections***, that is, code that must be executed atomically by only one thread at a time, by the locking of a mutex upon entrance and the unlocking of a mutex upon exit. Only one thread can acquire the mutex at any one time, so a thread will block if it attempts to acquire a mutex that is already held by another thread. When a mutex is unlocked, or ***relinquished***, one of the threads that was blocked attempting to acquire the lock will become unblocked and granted the mutex. The POSIX Threads standard defines no notion of fairness, so the order in which the locks are acquired is *not* guaranteed to match the order in which the threads attempted to acquire the locks.

It is an error to unlock a mutex that has not been locked, and it is an error to lock a mutex that is already held. The latter will lead to ***deadlock***, in which the thread cannot make progress because it is blocked waiting for an event that cannot happen. We will discuss deadlock and techniques to avoid deadlock in more detail later in the chapter.

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(             // Lock a mutex
   pthread_mutex_t *mutex);

int pthread_mutex_unlock(           // Unlock a mutex
   pthread_mutex_t *mutex);

int pthread_mutex_trylock(          // Non-blocking lock
   pthread_mutex_t *mutex);
```

Arguments:

- Each function takes the address of a mutex variable.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns `EBUSY` if the mutex is locked.

Code Spec 7. The POSIX Threads routines for acquiring and releasing mutexes.

Serializability

It's clear that our use of mutexes provides atomicity: the thread that acquires the mutex `m` will execute the code in the critical section until it relinquishes the mutex. Thus, in our above example, the counter will be updated by only one thread at a time. Atomicity is important because it ensures ***serializability***: A concurrent execution is serializable if the execution is guaranteed to execute in an order that corresponds to *some* serial execution of those threads.

Mutex Creation and Destruction

In our above example, we knew that only one mutex was needed, so we were able to statically allocate it. In cases where the number of required mutexes is not known *a priori*, we can instead allocate and deallocate mutexes dynamically. Code Spec 8 shows how such a mutex is dynamically allocated, initialized with default attributes, and destroyed.

Mutex Creation and Destruction

```
int pthread_mutex_init(             // Initialize a mutex
   pthread_mutex_t *mutex,
   pthread_mutexattr_t *attr);

int pthread_mutex_destroy (         // Destroy a mutex
   pthread_mutex_t *mutex);

int pthread_mutexattr_init(         // Initialize a mutex attribute
   pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy (     // Destroy a mutex attribute
   pthread_mutexattr_t *attr);
```

Arguments:

- The `pthread_mutex_init()` routine takes two arguments, a pointer to a mutex and a pointer to a mutex attribute. The latter is presumed to have already been initialized.
- The `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` routines take a pointer to a mutex attribute as arguments.

Notes:

- If the second argument to `pthread_mutex_init()` is `NULL`, default attributes will be used.

Code Spec 8. The POSIX Threads routines for dynamically creating and destroying mutexes.

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;             // Declare a pointer to a lock

lock = (pthread_mutex_lock_t *) malloc(sizeof (pthread_mutex_t));

pthread_mutex_init(lock, NULL);
   /*
    * Code that uses this lock.
    */

pthread_mutex_destroy (lock);
free (lock);
```

Code Spec 9. An example of how dynamically allocated mutexes are used in the POSIX Threads interface.

Synchronization

Mutexes are sufficient to provide atomicity for critical sections, but in many situations we would like a thread to synchronize its behavior with that of some other thread. For example, consider a classic bounded buffer problem in which one or more threads put items into a circular buffer while other threads remove items from the same buffer. As shown in Figure 1, we would like the producers to stop producing data—to wait—if the consumer is unable to keep up and the buffer becomes full, and we would like the consumers to wait if the buffer is empty.

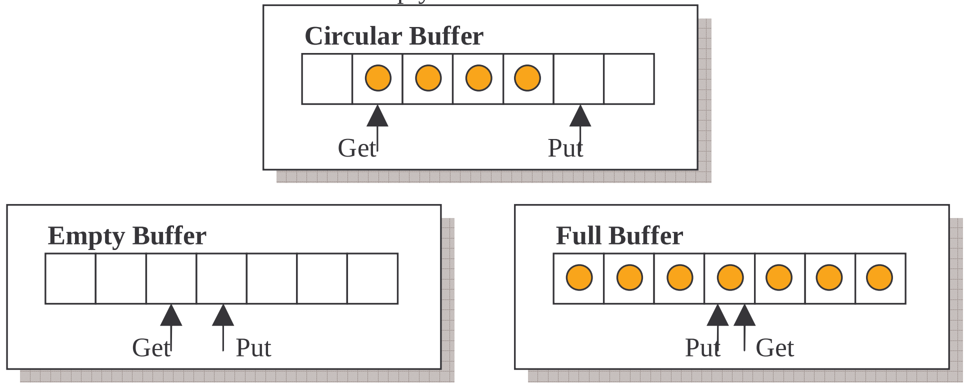


Figure 1. A bounded buffer with producers and consumers. The Put and Get cursors indicate where the producers will insert the next item and where the consumers will remove its next item, respectively. When the buffer is empty, the consumers must wait. When the buffer is full, the producers must wait.

Such synchronization is supported by ***condition variables***, which are a more general form of synchronization than joining threads. A condition variable allows threads to wait until some condition becomes true, at which point one of the waiting threads is non-deterministically chosen to stop waiting. We can think of the condition variable as a gate (see Figure 2). Threads wait at the gate until some condition is true. Other threads open the gate to signal that the condition has become true, at which point one of the waiters is allowed to enter the gate and resume execution. If a thread opens the gate when there are no threads waiting, the signal has no effect.

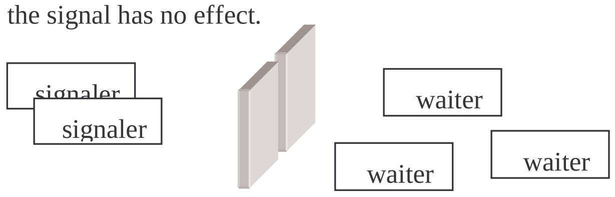


Figure 2. Condition variables act like a gate. Threads wait outside the gate by calling `pthread_cond_wait()`, and threads open the gate by calling `pthread_cond_signal()`. When the gate is opened, one waiter is allowed through. If there are no waiters when the gate is opened, the signal has no effect.

