Chapter 2

Software and Instruction Set Architecture

The primary function of a multiprocessor computer system to execute software as written by application programmers. The application programmer writes code in a high level language that embodies a higher level programming model. A compiler reduces the application software and associated libraries to machine code and operating system (OS) calls. Ultimately, all the software, in the form of machine code, is executed by hardware according to the instruction set architecture it is designed to implement. This is a classic use of layers of abstraction to manage complexity. Each abstraction layer specifies, through a well-defined interface, what should be done, not how it should be done. How it should be done is the implementation. When mapping application software to hardware there are three layers of abstraction, and three interfaces, that are of primary interest (see Figure 1). These are the Application Programming Interface (API), Application Binary Interface (ABI), and Instruction Set Architecture (ISA). Atop this stack of interfaces is a *programming model* which is a high level paradigm used for expressing an algorithm.

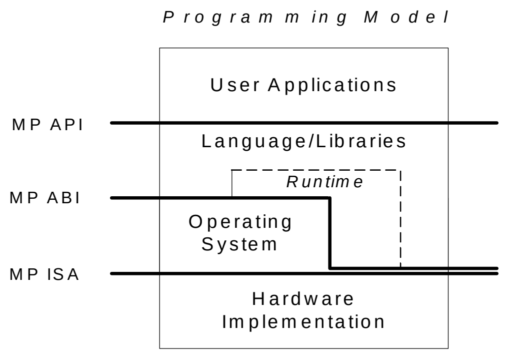


Figure 1. Three major multiprocessor interfaces are the API, ABI, and ISA.

The high level language programmer works at the API level and develops software using a programming language that is based on a programming model. For example, the Java API supports multithreaded applications using a shared memory programming model. Conversely, a given programming model may be implemented with a number of high level language APIs, all sharing common features. The API consists of the programming language, itself, plus libraries that may be used for implementing some or all of the multiprocessing capabilities. Strictly speaking the term "API" refers to the entire interface; in common usage, however, a particular library of routines is sometimes referred to as "an API"; we will occasionally use this terminology.

As part of many API implementations, *runtime software*, or simply "the runtime", provides certain management and service functions at the user level (i.e., with non-privileged instructions). These functions are often programming language-dependent, and performing them at the user-level saves the overhead of invoking the operating system. Consequently, the runtime not only executes API routines, but it may also manage some API-specific state information (e.g., variables and data structures). For example, the runtime may maintain message passing buffers on behalf of API routines that send messages between communicating threads.

The ABI consists of the operating system call interface and the user-level portion of the instruction set. The user-level instruction set consists of the non-privileged instructions that perform the basic load/store, branching, and ALU operations, for example. The operating system call interface is defined as part of the underlying operating system, Windows or Linux, for example.

The ISA consists of all the instructions, both privileged and non-privileged; the privileged instructions being used only the OS as part of its implementation. These privileged instructions often involve the management of hardware resources, real memory and I/O, for example. The ISA also contains aspects of the architecture other than the instructions themselves; for example, the virtual memory architecture and exceptions (traps and interrupts) are specified as part of an ISA. For multiprocessor systems, the ISA defines the sequences of values observed by processors when they execute load and store instructions that access shared regions of memory.

To summarize: a program written for an API is compiled into a binary program, that, when loaded into memory with API libraries, satisfies the specification of the ABI. The hardware implementing a specific ISA then executes the software (including both the program's binary and OS-implemented operations). In this chapter, we will describe all three of the major interfaces, as well as an overview of API and ABI implementations. The implementation of the ISA is covered in greater detail throughout the book.

2.1 Programming Models

The most widely-used programming model is the one implemented by a sequential procedural language such as C or FORTRAN. Object-oriented languages such as Java also fit this programming model. For throughput multiprocessing, where single-threaded programs run independently, this is the programming model being used. From the user's perspective this is just classic multiprogramming. For software that expresses concurrency, there are a number of parallel programming models, each of which has its own particular features. Two of the most common are adapted from the conventional procedural model, and these are ones that we will focus on. They are the shared memory and message passing programming models.

Before describing these parallel programming models, however, we should first define some terms more formally than we have done thus far. Over the years the terms "thread", "task", and "process" have meant different things depending on the operating system supporting them. The convergence toward Unix (including Linux) and Windows operating systems has also led to a convergence in terminology. "Task" now is usually used informally, while "process" and "thread" have specific meanings. We will define them here as they appear at the API level; later we will discuss them at lower levels of abstraction.

A *process* consists of a shared memory space and one or more execution threads (defined below) that can directly read and write values from/to the shared space. At the API level, memory may be arranged as a linear space containing individual variables and language supported structures, or memory may be organized as a heap that holds objects which are accessed via references (pointers) as in the case of Java or C#.

A *thread* consists of 1) a sequencer that steps through program statements and executes them, and 2) a private address space (in addition to the shared address space of the process to which it belongs). Hence, if a process consists of multiple threads, there are multiple sequencers and associated private address spaces. A private address space can only be accessed by statements executed by the thread with which the private space is associated. The private address space is sometimes arranged and accessed as a stack.

In the basic *shared memory* programming model, there is typically a single process with multiple threads. The threads communicate through the shared memory space. In the basic *message passing* programming model, there are multiple processes, each with a single thread. In a pure message passing model, each process has its own address space, with no shared memory accessible from the multiple processes, so the processes can communicate only by explicitly passing messages. In general, a program could employ a hybrid model that uses both message passing and shared memory and that supports multiple processes, each with multiple threads per process.

Programming models with explicit threads are so commonly used that it is easy to overlook the fact that there are other models. To provide some overall perspective, it is instructive to consider briefly a different type of programming model -- one that is functional rather than procedural. An example of such a model is embodied in spreadsheets which use a functional programming model. (In addition to being a functional programming model, spreadsheets also use a graphical interface, rather than a text-based interface as is common in many languages.) In a functional language, there is no explicit, procedural control flow as in languages such as C or Java. That is, there are no explicit threads or processes. Rather, the function is expressed as a series of equations that must simultaneously hold. Parallelism is implicit in the functional specification. The expressions in each spreadsheet cell can be evaluated in any consistent order (possibly involving re-evaluation of certain equations) as long as they stabilize to a set of numbers consistent with all the equations. Software implementing the spreadsheet may analyze the equations and determine a sequence of data dependencies, which can then be used to guide the order of evaluation. This evaluation can take place in parallel as long as the data dependencies are satisfied. Finally, it should be noted that computer hardware based on a functional model, *dataflow computers*, have also been proposed and experimental versions have been built, but thus far they have not been successfully commercialized.

As noted above, most parallel programming languages follow a procedural model rather than a functional model. The procedural model explicitly embodies control flow sequencing. The commonly used procedural programming model makes threads explicitly visible to the programmer so that the programmer can create and manage multiple threads. This may be done by constructs in the high level language or through libraries in the API, but regardless of how it is done, the key point is that the programmer has an awareness of the multiple threads, and develops software programs accordingly. In the following sections, we will discuss the characteristics of the two commonly used parallel programming models, with examples of the ways that APIs typically implement their primary features. The features of interest include the way that threads of control are represented and managed, are synchronized, and communicate with one another.

2.2 Shared Memory Programming

In this and the following section, shared memory programming will be described in a top down fashion: the API level is covered in this section, and its implementation at the ABI and ISA levels is in the next. Each of the major components – processes and threads, communication, and synchronization -- will be covered. In addition, example parallel programming patterns will be given. Throughout this section, examples will be drawn from two popular parallel programming APIs. The reader should refer to documentation for these two APIs for further details and examples.

- C and C++ are provided API level support for parallel programming via standard libraries that manage both process and threads. Because of the close historical relationship between C and the Unix operating system, many of the standard API level routines map directly to ABI level OS calls. The *pthreads* library [2]is a set of standard routines for managing threads.
- The Java API was originally developed around a multithreading model. The Thread class is contained in java.lang package, and is therefore central to the Java language.

2.2.1 Processes and Threads

The primary operations on processes and threads are creation, termination, suspension, and resumption of activity. Because a process is defined to have an address space, when a process is created, a new memory address space is created for it by the operating system. The process is given its initial program code, and execution begins at some specified initial program statement.

For example, the C library call `fork()` creates a duplicate of the process performing the call; the calling process is the parent and the forked process is the child. Because one would often want to run different code in the child than was running in the parent, a `fork()` is typically followed by an `exec()` call performed by the child, which loads in a new program for the child to execute. This fork/exec pairing is a feature (or quirk, depending on one's perspective) of Unix. The Windows OS performs what one would consider a more "natural" fork operation which both creates the new address space and loads in new software to execute. The C library routine `wait()` causes a parent to suspend and wait for a child to complete; when a process completes, it can terminate itself by calling the `exit()` routine.

With respect to threads, the pthreads `pthread_create` routine creates a new thread, running within the same memory space as the current, calling process. The routine `pthread_exit` terminates the thread. Figure 2 illustrates a Unix process with two threads. As shown in the figure, the new thread has its own stack within the user memory space. The stack is private to the thread. The private memory is not necessary for the shared memory paradigm, but it contains data that is local to the thread and typically holds intermediate results and other data that is specific to the thread.

A Java virtual machine runs as a single process which interprets or otherwise emulates a Java program. This single virtual machine process can support multiple Java threads, however. This is done through the `Threads` class; within the `Thread`s class, the method `start()` creates (initializes) a thread and the method `run()` contains the code that belongs to the thread. Consequently, the to create a new thread, the programmer instantiates a new `Thread` object, then calls the `start()` and `run()` methods for that object. When the `run()` method returns, the thread is terminated.

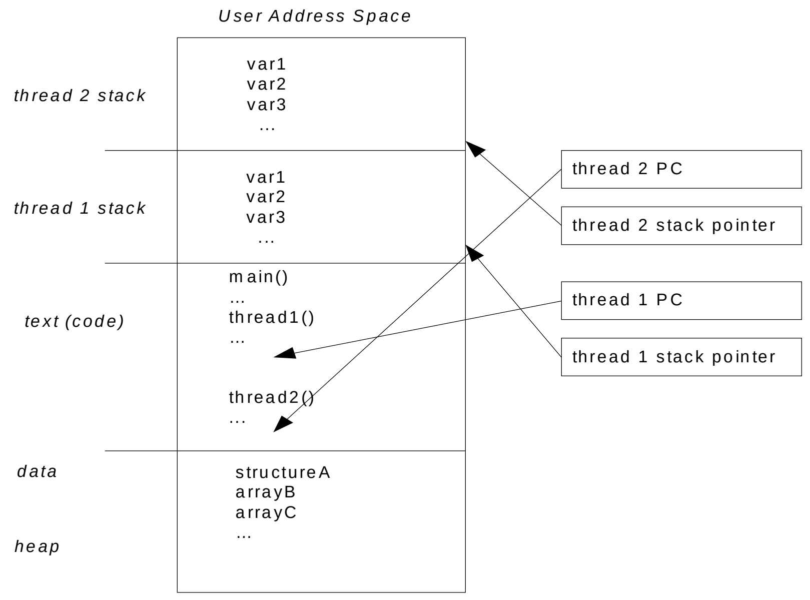


Figure 2. A Unix process with two threads.

2.2.2 Communication

An important aspect of a programming model is the way that threads communicate with one another. In fact, it is this aspect that gives the programming models their names: the shared memory model, and the message passing model. Although communication is a key feature of the shared memory model, it requires relatively little explanation.

At the API level, the shared memory programming model is illustrated in Figure 3. The architecture of the shared memory, itself, is determined by the high level language being used. It may be a flat homogeneous space where variables and structures reside (as in C), or it may be an object heap (as in Java). Regardless of its form, however, the all the threads of a process have access to it, and may read and write variables (or get and put object fields). It is through this shared memory space that values are communicated from one thread to another. On thread may write a value to a variable or field and another may read it; communication in the shared memory model is as simple as that (hence, one of the reasons the shared memory model is appealing).

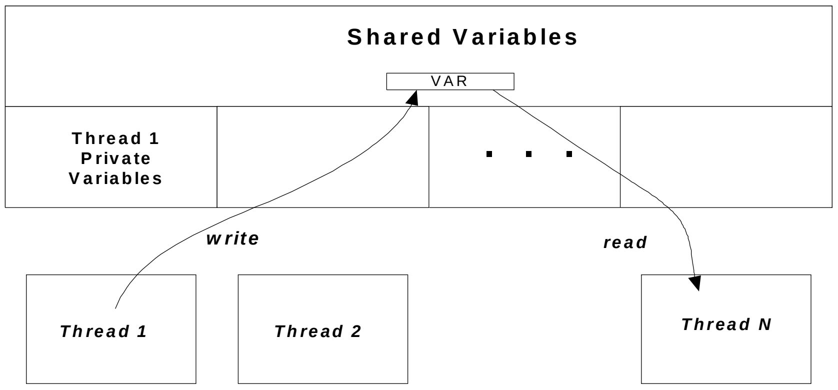


Figure 3. Shared Memory

2.2.3 Synchronization

Synchronization among communicating threads or processes is as important as data communication; if data communication is not properly synchronized, then the threads or processes can not reliably communicate. In the shared memory programming model synchronization is used in a number of ways. For example, synchronization may be used to let one thread know when communicated data is available, to prevent multiple threads from simultaneously updating the same data structure, or to wait for all threads to reach some point in their computation before proceeding.

In the shared memory programming model, synchronization is usually explicit, and is either built into the language or is supported via API library routines. The pthreads library contains support for a number of synchronization operations, for example. Java contains both language level and API library support.

In this section, we will focus on three primary forms of synchronization in the shared memory programming model: mutual exclusion, point-to-point synchronization, and rendezvous. In practice, these forms of synchronization are used as building-block elements of more complex *application design patterns*. An application design pattern is a general description or template for the solution to a commonly recurring software design problem. For example, it is common for two threads to have a producer/consumer relationship. One thread produces data that is consumed by another. There can be many types of producers, consumers, and data, but they all fit a common pattern, so the same synchronization structures can be used to implement all of them. Following subsections describe some of the important application design patterns, and the ways that conditions, locks, and rendezvous are employed to implement them.

MUTUAL EXCLUSION

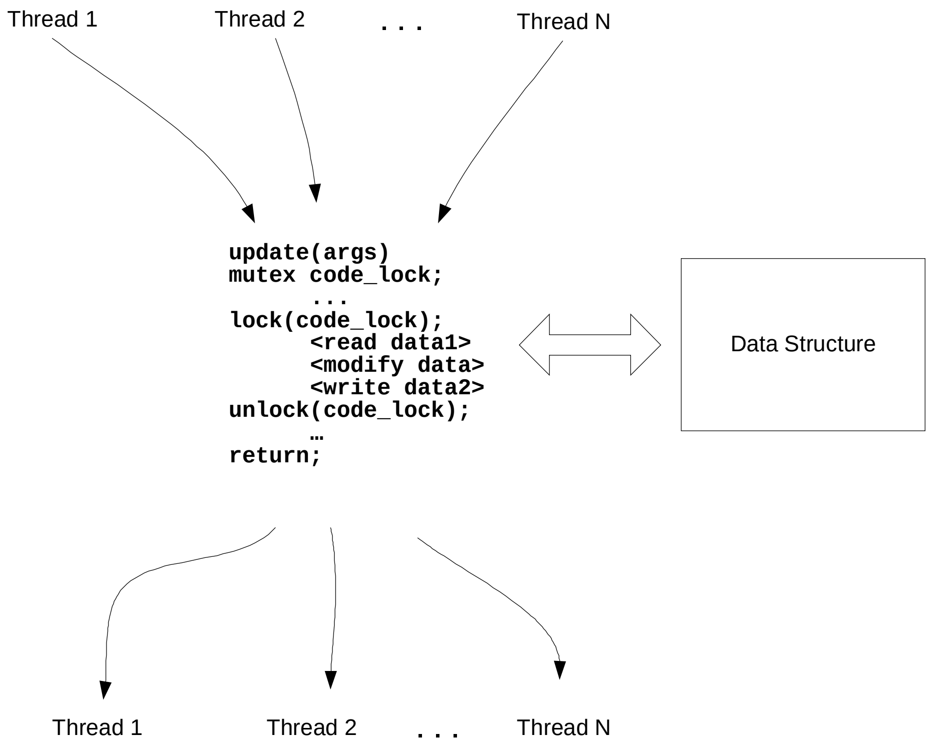
The first form of synchronization is *mutual exclusion*. The objective of mutual exclusion (*mutex*, for short) is to assure that only one thread at a time has access to a data structure or a piece of code. This is typically done via the abstraction of *locks*. A thread can set (or acquire) a lock, thereby excluding data or code access by other threads until the lock is cleared (or released).

The use of lock/unlock can be demonstrated via two commonly used programming patterns. Both of them implement synchronized updates of a shared data structure. Consider the situation where values held in a data structure are read, new data values are computed, and the data structure is updated with the new values. A read-modify-write code sequence of this type will require a number of machine instructions. A problem occurs if two or more different threads attempt to read and update the same data structure simultaneously; for example, a second thread may read from the data structure when it has been partially updated by the first, thereby reading inconsistent data values. Or, two threads may attempt to write simultaneously to the structure, again leading to inconsistent data. A solution is to force threads to "take turns" when accessing the shared data structure, with one performing its accesses (reads and/or writes) before the other is allowed to proceed with its reads and writes.

We consider two basic parallel programming patterns for the reliable sharing of data structures among multiple threads. The first is the *code locking* pattern. In this pattern, the data structure is always updated by a common set of code sequences, so that access to the data structure is controlled via access to the code sequences. The code sequences are often embodied as a set of procedures that are defined in conjunction with the shared data structure. This pattern is also called a *monitor* pattern. A similar technique is employed in object oriented programming where classes consist of objects and the methods that can access them.

The code locking pattern is illustrated in Figure 4. The code that updates a shared data structure is called `update`, and any thread wishing to update the structure must do so by calling `update`. Internal to `update` is a lock/unlock sequence that assures that at most one thread can be executing the read/modify/write update sequence at any given time. The variable `code_lock` is declared to be of type mutex, and it serves as a lock variable. If a thread calls `update` and the code is unlocked, then the execution of `lock(code_lock)` causes it to be locked, and the calling thread proceeds. When it is finished with the update, it unlocks the structure. On the other hand, if a thread calls `update` and another thread is currently executing `update`, then the second thread will block at the statement `lock(code_lock)` and wait until the first thread executes `unlock(code_lock)`.

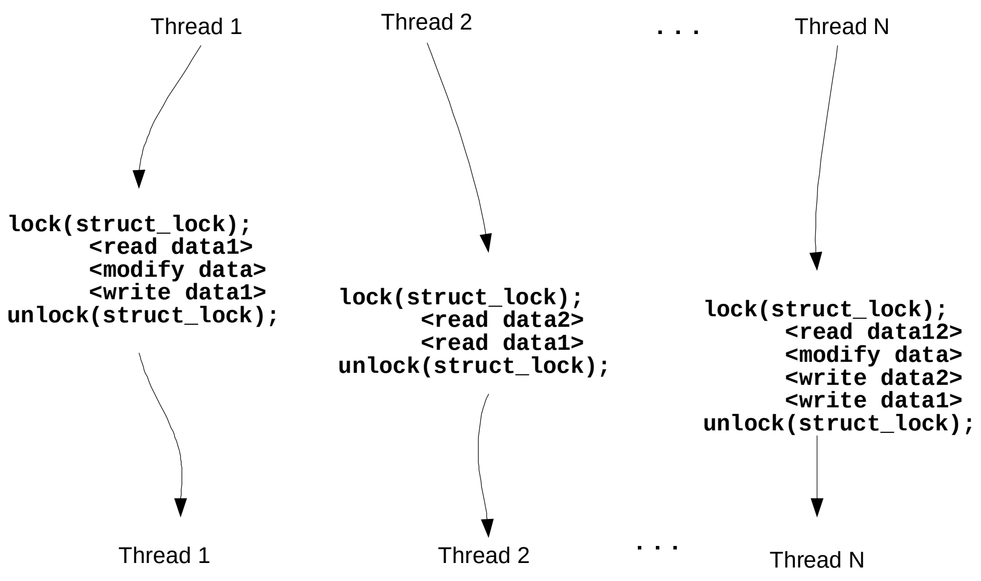
A code sequence of the type we have just described is an example of a *critical section*. It is a code sequence, consisting of a number of instructions that must be executed by only one thread at a time. That is, after a thread enters the critical section, no other thread should be allowed to enter until the first thread exits the critical section.



```
update(args)
mutex code_lock;
          ...
lock(code_lock);
      <read data1>
      <modify data>
      <write data2>
unlock(code_lock);
          …
return;
```

Figure 4. Code locking parallel programming pattern.