We can solve our bounded buffer problem with two condition variables, `nonempty` and `nonfull`, as shown below.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty = PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull= PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int in = 0;                        // Buffer index for next insertion
int out = 0;                       // Buffer index for next removal
```

```
 7
 8 void put (Item x)                // Producer thread
 9 {
10    pthread_mutex_lock(&lock);
11    while (in – out) == SIZE)     // While buffer is full
12       pthread_cond_wait(&nonfull, &lock);
13    buffer[in % SIZE] = x;
14    in++;
15    pthread_cond_signal(&nonempty);
16    pthread_mutex_unlock(&lock);
17 }
18
19 Item get()                       // Consumer thread
20 {
21    Item x;
22    pthread_mutex_lock(&lock);
23    while (out – in)              // While buffer is empty
24       pthread_cond_wait(&nonempty, &lock);
25    x = buffer[out % SIZE];
26    out++;
27    pthread_cond_signal(&nonfull);
28    pthread_mutex_unlock(&lock);
29    return x;
30 }
```

Of course, since multiple threads will be updating these condition variables, we need to protect their access with a mutex, so Line 1 declares a mutex. The remaining declarations define a buffer, `buffer`, and its two cursors, `in` and `out`, which indicate where to insert the next item and where to remove the next item. The two cursors wrap around when they exceed the bounds of `buffer`, yielding a circular buffer.

Given these data structures, the producer thread executes the `put()` routine, which first acquires the mutex to access the condition variables. (This code omits the actual creation of the producer and consumer threads, which are assumed to iteratively invoke the `put()` and `get()` routines, respectively.) If the buffer is full, the producer waits on the `nonfull` condition so that it will later be awakened when the buffer becomes non-full. If this thread blocks, the mutex that it holds must be relinquished to avoid deadlock. Because these two events—the releasing of the mutex and the blocking of this waiting thread—must occur atomically, they must be performed by `pthread_cond_wait(),` so the mutex is passed as a parameter to `pthread_cond_wait().` When the producer resumes execution after returning from the wait on Line 12, the protecting mutex will have been re-acquired by the system on behalf of the producer.

In a moment we will explain the need for the `while` loop on Line 11, but for now assume when the producer executes Line 13, the buffer is not full, so it is safe to insert a new item and to bump the `In` cursor by one. At this point, the buffer cannot be empty because the producer has just inserted an element, so the producer signals that the buffer is nonempty, waking one more consumers that may be waiting on an empty buffer. If there are no waiting consumers, the signal is lost. Finally, the producer releases the mutex and exits the routine. The consumer thread executes the `get()` routine, which operates in a very similar manner.

pthread_cond_wait()

```
int pthread_cond_wait(
   pthread_cond_t *cond,            // Condition to wait on
   pthread_mutex_t *mutex);         // Protecting mutex

int pthread_cond_timedwait (
   pthread_cond_t *cond,
   pthread_mutex_t *mutex,
 _ const struct timespec *abstime); // Time-out value
```

Arguments:

- A condition variable to wait on.
- A mutex that protects access to the condition variable. The mutex is released before the thread blocks, and these two actions occur atomically. When this thread is later unblocked, the mutex is reacquired on behalf of this thread.
-

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Code Spec 10. pthread_cond_wait(): The POSIX Thread routines for waiting on condition variables.

pthread_cond_signal()

```
int pthread_cond_signal(
   pthread_cond_t *cond);           // Condition to signal

int pthread_cond_broadcast (
   pthread_cond_t *cond);           // Condition to signal
```

Arguments:

- A condition variable to signal.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

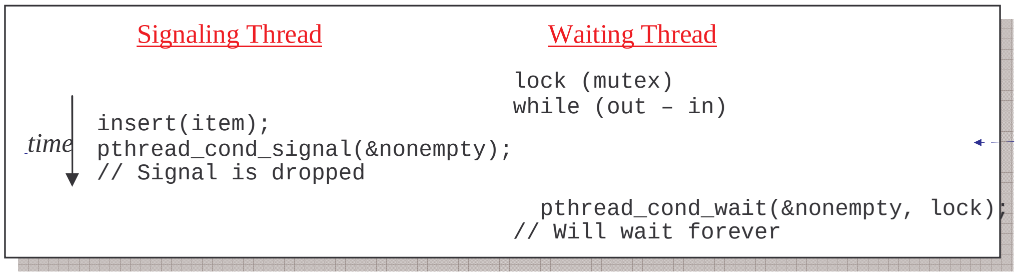
- These routines have no effect if there are no threads waiting on `cond`. In particular, there is no memory of the signal when a later call is made to `pthread_cond_wait().`
- The `pthread_cond_signal()` routine may wake up more than one thread, but only one of these threads will hold the protecting mutex.
- The `pthread_cond_broadcast()` routine wakes up all waiting threads. Only one awakened thread will hold the protecting mutex.

Code Spec 11. pthread_cond_signal(). The POSIX Threads routines for signaling a condition variable.

Protecting Condition Variables

Let us now return to the `while` loop on Line 11 of the bounded buffer program. If our system has multiple producer threads, this loop is essential because `pthread_cond_signal()` can wake up multiple waiting threads[1], of which only one will hold the protecting mutex at any particular time. Thus, at the time of the signal, the buffer is not full, but when any particular thread acquires the mutex, the buffer may have become full again, in which case the thread should call `pthread_cond_wait()` again. When the producer thread executes Line 13, the buffer is necessarily not full, so it is safe to insert a new item and to bump the `In` cursor.

We see on Lines 15 and 27 that the call to `pthread_cond_signal()` is also protected by the lock. The following example shows that this protection is necessary.