The second pattern for updating shared data is called *data locking*, and is illustrated in Figure 5. With the data locking pattern, a lock is associated with the data structure, rather than with the code that modifies it. In Figure 5, each of the threads updates the data structure in a different manner. Many code sequences can potentially update the structure, and a programmer can, more-or-less at will, develop new code sequences to access the structure, as long as the code sequence properly locks and unlocks the structure. The key point is that before any code sequence can access the structure, it must first lock the data structure (or wait if another code sequence has already locked it).



```
lock(struct_lock);
      <read data1>
      <modify data>
      <write data1>
unlock(struct_lock);
```

```
lock(struct_lock);
      <read data2>
      <read data1>
unlock(struct_lock);
```

```
lock(struct_lock);
      <read data12>
      <modify data>
      <write data2>
      <write data1>
unlock(struct_lock);
```

Figure 5. Data locking parallel programming pattern.

The data locking pattern is sometimes preferred if different combinations of data structures are accessed and modified collectively. For example, one code sequence may read data values from two data structures and update a third; another code sequence may access the third and use its values to update a fourth structure. For these types of operations, the programmer should first acquire the locks for all the structures involved in the collective operation, perform the operation, then release all the locks.

While the data locking pattern offers a lot of flexibility, it also exposes an important problem with multithreaded programming: *deadlock*. Deadlock can occur with many of the parallel programming patterns, but it is more apparent with the data locking pattern. With data locking, deadlock can occur when multiple locks must be acquired, but care is not taken with respect to the order of the lock acquisitions. This is illustrated in Figure 6. Here, two threads both attempt to acquire locks to struct1 and struct2. The first thread locks struct1 before struct2, and the second thread attempts to acquire them in the opposite order. Now, in practice, this may happen to work most of the time, but if the timing is such that both threads reach the locking code at the same time and the first thread locks struct1 while the second thread simultaneously locks struct2, then both threads will be blocked in their attempt to get their second lock. Neither thread can proceed; there is deadlock.

```
<Thread1>                  <Thread2>
Lock(lock_struct1)         Lock(lock_struct2)
Lock(lock_struct2)         Lock(lock_struct1)
<access struct1>           <access struct1>
<access struct2>           <access struct2>
Unlock(lock_struct2)       Unlock(lock_struct1)
Unlock(lock_struct1)       Unlock(lock_struct2)
```

Figure 6. Threads that may potentially deadlock.

Deadlock can be avoided in situations such as the one just described by following a convention of lock ordering amongst the various structures. In the example just given, a convention may be that all code must lock struct1 before locking struct2. For simple cases, this approach can be easily done. However, with more complex structures and sharing patterns (and programmer oversights), deadlock can, and does, occur in practice. There is no easy, universal solution to the deadlock problem; deadlock has become accepted as one of those issues that complicates the design of reliable multi-threaded code.

Another important issue when using mutual exclusion is the granularity at which locks are applied. By *granularity*, we mean the size of the data structure or code sequence that is protected with a single lock. Granularity affects overhead for locking, thereby affecting overall software performance. If locks are applied over larger regions (with fewer locks), then parallel programming tends to be simpler, however, threads may then be blocked by a lock when it isn't really necessary. As an extreme example, many early multi-threaded versions of Unix used what is called the "one big lock model" where a single lock surrounds all the code in the operating system kernel. This means that only one thread can be doing active work in the kernel at a time. Employing the one big lock model clearly has a negative effect on performance, as there are many cases where multiple threads in the kernel won't interact with each other at all. On the other hand, it is a very simple method for getting a multi-threaded operating system up and running.

As another example, consider a large array of data structures, for example banking records. In transaction processing software, one could implement data locking where there is one lock for the entire array of bank records. This would be simple, and might avoid some deadlock situations (because only a single lock must be acquired), but it would be unnecessarily inefficient. Most of the time, the bank records being updated are for different customers, and there would be no interference among threads performing simultaneous updates.

At the other extreme, one could provide every record with its own lock, and a thread would almost never have to block. There are some disadvantages to applying locks at a very fine granularity, however. In some cases, it may mean that many locks must be acquired and there is time overhead in acquiring the individual locks. Furthermore, there is memory overhead for maintaining lock variables. In general, the software developer should strike a balance between the extremes and strive for a granularity that is small enough that threads don't unnecessarily block for any significant fraction of time, but large enough that time and space aren't wasted.

With respect to some commonly used APIs, Java has built-in support for mutual exclusion [10]. Every Java object has a lock, and methods can be declared to be `synchronized`. If a synchronized method for a given object is called, then that object cannot be accessed by any other synchronized method until the first method is finished. This is essentially a form of data locking. Critical sections can be implemented in Java by defining a `synchronized` code block that incorporates an object whose sole purpose is to provide a lock. In the pthreads API, a lock variable of type `pthread_mutex_t` is first declared. Then the routine `pthread_mutex_lock` sets the lock variable and `pthread_mutex_unlock` unlocks it.

POINT TO POINT SYNCHRONIZATION

In *point to point synchronization* one thread signals another thread that some condition holds. The condition being signaled is often the availability of data. For example, in a producer/consumer programming pattern, a producer thread may place data into a shared buffer in memory, and then signal the (waiting) consumer thread that the data is ready to be consumed. This is illustrated in Figure 7. In this simple example, a data value is passed from a producer thread to a consumer thread through the variable `buffer`. The variable `full` serves as a synchronization variable and is used by the producer to signal the consumer that the buffer has been filled with a new data value. After reading the value, the consumer thread clears the `full` variable, thereby signaling the producer that it can pass another data

```
<Producer>                          <Consumer>
while (full ==1){} ; wait           while (full == 0){} ; wait
buffer = value;                     b = buffer;
full = 1;                           full = 0;
```

Figure 7. Example of produce/consumer pattern, implemented with ordinary variables.

In the example just given, an ordinary variable and memory operations are used for implementing the synchronization. However, most APIs provide special routines for this purpose. By using API routines, the synchronization variables and operations are implemented in lower abstraction layers, possibly all the way down to the ISA. Also, the API routine may be implemented in a fashion that is more efficient than when ordinary variables and operations are used. For example, the API, under runtime software control, may temporarily suspend a waiting thread and allow another (non-blocked) thread to run. This is described in more detail in the next subsection.

In the pthreads API, a mutex lock is associated with a condition that is to be signaled. The routine `pthread_cond_wait` causes a thread to suspend execution and wait for a condition to hold. The routine `pthread_cond_signal` when executed by a thread signals the condition to another thread (and if suspended waiting for the signal, the other thread will become active and continue execution).

RENDEZVOUS

A third important form of synchronization is the *rendezvous*, where two or more cooperating threads must all reach some point in the program before any of them can proceed. It is easiest to understand barriers by considering their application in the bulk synchronous programming pattern [11]. See Figure 8. In the bulk synchronous programming pattern multiple threads operate in a sequence of phases, where, in a given phase, multiple threads execute in parallel without synchronizing or communicating data values. Then, all threads wait until they have all completed the execution phase. At that point, they communicate data values, and perhaps a single thread does some additional serial computation. Then, the threads begin the next phase, and the process is repeated.

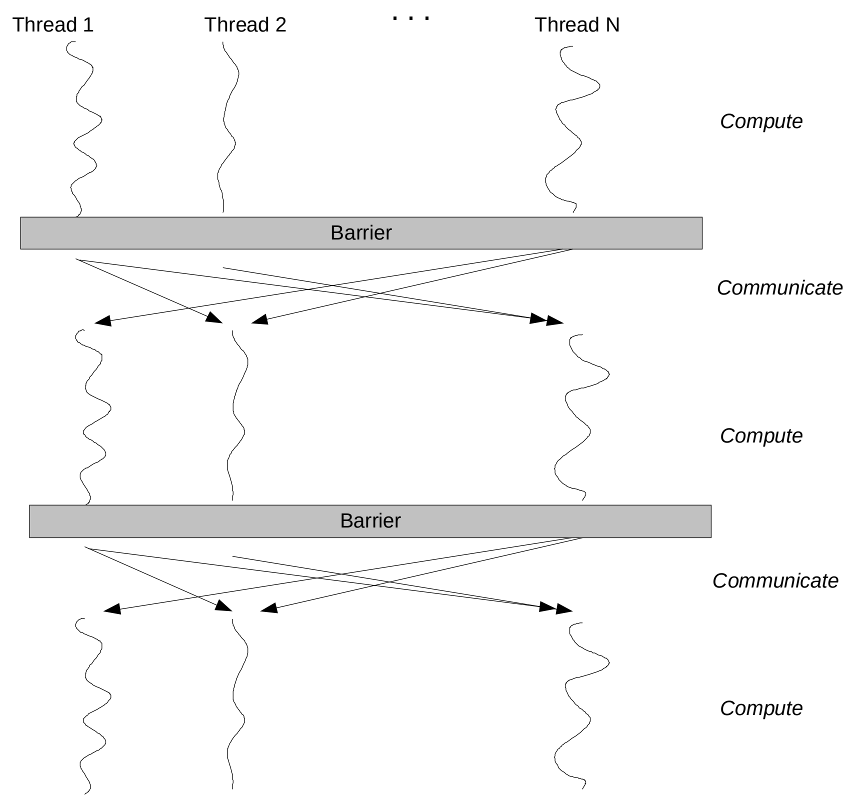


Figure 8. Bulk Synchronous programming pattern. There may also be a barrier immediately following the communication phase.

For example, multiple threads may be performing a complex database search. Each thread operates in parallel, searching an assigned portion of the database. Then, when all the threads are finished with their part of the search, they compare results, decide on a plan of action, then go back and do additional searching. Other examples come from numerical computing where iterative solver algorithms employ parallel threads which execute one computational iteration on arrays of data, then reach a barrier, check for convergence, then, if convergence hasn't occurred, they go back and perform another iteration.

2.3 Shared Memory Implementation

Now, we consider the way API level shared memory constructs are implemented at the ABI and ISA levels. This involves the definition of the ABI and the way an API is implemented in terms of the ABI. Keep in mind that characteristics of the API level are defined by a high level language and its libraries, while the ABI is defined by OS calls and user level instructions from the ISA.

2.3.1 Processes and Threads

Because a process has its own address space, and the OS controls memory mapping and protection, the OS must be involved in the creation of a new process. Consequently, creating a process at the API level is implemented with lower level code that performs one or more OS calls. In the case of C/Unix, the C `fork()`, `exec()`, and `wait()` library calls translate directly to equivalent OS calls.

Because there are often more processes than hardware processors, the OS schedules the processes by determining which ones get to run at any given time. It typically does this by maintaining a *run queue* for processes that are ready to execute. When a process requests I/O, or its time slice runs out, the OS chooses the next waiting process from the run queue (possibly taking process priorities into account) and starts the new process. And, because each process has its own address space, the OS must also change the address mapping (along with other process-related tables). The bottom line is that because processes have different address spaces, and the OS manages address space mapping, the OS must be involved with process management functions, including creation and scheduling.

Conceptually, one could support API level threads by associating each with an ABI level process, because most ABIs allow for the sharing of selected memory pages. This page sharing is set up by explicit OS calls. If this were done, however, then thread switches become equivalent (at the ABI level) to process switches. Because of the necessity to call the OS and for the OS to change page mappings for every process switch, this would lead to high overhead, or what is often referred to as a "heavy weight", thread switches.

By employing explicit ABI level threads, the weight of threads can be reduced; and, depending on the thread implementation the weight may be reduced significantly. The Unix OS supports a `pthread_create` system call, which is the counterpart of the pthreads API routine of the same name. The Linux OS supports a `clone()` system call which has similar functionality; so the `pthread_create` pthreads routine would be implemented with a `clone()`on a Linux system. Creating a thread does not require a new address space, and switching threads within the same address space does not require changes in the page mapping. Consequently, the OS can manage threads in a manner similar to the management of processes, except thread switches have become lighter because only the register state and stack pointer have to be changed at the time of a switch, not the entire address space.

Because address space mapping does not have to be changed on a thread switch, however, it is not even necessary to involve the OS at all. That is, the implementation of API threads can be done at the user level via the runtime software (Figure 1). In particular, the runtime can create threads by simply adding another stack within a process's existing address space. The runtime can maintain its own run queue, so that threads can be scheduled at the user level. For example, when a thread completes its task, it can jump to the runtime scheduler, pick up a new task from the run queue and then jump to the start location of the new task (essentially becoming a new thread). In this manner a single OS thread can be time-shared among multiple user level threads.

We have just described two types of threads: *kernel threads*, which the OS schedules, and *user threads*, which the user level runtime manages. To perform parallel processing, a program must create a certain number of kernel threads, because these are the ones that are assigned to hardware processors (program counters) by the OS. However, once the kernel threads are created, multiple user level threads can then be mapped to the kernel threads in whatever manner the runtime chooses. The relationship between kernel threads and user threads is illustrated in Figure 9. In the figure, the OS scheduler assigns kernel threads to processors. Then, additional scheduling of user level threads onto the kernel threads can be performed by user level runtime software associated with an API implementation.

At the ISA level no special support is required for processes and threads, beyond that already present for single threads. The management of threads and processes amounts to the assignment of the program counter of a processor to a location with a given thread or process. This is done by the OS, and all that is required is that the OS always maintain control over which thread/process is running at any given time.

As multiple processors are booted, operating system code is initially given control to each of the program counters. Then, just as with a uniprocessor, the OS always controls the software that has access to the program counter. When control is passed to user level code, the OS sets a timer which forces the control to be passed back to the OS after some maximum interval. If the user code performs a system call, is interrupted, or traps, it also relinquishes control jumping to an ISA-specified "vector" location in the OS.

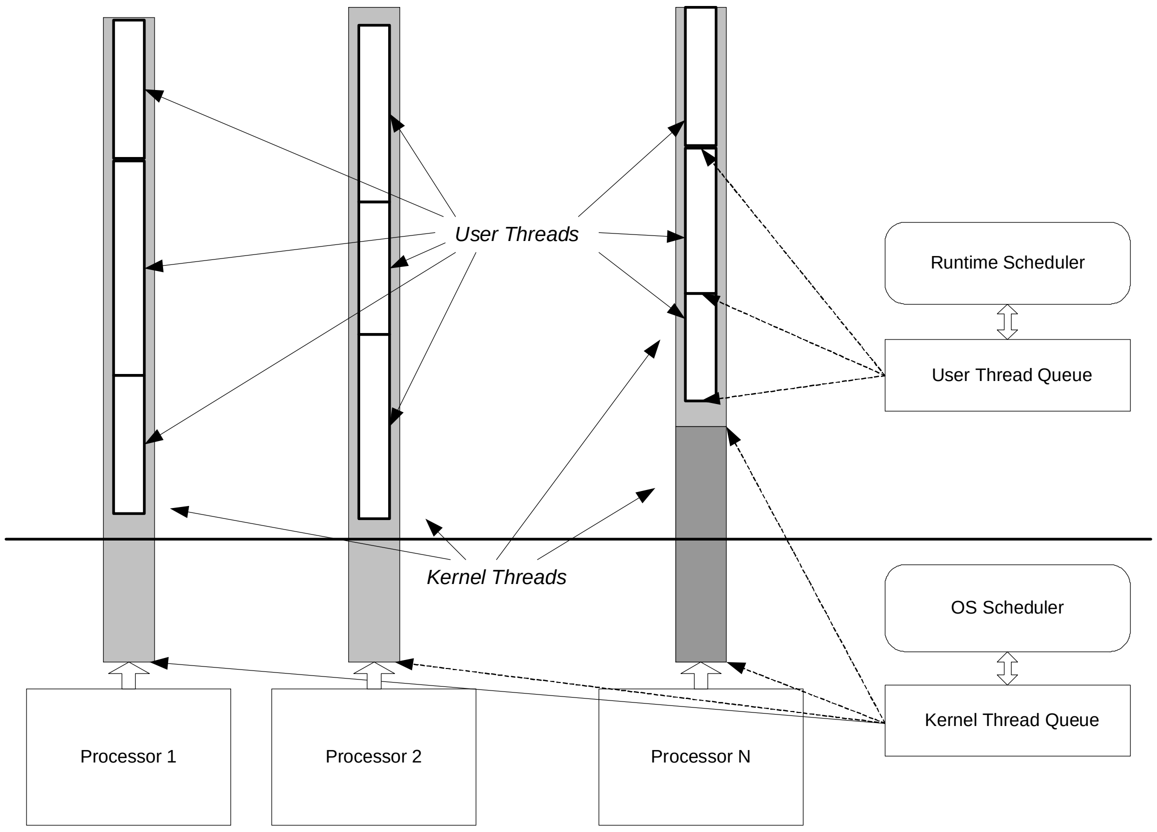


Figure 9. Kernel threads are scheduled onto hardware processors by the operating system; user threads are scheduled onto the kernel threads by a use level runtime scheduler.

2.3.2 Communication

A shared memory programming model is typically implemented on a hardware platform that supports shared memory directly in the hardware; that is, ordinary machine level load and store instructions can be used for reading and writing shared variables (or fields in objects). There have been exceptions to this, where a shared memory programming model has been implemented on a distributed memory hardware platform, but these exceptions have primarily been research projects. Generally speaking, implementing the shared memory programming model on distributed memory hardware poses a number of difficult performance problems and such implementations have not entered mainstream.

In the conventional shared memory implementation, data communication at the ABI and ISA levels is just a matter of mapping the API high level language reads and writes (or gets and puts) onto ordinary machine level load and store instructions. That is, no special instructions are required in the ISA for merely communicating data among threads. However, because shared memory can be accessed from multiple, concurrently executing threads of execution, additional semantics regarding the ordering of loads and stores from different threads must be specified as part of the ISA; and this is not as easy as it might first appear. In fact, memory ordering semantics are important enough that they will be discussed in a subsection of their own (Section 2.3.4), after we have discussed synchronization in the next subsection.

2.3.3 Synchronization

It is this aspect of the shared memory model that has the biggest impact on the ISA. Implementing synchronization ordinarily does not involve OS intervention (and costly system calls). Thread synchronization is typically implemented at the user level, using a small number of machine level instructions, sometimes with API runtime support. However, unlike simple data communication which uses ordinary load and store instructions, special instructions are often added to the ISA for of implementing synchronization primitives. The reason can best be understood by considering an example.

Figure 10 illustrates a naive ISA level implementation of a critical section where ordinary loads and stores are used for locking and unlocking. At first glance, this example may appear to work properly: only one thread at a time can be in the critical section. In fact, this is not a reliable implementation of a critical section. The reason is that it may happen that the timing is such the two threads perform the load of the lock (at LAB1) at the same time. They both will read a 0, will both set the lock to one, and both will enter and execute the code in critical section at the same time.

There are other, more clever implementations of a critical section that use ordinary loads and stores; however, they also involve longer code sequences, are more difficult to understand, and can lead to bugs if not properly implemented. Consequently, most ISAs include special instructions to simplify mutual exclusion and other synchronization operations. These instructions allow the indivisible, or *atomic*, reading and writing of a memory variable by the same instruction. An example will illustrate the utility of such instructions.

```
      <thread 1>                  <thread 2>
          .                           .
          .                           .

LAB1: Load R1, Lock           LAB2: Load R1, Lock
      Branch LAB1 if R1==1          Branch LAB2 if R1==1
      Enter R1, 1                   Enter R1,1
      Store Lock, R1                Store Lock, R1
          .                             .
      <critical section>            <critical section>
          .                             .
      Enter R1, 0                   Enter R1, 0
      Store  Lock, R1               Store Lock, R1
```

Figure 10. A naive implementation of a critical section using ordinary loads and stores for mutual exclusion.

In Figure 11, mutual exclusion is implemented with a Test&Set instruction. This one instruction both reads the value of variable `lock` and writes a one to `lock`. Between the read and write no other instruction has access to the `lock` memory location. That is the, the read and the write are indivisible, or *atomic*. Then, whichever thread gets to the locking instruction first will set the lock, and the other thread will see a lock that is already set. Also, note that the instruction we use looks more like a "Read&Set" than a "Test&Set". For the Test&Set used in some ISAs, the read operation also tests the value for zero and sets a condition code register accordingly. Then, the following conditional branch checks the condition codes. Because the pseudo-ISA used in our examples doesn't use condition codes, the lock variable is only read, and is tested by the subsequent conditional branch instruction. Finally, in the example, the lock is cleared with a reset instruction, although it could have been done with an ordinary store as in Figure 10. The reason is that some hardware implementations may be better optimized if it is known that a synchronization instruction is being executed rather than an ordinary store (see Section 2.3.4).

```
      <thread 1>                        <thread 2>
          .                                 .
          .                                 .
          .                                 .

LAB1: Test&Set R1, Lock           LAB2: Test&Set, Lock
      Branch LAB1 if R1==1              Branch LAB2 if R1==1
          .                             .
      <critical section>                <critical section>
          .                             .
      Reset Lock                        Reset Lock
```

Figure 11. Implementation of a critical section with an atomic Test&Set instruction.