```
                                        lock (mutex)
                                        while (out – in)
insert(item);
pthread_cond_signal(&nonempty);
// Signal is dropped
                                          pthread_cond_wait(&nonempty, lock);
                                        // Will wait forever
```

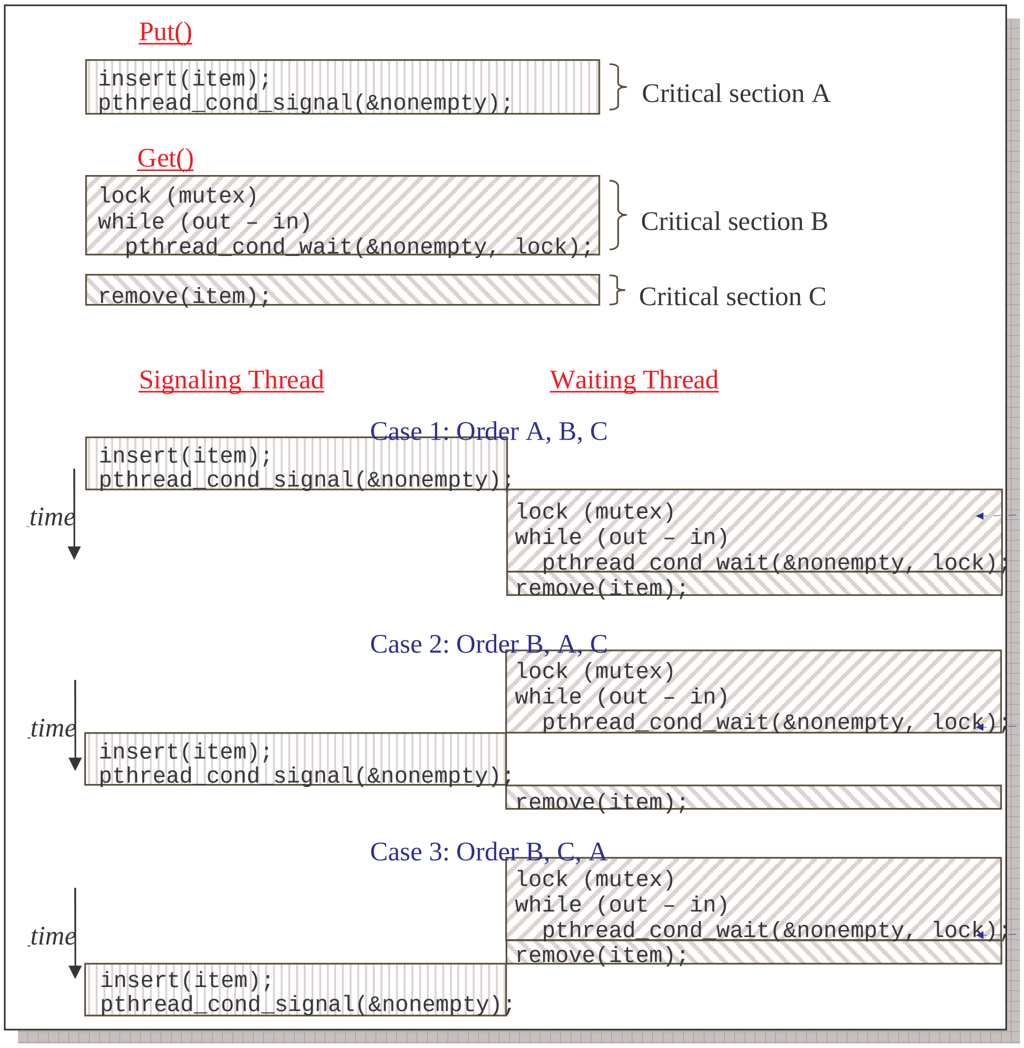
Formatted: Bullets and Numbering

Figure 3. Example of why a signaling thread needs to be protected by a mutex.

In this example, the waiting thread, in this case the consumer, acquires the protecting mutex and finds that the buffer is empty, so it executes `pthread_cond_wait().` If the signaling thread, in this case the producer, does not protect the call to `pthread_cond_signal()` with a mutex, it could insert an item into the buffer immediately after the waiting thread found it empty. If the producer then signals that the buffer is non-empty before the waiting thread executes the call to `pthread_cond_wait(),` the signal will be dropped and the consumer thread will not realize that the buffer is actually not empty. In the case that the producer only inserts a single item, the waiting thread will needlessly wait forever.

The problem, of course, is that there is a race condition involving the manipulation of the buffer. The obvious solution is to protect both the call to `pthread_cond_signal()` with the same mutex that protects the call to `pthread_cond_wait`(), as shown in the code for our bounded buffer solution. Because both the `Put()` and `Get()` routines are protected by the same mutex, we have three critical sections related to the nonempty buffer, as shown in Figure 4, and in no case can the signal be dropped while a waiting thread thinks that the buffer is empty.

[1] These semantics are due to implementation details. In some cases it can be expensive to ensure that exactly one waiter is unblocked by a signal.



```
insert(item);
pthread_cond_signal(&nonempty);
```

```
lock (mutex)
while (out – in)
   pthread_cond_wait(&nonempty, lock);
```

```
remove(item);
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Figure 4. Proper locking of the signaling code prevents race conditions. By identifying and protecting three critical sections pertaining to the nonempty buffer, we guarantee that each of A, B, and C will execute atomically, so our problem from Figure 3 is avoided: There is no way for the `Put()` routine's signal to be dropped while a thread executing the `Get()` routine thinks that the buffer is empty.

We have argued that the call to `pthread_cond_signal()`must be protected by the same mutex that protects the waiting code. However, notice that the race condition occurs not from the signaling of the condition variable, but with the access to the shared buffer. Thus, we could instead simply protect any code that manipulates the shared buffer, which implies that the `Put()`code could release the mutex immediately after inserting an item into the buffer but before calling `pthread_cond_signal()`. This new code is not only legal, but it produces better performance because it reduces the size of the critical section, thereby allowing more concurrency.

Creating and Destroying Condition Variables

Like threads and mutexes, condition variables can be created and destroyed either statically or dynamically. In our bounded buffer example above, the static condition variables were both given default attributes by initializing them to `PTHREAD_COND_INITIALIZER`. Condition variables can be dynamically allocated as indicated in Code Spec 12.

Dynamically Allocated Condition Variables

```
int pthread_cond_init(
   pthread_cond_t *cond,            // Condition variable
   const pthread_condattr_t *attr); // Condition attribute

int pthread_cond_destroy (
   pthread_cond_t *cond);           // Condition to destroy
```

Arguments:

- Default attributes are used if `attr` is `NULL`.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Code Spec 12. The POSIX Threads routines for dynamically creating and destroying condition variables.

Waiting on Multiple Condition Variables

In some cases a piece of code cannot execute unless multiple conditions are met. In these situations the waiting thread should test all conditions simultaneously, as shown below.

```
EatJuicyFruit()
{
   pthread_mutex_lock(&lock);
   while (apples==0 && oranges==0)
   {
      pthread_cond_wait(&more_apples, &lock);
      pthread_cond_wait(&more_oranges, &lock);
   }
   /* Eat both an apple and an orange */
   pthread_mutex_unlock(&lock);
}
```

By contrast, the following code, which waits on each condition in turn, fails because there is no guarantee that both conditions will be true at the same time. That is, after returning from the first call to `pthread_cond_wait()` but before returning from the second call to `pthread_cond_wait`(), some other thread may have removed an apple, making the first condition false.

```
EatJuicyFruit()
{
   pthread_mutex_lock(&lock);
   while (apples==0)
      pthread_cond_wait(&more_apples, &lock);
   while (oranges==0)
      pthread_cond_wait(&more_oranges, &lock);

   /* Eat both an apple and an orange */
   pthread_mutex_unlock(&lock);
}
```

Thread-Specific Data

It is often useful for threads to maintain private data that is not shared. For example, we have seen examples where a thread index is passed to the start function so that the thread knows what portion of an array to work on. This index can be used to give each thread a different element of an array, as shown below:

```
 . . .

   for (i=0; i<t; i++)
      err = pthread_create (&tid[i], NULL, start_function, i);

void start_function(int index)
{
   private_count[index] = 0;
. . .
```

A problem occurs, however, if the code that accesses `index` occurs in a function, `foo(),` which is buried deep within other code. In such situations, how does `foo()` get the value of `index`? One solution is to pass the `index` parameter to every procedure that calls `foo(),` including procedures that call `foo()` indirectly through other procedures. This solution is cumbersome, particularly for those procedures that require the parameter but do not directly use it.