In practice, there are a number of instructions that can be used to achieve the same effect as the Test&Set; the one thing they have in common is some form of atomic read/write to a memory location. And, in general, given one such instruction, the functional equivalent of the others can be easily encoded. Two such examples are the Fetch&Add and Swap instructions. The semantics of these instructions, along with the Test&Set are illustrated in Figure 12. The Fetch&Add loads a value from memory, adds a value to it and stores the result back. The Fetch&Increment instruction (not shown in the figure) is a special case of Fetch&Add where the value added is always a 1. A Swap instruction simply swaps a value in a register with a value in memory. The Test&Set we have been using is equivalent to a Swap where the initial register value is always one. Figure 12b illustrates the implementation of a Fetch&Add with a Test&Set; the Test&Set is used as a lock for a critical section that performs the fetch and add using conventional instructions.

```
Test&Set(reg,lock)         Fetch&Add(reg,value,sum)          Swap(reg,opnd)
 reg ← mem(lock);          reg ← mem(sum);                   temp ← mem(opnd);
mem(lock) ← 1;             mem(sum) ← mem(sum)+ value;       mem(opnd) ← reg;
                                                             reg ← temp
                                     (a)
```

```
try:    Test&Set(lock);
        if lock == 1 go to try;
        reg ←mem(sum);
        mem(sum) ← reg+value;
        reset (lock);
        (b)
```

Figure 12. a) The semantics of three synchronization instructions; memory reads and writes are indivisible b) the Fetch&Add operation implemented with a Test&Set.

Some RISC instruction sets have divided the read and write components of a test and set into separate, specialized load and store instructions, but the semantics of the special instructions are defined such that they can be combined to achieve the same effect as an atomic read/write. The MIPS instruction set includes a pair of load/store instructions of this type.

In the MIPS instruction set, the `load_linked` instruction performs a conventional memory load, but, in addition, it records the memory address in a special register and sets a flag indicating, in normal usage, that there is a pending `store_conditional` instruction to the same address. Then, any operation performed by any of the processors that might cause atomicity to be violated clear the flag. Exactly which operations clear the flag depend on the implementation. But a common situation occurs if another processor performs a store to the memory address held in the special register. The flag is also cleared when the process performing the `load_linked` instruction is context switched, because the special address register is not saved on a context switch. A `store_conditional` instruction fails (does not write to memory) if the flag is cleared at the time it is executed. As defined in the MIPS instruction set, the `store_conditional` instruction also sets a condition code corresponding to whether the flag was set or clear. This allows the success of the store instruction to be tested via a conditional branch instruction.

Figure 13 illustrates the use of the load_linked store_conditional pair; the code sequence in the figure implements an atomic swap operation between the values in register r4 and the memory location addressed by register r1. The first four instructions form a loop that tries the load_locked and store_conditional until the store succeeds. Then, after it succeeds, the value loaded from memory is copied into register r4. In most real cases, the store_conditional will succeed on the first try.

The separation of the load and store components is in line with the RISC philosophy of separating complex instructions into simple ones. In a bus-based multiprocessor, this means there is no need for a special bus operation that performs an indivisible read/write pair to a memory location; normal, separate bus cycles can be used. However, as with some other early RISC features, such as delayed branches, the separation of the load and store operations probably seemed like a good idea at the time. In retrospect, this feature is closely tied to a specific implementation (in this case, a shared bus). In a non-bus system, it would probably be simpler to implement an indivisible read/write as a single operation.

```
        Swap(r4, mem(r1))
try:    mov     r3,r4           ;move exchange value
        ll      r2,0(r1)        ;load locked
        sc      r3,0(r1)        ;store conditional
        beqz    r3,try          ;if store fails
        mov     r4,r2           ;load value to r4
```

Figure 13. Implementing an atomic Swap with the MIPS load_linked and store_conditional instructions.

LOCK EFFICIENCY

An important issue when implementing locks is efficiency, and there is a number of aspects to efficiency. One of them was addressed earlier in Section 2.2.3 with respect to lock granularity. Recall that granularity is the size of the code or data region that is protected by a single lock. Finer granularity means more locks, but it reduces the probability of needless thread blocking. Lock granularity is an efficiency feature that is visible to the HLL programmer, and the HLL programmer is therefore responsible for determining the granularity of locks, and, indirectly, the number of times that threads are blocked.

There are other lock efficiency issues that are determined by the implementation of the API. Consider the critical section as implemented in Figure 11. This is an example of a *spin lock,* so called because of the small loop that repeatedly tests ("spins" on) the lock until it is found to be clear. Such a spin lock, while effective, can lead to a significant waste of resources. First, it repeatedly hammers on the memory system, consuming bandwidth. If a number of threads are simultaneously spinning on the same lock, the aggregate memory bandwidth demand can be huge. In some cases, adjusting lock granularity can at least partially alleviate this problem. A second problem is that the processor doing the spinning is not performing what one would consider useful computation; it is simply performing the same operation over and over.

A partial solution to the first problem is based on the observation that in multiprocessor systems that employ cache memories, a repeated series of read operations to the same address only access the local cache – unless some other processor writes to the address. On the other hand, a write operation involves notifying, one way or another, all the other caches in the system of the modification to memory. Hence, the Test&Set as we have defined it consumes more memory bandwidth resources than a simple read (or Test). The reasons for this will be more apparent when we discuss cache coherence techniques in Chapter 4, however, the summary just given will suffice for now. In any case, the above observation regarding the relative bandwidth requirements of Test&Set and ordinary loads, leads to a more efficient spin lock implementation referred to as Test&Test&Set (see Figure 14). In Test&Test&Set, there is a pair of loops. First, there is a loop that spins on the lock performing only a read. When it finds that the lock is clear, it attempts to acquire it via a Test&Set. If this fails (because some other thread acquired it first), then it goes back and spins some more.

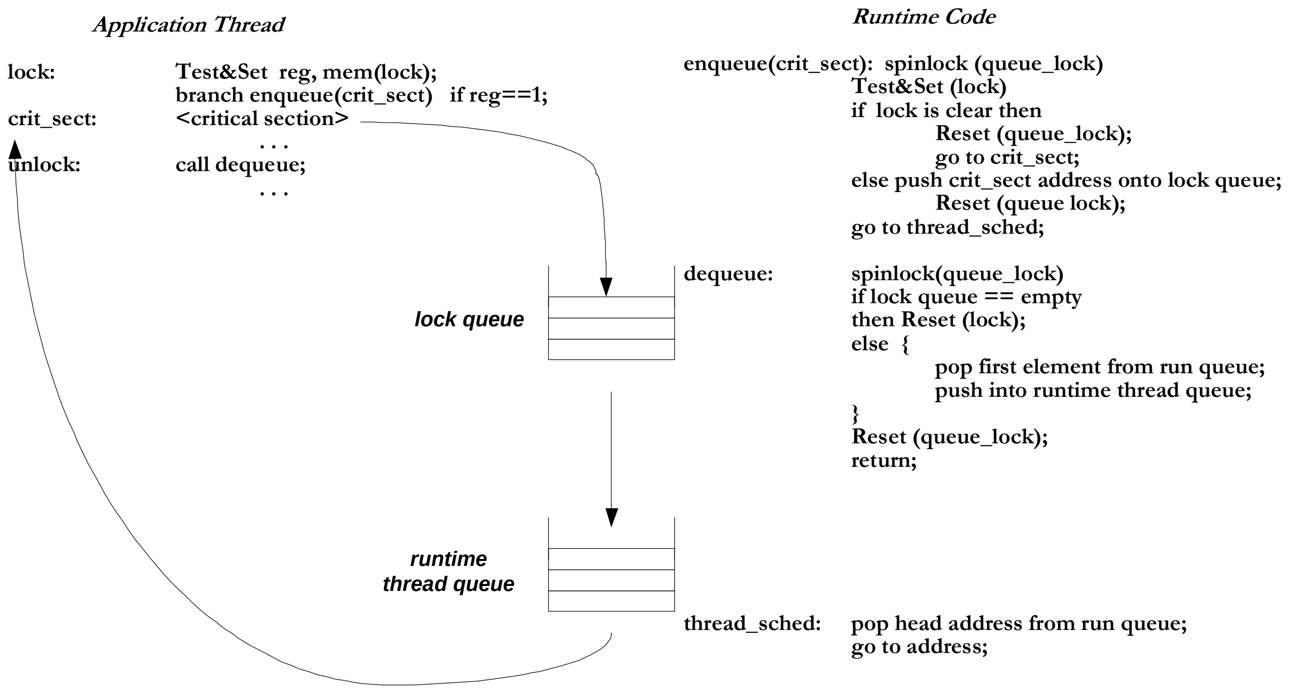
Test&Test&Set(reg, lock)

```
test_it:    load        reg, mem(lock)
            branch      test_it if reg==1
lockit:     Test&Set    reg, mem(lock)
            branch      test_it  if reg==1
```

Figure 14. Implementation of Test&Test&Set spin lock.

Other solutions for reducing spin overhead have been proposed. For example, one technique is to "back off" and wait for a period of time after lock acquisition fails before trying again. The more failures, the longer the back off time. This technique is similar to the method used in Ethernet for network access.

A solution to both types of wasted resources (memory bandwidth and processor cycles) can be implemented in the runtime. The basic idea is, when a thread fails to acquire a lock, it is placed on a runtime managed queue to wait for the lock to become available and there is a switch to a different (unblocked) thread. When a thread finishes with lock, it branches to the runtime where any threads in the lock's queue can be re-awakened, given the lock, and placed into the runtime's run queue. This is illustrated in Figure 15. This approach also allows the runtime to enforce some higher level discipline/priorities regarding the use of a critical section. The response time of a queued lock is somewhat slower than a spin lock, but it may be more efficient overall.



```
lock:       Test&Set  reg, mem(lock);
            branch enqueue(crit_sect)   if reg==1;
crit_sect:  <critical section>
                    . . .
unlock:     call dequeue;
                    . . .
```

```
enqueue(crit_sect):  spinlock (queue_lock)
              Test&Set (lock)
              if  lock is clear then
                        Reset (queue_lock);
                        go to crit_sect;
              else push crit_sect address onto lock queue;
                        Reset (queue lock);
              go to thread_sched;

dequeue:      spinlock(queue_lock)
              if lock queue == empty
              then Reset (lock);
              else  {
                        pop first element from run queue;
                        push into runtime thread queue;
              }
              Reset (queue_lock);
              return;

thread_sched: pop head address from run queue;
              go to address;
```

Figure 15. A queueing lock. If a thread finds a lock busy, it is enqueued to wait until the lock is released. In the meantime, another user-level thread can run.

BARRIERS

In pthreads, barrier synchronization can be implemented with the `pthread_join` routine; a thread executing this operation suspends and waits for all the other threads to execute a `pthread_join` before they are all allowed to proceed. Barrier synchronization can also be implemented with a counter variable and a critical section that each thread uses to update and check the count. This illustrated in Figure 16. In this example, a barrier is a structure that consists of a lock, a flag, and a count. The count is initialized to zero. Each thread calls the barrier code with arguments that indicate the name of the barrier and the number of threads that are taking part in the barrier synchronization (in this example there are *n* threads). Each thread first sets the barrier's lock and increments the counter; the very first thread also clears the flag (which may still be set from the previous use of the barrier).

Then, the thread checks to see if the count was equal to *n*, the total number of threads. If not, (the else case) the thread waits for the flag to be set. If so, it clears the count (to initialize it for the next usage of the barrier) and sets the flag to one, notifying all the other waiting threads that the barrier has now been satisfied, and they can go ahead and continue computing.

```
Barrier (bar_name, n) {
        Lock (bar_name.lock);
        if (bar_name.counter = 0)  bar_name.flag = 0;
        mycount = bar_name.counter++;

        Unlock (bar_name.lock);
        if (mycount == n) {
                bar_name.counter = 0;
                bar_name.flag = 1;
        }
        else  while(bar_name.flag = 0) {};   /* busy wait */
}
```

Figure 16. Example of barrier synchronization code.

If there are a large number of threads waiting to synchronize at the barrier and they arrive at about the same time, the critical section code can be a bottleneck. To relieve this bottleneck, a barrier can be implemented as a hierarchy of smaller barriers. Each of the barriers has a counter and lock, but only the barrier at the very top of the hierarchy has a flag. This is illustrated in Figure 17. Here, each thread is associated with one of the lower level barriers. The thread first checks the counter at the lowest level, if it is the last thread to reach the lower level barrier, then it moves up one level and checks the counter at that level. When it reaches a level where it is not the last, then it begins waiting for the flag (which is set at the highest level). The very last thread to reach the barrier goes all the way to the top barrier, and sets the flag, thereby releasing all the threads. By providing multiple barriers in a hierarchy, threads can "check-in" at low level barriers in parallel, and the bottleneck is relieved.

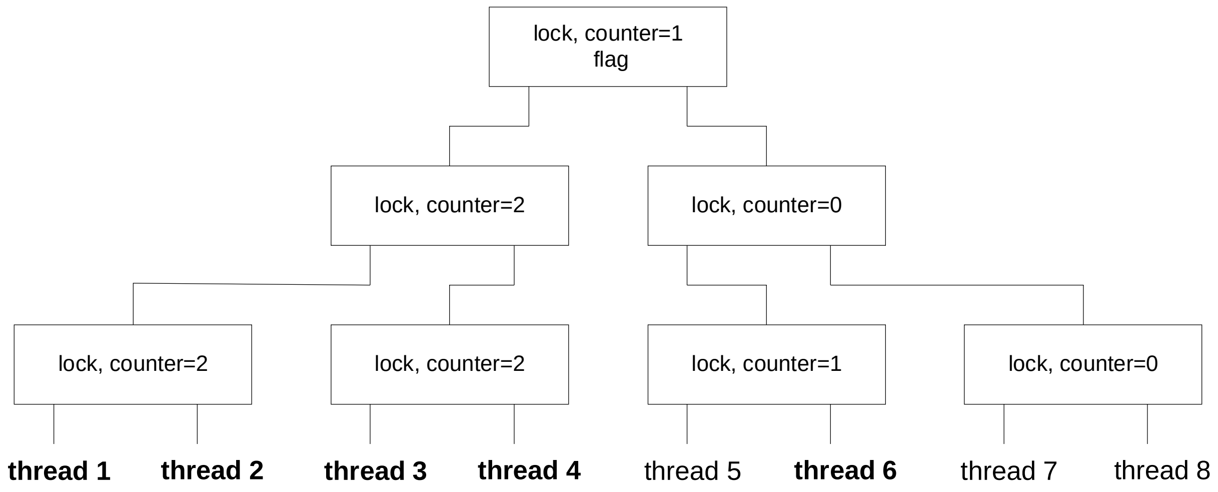


Figure 17. Barrier implemented as a hierarchy. Threads in bold have already reached the barrier. The individual barriers in this example service only two threads or lower level barriers; in a real implementation, the fan-in would probably be higher.

In some research and special purpose parallel computer systems, ISA level support for barriers has been provided. For example, the Cray T3D had a special hardware barrier network for implementing barriers. In most mainstream applications, however, barriers do not occur frequently enough, and there aren't enough threads to justify special purpose hardware for barriers.

2.3.4 Memory Consistency and Coherence

Now, we return to the topic of shared memory communication and deal with what is probably the most difficult issue when it comes to a shared memory implementation – correct sequencing of load and store instructions performed by multiple threads.

In a uniprocessor program, there is a single thread, and the program counter defines the sequence with which instructions are to be executed. Naturally, the programmer expects instructions to modify registers and memory in exactly the order specified by the sequencing of the program counter. This is what is referred to as *program order*. Program order is part of the ISA specification (although it is so basic to the understanding of how computers work that it is often not explicitly stated). And, although program order is part of the architected specification, an actual hardware implementation does not literally have to execute instructions in the same order; many superscalar processors do not, as a matter of fact (see Chapter 3). What is important is that the software *behaves* exactly as (gives the same answers as) it would if the hardware did execute instructions in program order.

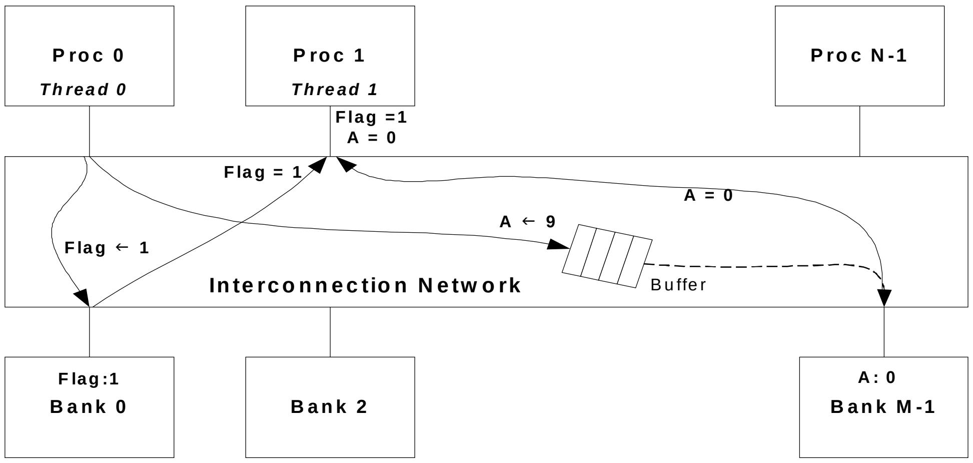
SEQUENTIAL CONSISTENCY

With the shared memory programming model, the multiple threads of control modify the same shared memory, and each one of them operates according to its own program order. Because memory is shared, one thread can read a value from memory that another thread has written. This leads to the need for an ISA specification regarding the observed ordering of memory events among the concurrently executing threads.

At first, this seems as basic as program order is with single threads. But let's consider an example. Figure 18a is a section of code that is part of a simple producer/consumer programming pattern. Initially, variables *A* and *Flag* are 0. Thread 0 produces a value of 9, places it into variable *A* and sets *Flag* to 1, indicating that the value in *A* is ready to be consumed. Thread 1 repeatedly tests the flag for 0, and, eventually, when *Flag* becomes 1, Thread 1 proceeds by reading the value of *A*, and compares it with 0. It seems obvious that *A* should not be 0 when Thread 1 tests it and the test should fail. That is, the multi-threaded program seems to suggest that *A* is set to 9 before *Flag* is set to 1, and when Thread 1 tests and finds *Flag* to be 1, the value of *A* must be a 9 at that point.

```
Thread0:    A=0;                     Thread1:
            Flag = 0;                          ....
            ....                               ....
            A=9;                               While (Flag==0){};
            Flag = 1;                     L2:  if (A==0)...
```

(a)



(b)

Figure 18. a) Multi-threaded Producer/Consumer code b) Implementation on a system with buffering in an interconnection network. The assignment of 9 to A gets delayed and his held in a buffer internal to the network. Meanwhile, the other memory accesses proceed smoothly. Consequently, Thread 1 reads an unexpected result for variable A.

However, in real hardware implementations, unless care is taken, the "impossible" condition of *A*==0 may be true when Thread 1 reaches L2. For example, consider an implementation as shown Figure 18b. This hardware implementation employs an interleaved memory with multiple memory banks and an interconnection network containing buffering at intermediate stages, in case there is contention for links in the network. Because of memory addressing patterns elsewhere in the system (from other active processors in the system), some paths from certain processors to certain memory banks may be more or less congested than others. This can affect the delay getting to and from memory banks. In this particular example, variables *A* and *Flag* are in different memory banks, and the path from processor 0 (which is running Thread 0) to *A* contains some congested links and the update to *A* gets hung up in a buffer. Meanwhile the path from Processor 0 to Flag is not congested (and fast). The paths from Processor 1 (running Thread 1) to both *A* and *Flag* are also fast. Under these conditions, when Thread 1 reads the updated *Flag* value, it immediately reads the value of *A*, and gets the old value of 0, not the new value of 9.

This suggests that implementing program order in individual processors and then combining them arbitrarily into a multiprocessor is not sufficient to yield the expected, or even reliable, results when a multi-threaded application is executed. Furthermore, the example just given contains only processors and memories; if cache hierarchies are added to the mix, the problem becomes much more difficult because the caches are, in a sense, very large buffers that can hold data values for a very long time before they reach memory.