Instead, what we really want is a variable that is global in scope to all code but which can have different values for each thread. POSIX Threads supports such a notion in the form of ***thread-specific data***, which uses a set of ***keys***, which are shared by all threads in a process, but which map to different pointer values for each thread. (See Figure 4.)

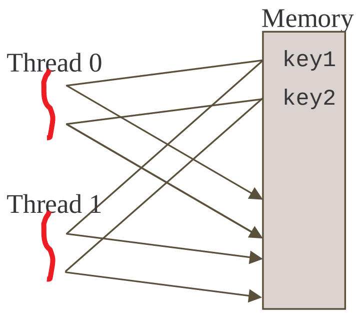


Figure 5. Example of thread-specific data in POSIX Threads. Thread-specific data are accessed by keys, which map to different memory locations in different threads.

As a special case, the error values for POSIX Threads routines are returned in thread-specific data, but such data does not use the interface defined by Code Specs 13-17. Instead, each thread has its own value of `errno`.

Thread-Specific Data

```
pthread_key_t *my_index;
#define index (pthread_getspecific (my_index))

main()
{
   . . .
   pthread_key_create(&my_index, 0);
   . . .
}

void start_routine(int id)
{
   pthread_setspecific (my_index, id);
   . . .
}
```

Notes:

- Avoid accessing index in a tight inner loop because each access requires a procedure call.

Code Spec 13. Example of how thread-specific data is used. Once initialized with this code, any procedure can access the value of `my_index`.

pthread_key_create

```
int pthread_key_create (
   pthread_key_t *key,              // The key to create
   void (*destructor) (void*));     // Destructor function
```

Arguments:

- A pointer to the key to create.
- A destructor function. `NULL` indicates no destructor.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- Avoid accessing index in a tight inner loop because each access requires a procedure call.

Code Spec 14. pthread_key_create. POSIX Thread routine for creating a key for thread-specific data.

pthread_key_delete

```
int pthread_key_delete (
   pthread_key_t *key);              // The key to delete
```

Arguments:
- A pointer to the key to delete.

Return value:
- 0 if successful. Error code from <errno.h> otherwise.

Notes:
- Destructors will not be called.

Code Spec 15. pthread_key_delete. POSIX Thread routine for deletin a key.

pthread_setspecific

```
int pthread_setspecific (
   pthread_key_t *key,              // Key to set
   void *value));                   // Value to set
```

Arguments:
- A pointer to the key to be set.
- The value to set.

Return value:
- 0 if successful. Error code from <errno.h> otherwise.

Notes:
- It is an error to call `pthread_setspecific()` before the key has been created or after the key has been deleted.

Code Spec 16. pthread_setspecific. POSIX Thread routine for setting the value of thread-specific data.

pthread_getspecific

```
int pthread_getspecific (
   pthread_key_t *key);             // Key to value
```

Arguments:

- Key whose value is to be retrieved.

Return value:

- Value of `key` for the calling thread.

Notes:

- The behavior is undefined if a thread calls `pthread_getspecific()` before the key is created or after the key is deleted.

Code Spec 17. pthread_getspecific. POSIX Thread routine for getting the value of some thread-specific data.

Safety Issues

Many types of errors can occur from the improper use of locks and condition variables. We've already mentioned the problem of double-locking, which occurs when a thread attempts to acquire a lock that it already holds. Of course, problems also arise if a thread accesses some shared variable without locking it, or if a thread acquires a lock and does not relinquish it. One particularly important problem is that of avoiding deadlock. This section discusses various methods of avoiding deadlock and other potential bugs.

Deadlock

There are four necessary conditions for deadlock:

1. ***Mutual exclusion***: a resource can be assigned to at most one thread.
2. ***Hold and wait***: a thread that holds resources can request new resources.
3. ***No preemption***: a resource that is assigned to a thread can only be released by the thread that holds it.
4. ***Circular wait***: there is a cycle in which each thread waits for a resource that is assigned to another thread. (See Figure 6.)

Of course, for threads-based programming, mutexes are resources that can cause deadlock. There are two general approaches to dealing with deadlock: (1) prevent deadlocks, and (2) allow deadlock to occur, but detect their occurrence and then break the deadlock. We will focus on the deadlock avoidance, because POSIX Threads does not provide a mechanism for breaking locks.

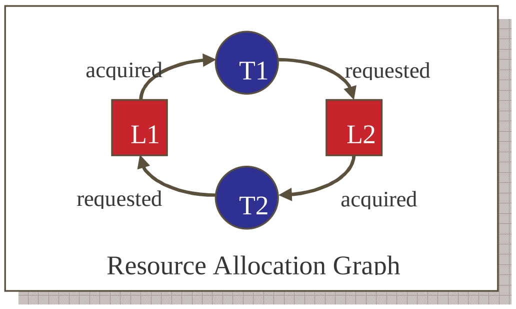


Figure 6. Deadlock example. Threads T1 and T2 hold locks L1 and L2, respectively, and each thread attempts to acquire the other lock, which cannot be granted.

Lock Hierarchies

A simple way to prevent deadlocks is to prevent cycles in the resource allocation graph. We can prevent cycles by imposing an ordering on the locks and by requiring all threads to acquire their locks in the same order. Such a discipline is known as a ***lock hierarchy.***

One problem with a lock hierarchy is that it requires programmers to know *a priori* what locks a thread needs to acquire. Suppose that after acquiring locks L1, L3, and L7, a thread finds that it needs to also acquire lock L2, which would violate the lock hierarchy. One solution would be for the thread to release locks L3 and L7, and then reacquire locks L2, L3, and L7 in that order. Of course, this strict adherence to the lock hierarchy is expensive. A better solution would be to attempt to lock L2 using `pthread_mutex_trylock()`(see Code Spec 7), which either obtains the lock or immediately returns without blocking. If the thread is unable to obtain lock L2, it must resort to the first solution.

Monitors

The use of locks and condition variables is error prone because it relies on programmer discipline. An alternative is to provide language support, which would allow a compiler to enforce mutual exclusion and proper synchronization. A ***monitor*** is one such language construct, and although almost no modern language provides such a construct, it can be implemented in an object oriented setting, as we will soon see. A monitor encapsulates code and data and enforces a protocol that ensures mutual exclusion. In particular, a monitor has a set of well-defined entry points, its data can only be accessed by code that resides inside the monitor, and at most one thread can execute the monitor's code at any time. Monitors also provide condition variables for signaling and waiting, and they ensure that the use of these condition variables obeys the monitor's protocol. Figure 7 shows a graphical depiction of a monitor.