The first step in dealing with memory ordering among multiple threads is to define the semantics of memory ordering in a simple way. To provide one such a specification, consider a straightforward memory ordering model (see Figure 19). The figure shows multiple processors, each performing loads and stores to shared memory in program order. The memory system selects and acts upon the memory requests in some order (the exact order is not important, any order can be used). The key point is that memory receives and *appears to* act upon the memory requests one at a time. Note that it is not necessary that only one access literally happens at a time (e.g., during a given clock cycle); in fact some of the accesses can take place at the same time, as long as the data values loaded and stored are the same as if the selector only chooses one at a time to act upon.

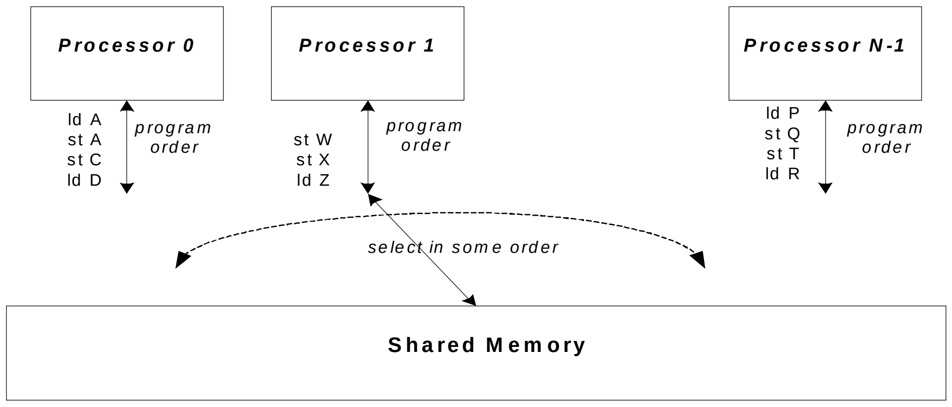


Figure 19. Model for sequential consistency. Processors perform loads and stores in program order, and memory systems services requests in some (arbitrary) order.

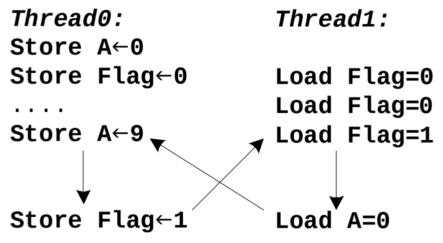
The need for a memory ordering model was recognized at least as early as 1979 by Leslie Lamport [12], who identified the model shown in Figure 19 as "sequential consistency" and defined it as:

"A system is sequentially consistent *if the result of any execution is the same as if the operations of all processors were executed in some sequential order and the operations of each individual processor appears in this sequence in the order specified by its program"*

In a sequentially consistent system, the program ordered loads and stores from each processor can be interleaved in a consistent manner. Return to the producer/consumer example given in Figure 18. If we look at the dynamic sequence of loads and stores, we will see something like that shown in Figure 20a. Time goes from top to bottom. Note that the `While {}` is essentially a spin loop on Flag, waiting for it to become a 1, so there will be a series of loads of Flag, until finally one of the loads returns a 1. In Figure 20, we assume that there are two such loads of Flag before it becomes a 1 on the third load. Then, considering the sequences of loads and stores, we see that if they behave in a sequentially consistent manner, then they can be interleaved in a consistent sequence of loads and stores; this is shown in Figure 20a. On the other hand, if we consider the sequence of events that leads to the results shown in shown in Figure 18, then it would be impossible to interleave the loads and stores in a sequentially consistent way. This is shown in Figure 20b. Arcs have been added to show the necessary event sequence as follows. Thread 0's store of A ←9 must precede the Store of Flag ← 1 because Thread 0 must satisfy program order. Similarly, Thread 1's load of Flag=1 must precede the Load of A=0. Moreover, Thread 0's store of Flag ← 1 must precede Thread 1's load of Flag=1 and Thread 1's load of A=0 must precede Thread 0's store of A ←9. The "precedes" arcs form a loop, which implies it is impossible to interleave the two sequences in a sequentially consistent way.

```
Thread0:            Thread1:
Store A←0
Store Flag←0
....
Store A←9
                    Load Flag=0
                    Load Flag=0
Store Flag←1
                    Load Flag=1
                    Load A=9
```

(a)



```
Thread0:            Thread1:
Store A←0
Store Flag←0        Load Flag=0
....                Load Flag=0
Store A←9           Load Flag=1

Store Flag←1        Load A=0
```

(b)

Figure 20. a) Load and store instructions from Threads 0 and 1 can be interleaved in a sequentially consistent manner. B) Example where it is impossible to interleave loads and store in a sequentially consistent manner.

If the interconnection network of Figure 18b were implemented in a way that forces sequential consistent way, then the "expected" result of the example in Figure 20a would occur, and the "unexpected" result of Figure 20b could never occur. Forcing sequential consistency in a multiprocessor implementation may lead to reduced performance, however, because it may place additional constraints on the system. For example, one could implement sequential consistency in the network of Figure 18 by having the memory system send an acknowledge signal back to the storing processor at the time a store actually takes place. Then, if a processor is forced to wait for each store to be acknowledged before performing any other memory operations (loads or stores), sequential consistency follows. However, this implementation would cause the system to slow considerably; every store from a processor would require a "round trip" delay to memory, and overlapping any other memory operation with a store would be inhibited.

In practice, other, cleverer sequential consistency implementations have been proposed and used. We will discuss them in greater detail in Chapter 4. In general, they allow a processor to overlap memory operations, but have additional hardware to check whether sequential consistency may have been violated. If so, the processor is able to "roll back" its state, and resume execution at an earlier point so that sequential consistency is assured. The complexity of implementing SC in a high performance manner has also led to relaxed consistency models, with simpler and/or higher performance implementations, where all of the constraints of sequential consistency are not required.

RELAXED CONSISTENCY MODELS

A relaxed consistency model is a memory ordering model that is not as strict as sequential consistency; this makes them either easier to implement and/or better performing. A wide variety of relaxed consistency models have been proposed and used; some of which contain subtleties and complexities that make them difficult to understand. Rather than consider the full range of consistency models here, let's consider one of the basic relaxed consistency models that is both useful and easy to understand. Further discussion will wait until Chapter 4 when we cover hardware implementation of memory systems. We defer discussion because many of the relaxed consistency models are intertwined with implementation aspects of the memory hardware, and Chapter 4 discusses memory implementations.

In virtually all the relaxed models, synchronization operations such as the Test&Set are defined to be points where consistency must be maintained. This makes sense, because synchronization operations are explicitly included in an ISA to facilitate communication through memory. One of the most relaxed forms of memory consistency, therefore, defines consistency to be maintained only at synchronization instructions; that is, sequential consistency does not have to be maintained for ordinarily loads and stores. This form of consistency is called *Weak Ordering*. (Sequential Consistency is sometimes referred to as "Strong Ordering", but owing to the way that Strong Ordering was originally defined [13], the two, strictly speaking, are not equivalent [14]).

Weak Ordering is defined in the following manner. First, in any program execution, it must appear that, before any synchronization instruction executes, all preceding loads and stores must have been completed. Second, it must appear that the synchronization instruction completes before any subsequent load or store instructions are executed. The term "appear" means that the values observed in registers and memory must be identical to those that would be present if the stated condition holds. In a real implementation, there are often clever things a designer can do so that these conditions do not literally hold at all times; they only appear to, and that is sufficient.

Now we apply Weak Ordering to the example in Figure 18. In this example, Flag is a synchronization variable, and we will use a new opcode Set to set such a synchronization variable to a one. We will use Test as a synchronization instruction which loads a flag (in an ISA that uses condition codes, this load instruction could also set a condition code to be tested with a conditional branch; in our ISA, we will do the test as part of the branch.). The code sequence is illustrated in Figure 21a. Here, because of the ordering conditions for synchronization instructions, the Set by Thread 0 must wait for the store to A to complete before it can execute. The Test by Thread 1 must also complete before following loads execute. If the constraints on the Set and Test instructions are enforced, then the expected result (A=9) will be observed.

```
Thread0:              Thread1:
Store A←0
Reset Flag←0
....                  Test Flag=0
                      Test Flag=0
Store A←9
Set Flag←1
                      Test Flag=1
                      Load A=9
```

(a)

```
Thread0:              Thread1:
Store A←0
MB
Store Flag←0
....                  Load Flag=0
                      Load Flag=0
Store A←9
MB
Set Flag←1
                      Load Flag=1
                      MB
                      Load A=9
```

(b)

Figure 21. Synchronization with Weak Ordering. Consistency is maintained at synchronization instructions.

To implement correct operation with relaxed memory ordering, some ISAs provide explicit *memory barrier* instructions which functionally act as a no-op, but force ordering of preceding and/or following memory operations. For example, the Alpha ISA has a memory barrier (MB) instruction that must wait for all preceding loads and stores to complete before it executes (as a no-op). The Intel IA32 ISA has *fence* instructions that perform a similar function. In the producer/consumer example, then, a regular store instruction can then be used for setting Flag, as long as it is immediately preceded by a MB instruction (see Figure 21b). These barrier instructions can be placed as appropriate (by the compiler or library writer) to enforce ordering when needed. In practice, however, these barriers tend to be applied very liberally to avoid any possibility of bugs, and in some cases may give performance equivalent to brute-force implementations of sequential consistency.

Now, the reason for providing a separate Reset instruction for clearing a lock instead of an ordinary Load instruction (see Figure 11) should be apparent. When a relaxed consistency model, such as Weak Ordering, is implemented, the special opcode tells the hardware that it must impose memory ordering constraints at that point. An ordinary load instruction does not require this ordering enforcement.

As mentioned above, there are a number of consistency models that fall somewhere between sequential consistency and Weak Ordering. We will not discuss the details here; the range of ordering models is discussed in Chapter 4, along with their implementations.

MEMORY COHERENCE

Historically, the terms "consistency" and "coherence" (or "coherency") sometimes have been used interchangeably, but in recent years, they have come to have more specific meanings. In particular, *coherence* describes the memory ordering behavior when accessing a single variable and *consistency* describes the ordering behavior when multiple variables are accessed (as just discussed).