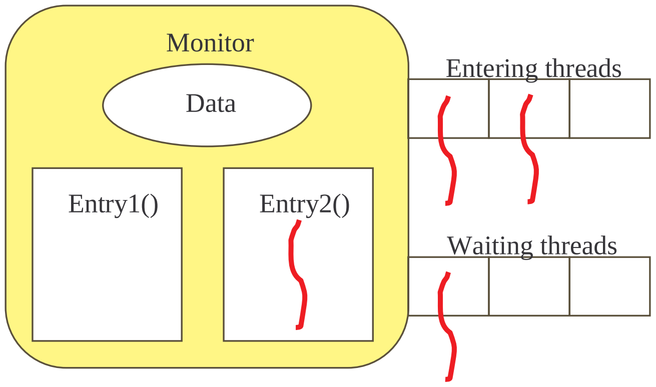


Figure 7. Monitors provide an abstraction of synchronization in which only one thread can access the monitor's data at any time. Other threads are blocked either waiting to enter the monitor or waiting on events inside the monitor.

We can implement monitors in an object oriented language, such as C++, as shown below.

```
class BoundedBuffer
{                                 // Emulate a monitor
private:
   pthread_mutex_t lock;          // Synchronization variables
   pthread_cond_t nonempty, nonfull;
   Item *buffer;                  // Shared data
   int in, out;                   // Cursors
   CheckInvariant();

public:
   BoundedBuffer(int size);       // Constructor
   ~BoundedBuffer();              // Destructor
   void put(Item x);
   Item get();
}

// Constructor and Destructor
BoundedBuffer::Bounded (int size)
{
   // Initialize synchronization variables
   pthread_mutex_init(&lock, NULL);
   pthread_cond_init(&nonempty, NULL);
   pthread_cond_init(&nonfull, NULL);

   // Initialize the buffer
   buffer = new Item[size];
   in = out = 0;
}

BoundedBuffer::~BoundedBuffer()
{
   pthread_mutex_destroy(&lock);
   pthread_cond_destroy(&nonempty);
   pthread_cond_destroy(&nonfull);
   delete buffer;
}

// Member functions
BoundedBuffer::Put(Item x)
{
   pthread_mutex_lock(&lock);
   while (in – out == size)      // while buffer is full
      pthread_cond_wait(&nonfull, &lock);
   buffer[in%size] = x;
   in++;
   pthread_cond_signal(&nonempty);
   pthread_mutex_unlock(&lock);
}

Item BoundedBuffer::Get()
{
   pthread_mutex_lock(&lock);
   while (in = out)              // while buffer is empty
      pthread_cond_wait(&nonempty, &lock);
   x = buffer[out%size];
   out++;
   pthread_cond_signal(&nonfull);
   pthread_mutex_unlock(&lock);
   return x;
}
```

Monitors not only enforce mutual exclusion, but they provide an abstraction that can simplify how we reason about concurrency. In particular, the limited number of entry points facilitates the preservation of invariants. Monitors have ***internal*** functions and ***external*** functions. Internal functions assume that the monitor lock is held. By contrast, external functions must acquire the monitor lock before executing, so external functions cannot invoke each other. In this setting ***invariants*** are properties that can are assumed to be true upon entry and which must be restored upon exit. These invariants may be violated while the monitor lock is held, but they must be restored before the monitor lock is released. This use of invariants is graphically depicted in Figure 8.

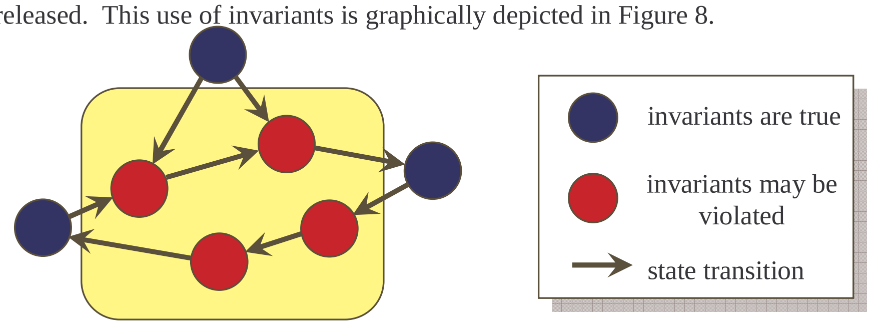


Figure 8. Monitors and invariants. The red circles represent program states in which the invariants may be violated. The blue circles represent program states in which the invariants are assumed to be maintained.

For example, in our bounded buffer example, we have two invariants:

1. The distance between the In and Out cursors is at most the size of the buffer.

2. The In cursor is not left of the Out cursor. (In Figure 1, the Put arrow is not left of the Get arrow.)

Once we have identified our invariants, we can write a routine that checks all invariants, and this routine can be invoked before every entrance to the monitor and after every exit from the monitor. The use of such invariants can be a significant debugging tool. For example, the following code checks these invariants to help debug the monitor's internal routines.

```
BoundedBuffer::CheckInvariant()
{
   if (in – out > size)           // Check invariant (1)
      return (0);
   if (in < out)                  // Check invariant (2)
      return (0);
   return (1);
}

Item BoundedBuffer::Get()
{
   pthread_mutex_lock(&lock);
   assert(CheckInvariant());     // Check on every entrance
   while (in = out)              // while buffer is empty
   {
      assert(CheckInvariant());  // Check on every exit
      pthread_cond_wait(&nonempty, &lock);
      assert(CheckInvariant());
   }
   x = buffer[out%size];
   out++;
   pthread_cond_signal(&nonfull);
   assert(CheckInvariant());
   pthread_mutex_unlock(&lock);
   return x;
}
```

As we have mentioned before, the call to `pthread_cond_wait()` may implicitly release the lock, so it is a potential monitor exit, and the return from `pthread_cond_wait()` will implicitly re-acquire the lock, so it is a monitor entrance.

Re-entrant Monitors

While monitors help enforce a locking discipline, they do not ensure that all concurrency problems go away. For example, if a procedure in a monitor attempts to re-enter the monitor by calling an entry procedure, deadlock will occur. To avoid this problem, the procedure should first restore all invariants, release the monitor lock, and then try to re-enter the monitor. Of course, such a structure means that atomicity is lost. This same problem occurs if a monitor procedure attempts to re-enter the monitor indirectly by calling some external procedure that then tries to enter the monitor, so monitor procedures should invoke external routines with care.

Monitor functions that take a long time or wait for some outside event will prevent other threads from entering the monitor. To avoid such problems, such functions can often be rewritten to wait on a condition, thereby releasing the lock and increasing parallelism. As with re-entrant routines, such functions will need to restore invariants before releasing the lock.

Performance Issues

We saw in Chapter 3 that dependences among threads constrain parallelism. Because locks dynamically impose dependences among threads, the granularity of our locks can greatly affect parallelism. At one extreme, the coarsest locking scheme uses a single lock for all shared variables, which is simple but severely limits concurrency when there is sharing. At the other extreme, fine-grained locks may protect small units of data. For example, in our Count 3's example, we might use a different lock to protect each node of the accumulation tree. As an intermediate point, we might use one lock for the entire accumulation tree. As we reduce the lock granularity, the overhead of locking increases while the amount of available parallelism increases.

Readers and Writers Example: Granularity Issues

Just as there are different granularities for locking, there are different granularities of condition variables. Consider a resource that can be shared by multiple readers or accessed exclusively by a single writer. To coordinate access to such a resource, we can provide four routines—`AcquireExclusive()`, `ReleaseExclusive()`, `AcquireShared()`, and `ReleaseShared()`—that readers and writers can invoke. These routines are each protected by a single mutex, and they collectively use two condition variables. To acquire the resource in exclusive mode, a thread waits on the `wBusy` condition variable, which ensures that no readers are still accessing the resource. When the last reader is done accessing a resource in shared mode, it signals the `wBusy` condition to allow the writer to proceed. Likewise, when a writer is done accessing the resource in exclusive mode, it signals the `rBusy` condition to allow any readers to have access to the resource; and before accessing the shared resource, threads wait on the `rBusy` condition variable.

```
int readers;                     // Negative value => active writer
pthread_mutex_t lock;
pthread_cond_t rBusy, wBusy;     // Use separate condition variables
                                 // for readers and writers
AcquireExclusive()
{
   pthread_mutex_lock(&lock);
   while (readers != 0)
      pthread_cond_wait(&wBusy, &lock);
   readers = -1;
   pthread_mutex_unlock(&lock);
}

AcquireShared()
```

```
{
   pthread_mutex_lock(&lock);
   readWaiters++;
   while (readers<0)
      pthread_cond_wait(&rBusy, &lock);
   readWaiters--;
   pthread_mutex_unlock(&lock);
}

ReleaseExclusive()
{
   pthread_mutex_lock(&lock);
   readers = 0;
   pthread_cond_broadcast(&rBusy);  // Only wake up readers
   pthread_mutex_unlock(&lock);
}

ReleaseShared(
{
   int doSignal;

   pthread_mutex_lock(&lock);
   readers--;
   doSignal = (readers==0)
   pthread_mutex_unlock(&lock);
   if (doSignal)                    // Signal is performed outside
      pthread_cond_signal(&wBusy);  // of critical section
}
```

Two points about this code are noteworthy.

First, the code uses two condition variables, but it's natural to wonder if one condition variable would suffice. In fact, one condition variable could be used, as shown below, and the code would be functionally correct. Unfortunately, by using a single condition variable, the code suffers from ***spurious wakeups*** in which writers can be awoken only to immediately go back to sleep. In particular, when `ReleaseExclusive()` is called both readers and writers are signaled, so writers will suffer spurious wakeups whenever any reader is also waiting on the condition. Our original solution avoids spurious wakeups by using two condition variables, which forces exclusive access and shared access to alternate as long as there is demand for both types of access.

```
int readers;                      // Negative value => active writer
pthread_mutex_t lock;
pthread_cond_t busy;              // Use one condition variable to
                                  // indicate whether data if busy
AcquireExclusive()
{
   pthread_mutex_lock(&lock);     // This code suffers from spurious
   while (readers != 0)           // wakeups!!!
      pthread_cond_wait(&busy, &lock);
   readers = -1;
   pthread_mutex_unlock(&lock);
}

```

```
AcquireShared()
{
   pthread_mutex_lock(&lock);
   while (readers<0)
      pthread_cond_wait(&busy, &lock);
   readers++;
   pthread_mutex_unlock(&lock);
}

ReleaseExclusive()
{
   pthread_mutex_lock(&lock);
   readers = 0;
   pthread_cond_broadcast(&busy);
   pthread_mutex_unlock(&lock);
}

ReleaseShared(
{
   pthread_mutex_lock(&lock);
   readers--;
   if (readers==0)
      pthread_cond_signal(&busy);
   pthread_mutex_unlock(&lock);
}
```

Second, the `ReleaseShared()` routine signals the `wBusy` condition variable outside of the critical section to avoid the problem of ***spurious lock conflicts***, in which a thread is awoken by a signal, executes a few instructions, and then immediately blocks in attempt to acquire the lock. If the `ReleaseShared()`were instead to execute the signal inside of the critical section, as shown below, then any writer that would be awakened would almost immediately block trying to acquire the lock.

```
ReleaseShared(
{
   pthread_mutex_lock(&lock);
   readers--;
   if (readers==0)
      pthread_cond_signal(&wBusy);  // Wake up writers inside of
   pthread_mutex_unlock(&lock);     // the critical section
}
```

The decision to move the signal outside of the critical section represents a tradeoff, because it allows a new reader to enter the critical section before the `ReleaseShared()` routine is able to awaken a waiting writer, allowing readers to again starve out writers, albeit with much less probability than would occur with a single condition variable.

Thread Scheduling

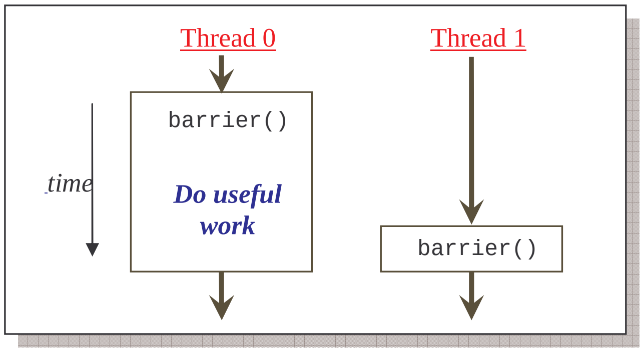
[This might be out of place—perhaps it belongs much earlier.]

POSIX Threads supports two scheduling scopes. Threads in system contention scope are called ***bound*** threads because they are bound to a particular processor, and they are scheduled by the operating system. By contrast, threads in process contentions scope are called ***unbound*** threads because they can execute on any of the Pthreads library's set of processors. These unbound threads are scheduled by the Pthreads library. For parallel computing, we typically use bound threads.

[Need a few more details: what is the default scope?
Are scheduling priorities an optional feature of POSIX Threads?
If not, talk here about scheduling attributes and priority inversion.]

Overlapping Synchronization with Computation

As we mentioned in Chapter 4, it is often useful to overlap long-latency operations with independent computation. For example, in Figure 9 Thread 0 reaches the barrier well before Thread 1, so would be profitable for Thread 0 to do some useful work rather than simply sit idle.



Formatted: Bullets and Numbering

Figure 9. It's often useful to do useful work while waiting for some long-latency operation to complete.

To take advantage of such opportunities, we often need to create ***split-phase operations***, which separate a synchronization operation into two phases: initiation and completion, as shown in Figure 10.

```
// Initiate synchronization
barrier.arrived();

// Do useful work

// Complete synchronization
barrier.wait();
```

Figure 10. Split-phase barrier allows a thread to do useful work while waiting for other threads to arrive at a barrier.

To see a concrete example of how split-phase operations can help, consider a 2D successive relaxation program, which is often used—often in 3D form—to solve systems of differential equations, such as the Navier-Stokes equations for fluid flow. This computation starts with an array of n+2 values: n interior values and 2 boundary values. At each iteration, it replaces each interior value with the average of its 2 neighbor values,

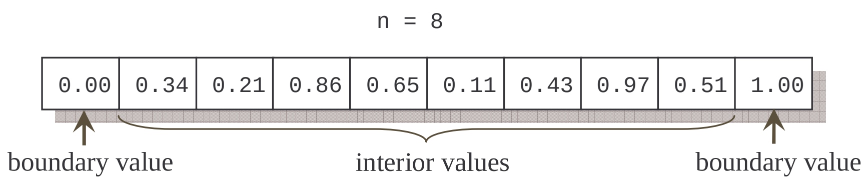


Figure 11. A 2D relaxation replaces, on each iteration, all interior values by the average their two nearest neighbors.

The code for computing a 2D relaxation with a single-phase barrier is shown below. Here, we assume that we have `t` threads, each of which is responsible for computing the relaxation of `n/t` values.

```
double *val, *new;                // Hold n values
int n;                            // Number of interior values
int t;                            // Number of threads
int iterations                    // Number of iterations to perform

thread_main(int index)
{
   int n_per_thread = n / t;
   int start = index * n_per_thread;

   for (int i=0; i<iterations, i++)
   {
      // Update values
      for (int j=start; j<start+n_per_thread; j++)
      {
         new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
      }

      swap(new, val);
      // Synchronize
      barrier();
   }
}
```

With a split-phase barrier, the main routine is changed as follows:

```
thread_main(int index)
{
   int n_per_thread = n / t;
   int start = index * n_per_thread;

   for (int i=0; i<iterations, i++)
   {
      // Update local boundary values
      int j = start;
      val[j] = (val[j-1] + val[j+1]) / 2.0;
16       j = start+n_pre_thread -1;
17       val[j] = (val[j-1] + val[j+1]) / 2.0;
18
19       // Start barrier
20       barrier.arrived();
21
22       // Update local interior values
23       for (j=start+1; j<start+n_per_thread-1; j++)
24       {
25          new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
26       }
27       swap(new, val);
28
29       // Complete barrier
30       barrier.wait();
31    }
32 }
```

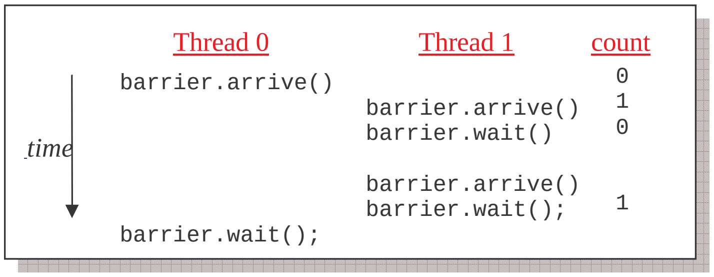
The code to implement the split-phase barrier seems straightforward enough. As shown below, we can implement a Barrier class that keeps a counter of the number of threads that should arrive at the barrier. To initiate the synchronization, each thread calls the `arrived()` routine, which increments the counter. The last thread to arrive at the barrier (line 30) then signals all waiters to wakeup and resume execution; the last thread also sets the counter to 0 in preparation for the next use of the barrier. To complete the synchronization, the `wait()` routine checks to see if the counter is non-zero, in which case it waits for the last thread to arrive. Of course, a lock is used to provide mutual exclusion, and a condition variable is used to provide synchronization.

Deleted: 31

```
 1 class Barrier
 2 {
 3    int nThreads;                  // Number of threads
 4    int count;                     // Number of threads participating
 5    pthread_mutex_t lock;

 6    pthread_cond_t all_here;
 7 public:
 8    Barrier(int t);
 9    ~Barrier(void);
10    void arrived(void);           // Initiate a barrier
11    int done(void);               // Check for completion
12    void wait(void);              // Wait for completion
13 }
14
15 int Barrier::done(void)
16 {
17    int rval;
18    pthread_mutex_lock(&lock);
19
20    rval = !count;                // Done if the count is zero
21
22    pthread_mutex_unlock(&unlock);
23    return rval;
24 }
25
```

```
void Barrier::arrived(void)
{
   pthread_mutex_lock(&lock);
   count++                      // Another thread has arrived

   // If last thread, then wake up any waiters
   if (count==nThreads)
   {
      count = 0;
      pthread_cond_broadcast (&all_here);
   }

   pthread_mutex_unlock(&lock);
}

void Barrier::wait(void)
{
   pthread_mutex_lock(&lock);

   // If not done, then wait
   if (count != 0)
   {
      pthread_cond_wait(&all_here, &lock);
   }

   pthread_mutex_lock(&lock);
}
```

Unfortunately, the code presented above does not work correctly! In particular, consider an execution with two threads and two iterations, as shown in Figure 12. Initially, the counter is 0, and Thread 0's arrival increments the value to 1. Thread 1's arrival increments the counter to 2, and because Thread 1 is the last thread to arrive at the barrier, it resets the counter to 0 and wakes up any waiting threads, of which there are none. The problem arises when Thread 1 gets ahead of Thread 0 and executes its next iteration—and hence its next calls to `arrive()` and `wait()`—before Thread 0 invokes `wait()` for its first iteration. In this case, Thread 1 will increment the counter to 1, and when Thread 0 arrives at the wait, it will wait block. At this point, Thread 0 is blocked waiting for the completion of the barrier in the first iteration, while Thread 1 is blocked waiting for the completion of the second iteration, resulting in deadlock. Of course, the first barrier has completed, but Thread 0 is unaware of this important fact.



```
barrier.arrive()                        0
                   barrier.arrive()     1
                   barrier.wait()       0

                   barrier.arrive()
                   barrier.wait();      1
barrier.wait();
```

Figure 12. Deadlock with our initial implementation of a split-phase barrier.

Formatted: Bullets and Numbering

Of course, we seem to have become quite unlucky to have Thread 0 execute so slowly relative to Thread 1, but because our barrier needs to work in all cases, we need to handle this race condition.

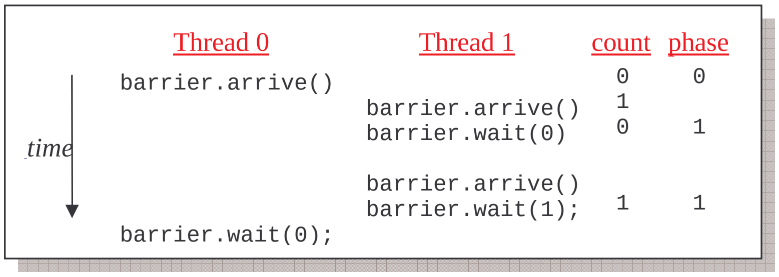
The problem in Figure 12 occurs because Thread 0 was looking at the state of the counter for the wrong invocation of the barrier. A solution then is to keep track of the current phase of the barrier. In particular, the `arrived()` method returns a phase number, which is then passed to the `done()` and `wait()` methods. The correct code is shown below.

```
class Barrier
{
   int nThreads;                 // Number of threads
   int count;                    // Number of threads participating
   int phase;                    // Phase # of this barrier
   pthread_mutex_t lock;
   pthread_cond_t all_here;
public:
   Barrier(int t);
   ~Barrier(void);
   void arrived(void);          // Initiate a barrier
   int done(int p);             // Check for completion of phase p
   void wait(int p);            // Wait for completion of phase p
}

int Barrier::done(int p)
{
   int rval;
   pthread_mutex_lock(&lock);

   rval = (phase != p)          // Done if the phase # has changed

   pthread_mutex_unlock(&unlock);
   return rval;
}

void Barrier::arrived(void)
{
   int p;
   pthread_mutex_lock(&lock);

   p = phase;                   // Get phase number
   count++                      // Another thread has arrived

   // If last thread, then wake up any waiters, go to next phase
   if (count==nThreads)
   {
      count = 0;
      pthread_cond_broadcast (&all_here);
      phase = 1 – phase;
   }

   pthread_mutex_unlock(&lock);
   return p;
43 }
44
45 void Barrier::wait(int p)
46 {
47    pthread_mutex_lock(&lock);
48
49    // If not done, then wait
50    while (p == phase)
51    {
52       pthread_cond_wait(&all_here, &lock);
53    }
54
55    pthread_mutex_lock(&lock);
56 }
```

Since the interface to the barrier routines have changed, we need to modify our relaxation code as shown below.

```
 6 thread_main(int index)
 7 {
 8    int n_per_thread = n / t;
 9    int start = index * n_per_thread;
10    int phase;
11
12    for (int i=0; i<iterations, i++)
13    {
14       // Update local boundary values
15       int j = start;
16       val[j] = (val[j-1] + val[j+1]) / 2.0;
17       j = start+n_pre_thread -1;
18       val[j] = (val[j-1] + val[j+1]) / 2.0;
19
20       // Start barrier
21       phase = barrier.arrived();
22
23       // Update local interior values
24       for (j=start+1; j<start+n_per_thread-1; j++)
25       {
26          new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
27       }
28       swap(new, val);
29
30       // Complete barrier
31       barrier.wait(phase);
32    }
33 }
```

With this new barrier implementation, the situation in Figure 12 no longer results in deadlock. As depicted in Figure 13, Thread 0's invocation of `wait(0)` explicitly waits for the completion of the first invocation of the barrier, so when it executes line 50 in the `wait()` routine, it falls out of the while loop and never calls `pthread_cond_wait().` Thus, deadlock is avoided.



```
       Thread 0              Thread 1              count  phase
       barrier.arrive()                              0      0
                             barrier.arrive()        1
  time                       barrier.wait(0)         0      1

                             barrier.arrive()
                             barrier.wait(1);        1      1
       barrier.wait(0);
```

Figure 13. Deadlock does not occur with our new split-phase barrier.

Formatted: Bullets and Numbering

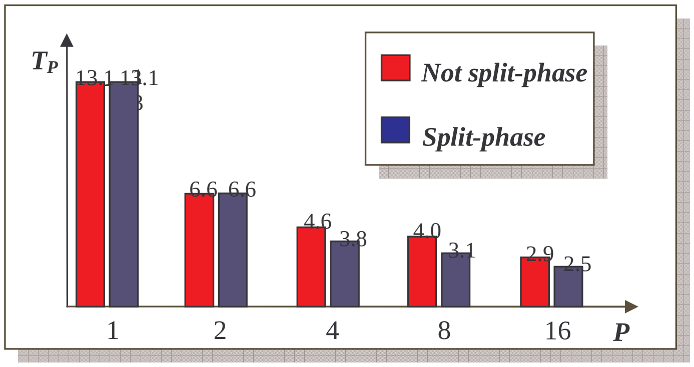


Figure 14. Performance benefit of split-phase barrier on a Sun E4000. n=10,000,000, 10 iterations.

Formatted: Bullets and Numbering

Java Threads

[Discussion of Java threads and a larger discussion of hiding concurrency inside of libraries.

- Nice model: explicit and convenient support for some common cases, but provides the freedom to use lower-level locks and condition variables where necessary. Can also hide concurrency inside of specific classes.
- Synchronized methods and synchronized classes
- Wait() and Notify()

Can we come up with examples where modular decisions about locking and synchronization are sub-optimal? In particular, we need examples where the context in which the data structure is used affects the synchronization policies.]

Critique

[What’s good about threads. What’s bad about threads.]

Shared virtual memory. Why can't threads-based programs execute on machines that do not support shared memory? Why can't we use software to provide a virtually shared address space on top of such machines? This question was heavily studied in the 1980's and 1990's. The basic issue is that the Shared Virtual Memory system needs to handle all data movement, and it is difficult to do this efficiently without knowledge of the application's sharing behavior. In particular, there is a tradeoff regarding the granularity of sharing: Large units of sharing can amortize inter-processor communication costs, at the expense of false sharing. Small units of sharing reduce false sharing but increase the overhead of moving data. In general, we'd ideally like the shared virtual memory system's granularity of sharing to match the application's logical granularity of sharing. Of course, even if the underlying shared virtual memory system were extremely efficient, there is still the question of whether threads-based programming is the right programming model.

Exercises

1. Our bounded buffer example uses a single mutex to protect both the `nonempty` and `nonfull` condition variables. Could we instead use one mutex for each condition variable? What are the tradeoffs?

   A: Yes, but this would not be a good tradeoff because both the producer and consumer access both condition variables, so both routines would have to acquire both locks instead of just one lock. Thus, there is added locking overhead but no greater concurrency.

2. The pthread_cond_wait() routine takes the address of the protecting mutex as a parameter so that the routine can atomically block the waiting thread and release the lock that is held by the waiting thread. Explain why these two operations must be performed atomically.

   A: If the two operations are not atomic, there are two cases: either (1) the thread is blocked first or (2) the lock is released first. In case (1), we have deadlock. In case (2), the code that blocks the waiting thread must first acquire the lock so that it knows that it is the only thread that is manipulating the queues associated with the condition variable, so the solution is possible but increases the latency of the operation. [Perhaps need to think about this answer some more.]

3.

Bibliographic Notes

The four necessary conditions for deadlock were first identified by Coffman, et al.

Hoare and Brinch Hansen proposed slightly different variations of monitors in the mid-seventies.

E.G. Coffman, M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, volume 3, pp. 67-78, June 1971.

C.A.R. Hoare, "Monitors, An Operating System Structuring Concept," *Communications of the ACM*, volume17, pp. 549-557, Oct 1974; Erratum in *Communications of the ACM*, volume 18, p. 95, Feb 1975.

P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, volume SE-1, pp. 199-207, June 1975.