Consider again the producer/consumer example of Figure 18. We consider the ordering of loads and stores separately for the variables A and Flag as shown in Figure 22. If we consider variable A alone (Figure 22a) we see that there is consistent ordering. Similarly, there is a consistent ordering for variable Flag (Figure 22b). Because there is a consistent ordering for both variables when considered separately, we say that the memory system is *coherent*. On the other hand, as shown earlier, the overall ordering is not sequentially consistent. Coherency is defined by considering ordering of loads and stores to individual variables; consistency is defined by considering the ordering among loads and stores to all the variables.

```
Thread0:            Thread1:
Store A←0
Store A←9
                    Load A=9
```

(a)

```
Thread0:            Thread1:
Store Flag←0
                    Load Flag=0
                    Load Flag=0
Store Flag←1
                    Load Flag=1
```

(b)

Figure 22. An illustration that the results of example in Figure 20 provide memory coherence for both a) variable A and b) variable Flag. Consequently, memory is coherent for this example, although it is not sequentially consistent.

In a multiprocessor system with a cache hierarchy changes to variables do not get propagated to all the cached copies instantaneously; changes must propagate through the system, and different cached copies and main memory may get updated at different times. This non-simultaneous update of copies can lead to memory coherence problems if it is not properly handled. Because caches are often a dominant part of the memory coherence problem, this is often characterized as a "cache coherence" problem. However, in general, it is the memory architecture that must be coherent, not just the caches.

Virtually all multiprocessor systems in use today implement cache coherence in hardware. In some respects, memory coherence seems more fundamental than memory consistency. With coherence, loads and stores can be tracked with respect to individual variables; while for consistency, loads and stores must essentially be tracked in all combinations. Straightforward, high performance solutions to the memory coherence problem are well-known; the same cannot be said for sequential consistency implementations.

Implementations of memory (cache) coherence will be discussed in some detail in Chapter 4. In this chapter, where our interest is in the ISA, not hardware implementations, it is safe to say that all the common ISAs embody memory coherence in their definitions.

2.3.5 Transactional Memory

With conventional shared memory, API-visible locks are used as a primitive element for many types of synchronization, and the locks typically map directly to hardware mutex variable and atomic read-modify-write instructions. The high level language programmer is responsible for dealing directly with low level primitives. It would be better to provide API-level features with semantics that avoid details of an implementation in favor of conceptually clear sharing patterns. This is the goal of *transactional memory*, which casts operations on shared data as higher level transactions. Each transaction is a group of memory accesses and modifications which fit together naturally. Transactional memory effectively combines data communication and synchronization.

Transactional semantics have been used in database systems for some time, and have only recently made their way into the shared memory programming model [18]. From the perspective of architecture and hardware implementations, transactional memory is still in the research domain, but it is an approach that is receiving considerable interest and deserves some discussion here.

With transactional memory, sequences of shared memory operations are bundled into "transactions". The key property is that all the operations in one thread's transaction appear to occur atomically to all the other threads. An example, derived from the example Figure 6 is shown in Figure 23. Two data structures are accessed, and both are potentially modified. In Figure 6 there are two separate programmer-defined locks, one for each structure, which must both be acquired before the structures can be operated upon. However, at a higher level, the entire sequence that accesses both structures is conceptually a single operation. In Figure 23 the operations on the two structures are bundled together as a single transaction that will be performed atomically with respect to the other threads. The effect is the same as acquiring, and later releasing, the two separate locks, but the transaction expresses the overall intent in a more transparent way.

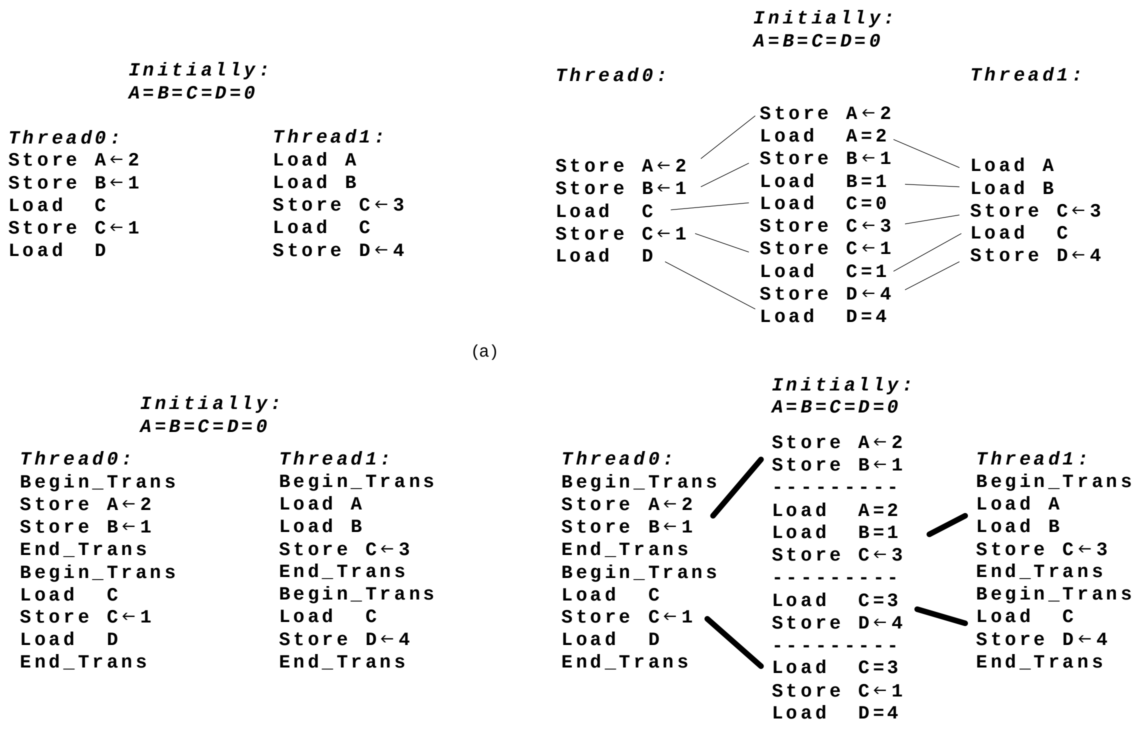
A transaction is at a higher, more abstract level and describes what should be done, rather than how it should be done, as with the case with explicit locks. Of course, the underlying implementation, as determined by the language implementer, could use locks, but it does not have to use locks. A key problem is that current instruction sets and hardware do not provide primitives that lead to high performance implementations of transactional memory. Hence, language designers and computer architects are working together to define both the high level language semantics and the hardware primitives that will support transactional memory in a conceptually clean and efficient manner.

```
<Thread0>                    <Thread1>
Begin_Transaction            Begin_Transaction
<access sruct1>              <access struct1>
<access sruct2>              <access struct2>
End_Transaction              End_Transaction
```

Figure 23. Example of a transaction pattern for shared memory programming.

Transactions are not only more natural at a high level, but they may also provide some relief from the tight constraints of sequential consistency. In particular, transactions become the fundamental unit for memory ordering models rather than individual loads and stores.

Figure 24 shows sequences of memory instructions that access memory; note that there may be other non-memory instructions in the sequences, but they are not shown. Recall that with sequential consistency, only certain interleavings of loads and stores are allowed. For example, one such interleaving is illustrated on the right side of Figure 24a. However, if accesses are divided into transactions (Figure 24b), then proper interleavings must only be maintained at the transaction level. One example is shown on the right side of Figure 24b. The sequentially consistent interleaving in Figure 24a would not be allowed for the transactions given in Figure 24b.



```
        Initially:
        A=B=C=D=0

Thread0:            Thread1:
Store A←2           Load A
Store B←1           Load B
Load  C             Store C←3
Store C←1           Load  C
Load  D             Store D←4
```

```
                      Initially:
                      A=B=C=D=0
Thread0:                                  Thread1:
                      Store A←2
                      Load  A=2
Store A←2             Store B←1           Load A
Store B←1             Load  B=1           Load B
Load  C               Load  C=0           Store C←3
Store C←1             Store C←3           Load  C
Load  D               Store C←1           Store D←4
                      Load  C=1
                      Store D←4
                      Load  D=4
```

(a)

```
        Initially:
        A=B=C=D=0
Thread0:            Thread1:
Begin_Trans         Begin_Trans
Store A←2           Load A
Store B←1           Load B
End_Trans           Store C←3
Begin_Trans         End_Trans
Load  C             Begin_Trans
Store C←1           Load  C
Load  D             Store D←4
End_Trans           End_Trans
```

```
                      Initially:
                      A=B=C=D=0
                      Store A←2
Thread0:              Store B←1           Thread1:
Begin_Trans           ---------           Begin_Trans
Store A←2             Load  A=2           Load A
Store B←1             Load  B=1           Load B
End_Trans             Store C←3           Store C←3
Begin_Trans           --------            End_Trans
Load  C               Load  C=3           Begin_Trans
Store C←1             Store D←4           Load  C
Load  D               ---------           Store D←4
End_Trans             Load  C=3           End_Trans
                      Store C←1
                      Load  D=4
```

(b)

Figure 24. Examples of Sequential Consistency and Transactional Semantics. On the left are code sequences executed by two threads; on the right are potential interleavings under a) Sequential Consistency and b) Transactional Semantics.

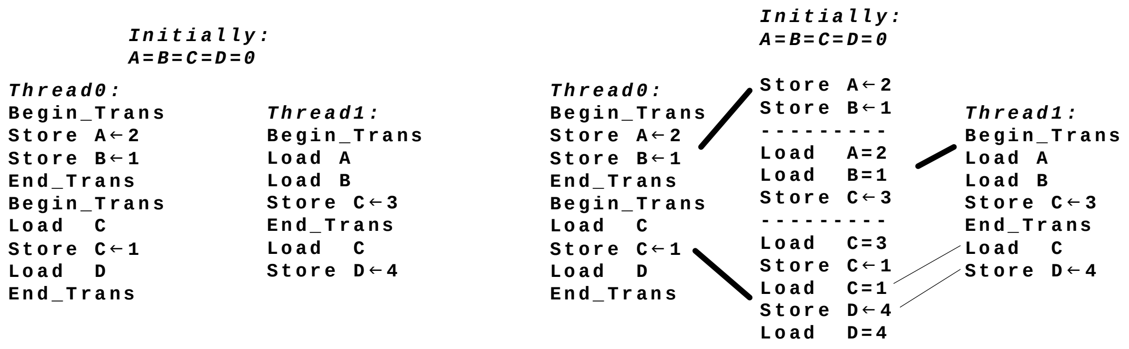
By making transactions atomic, and not the individual loads and stores, there are fewer ordering constraints on individual loads and stores *within* a transaction. The compiler or the hardware can reorder these operations without violating transactional semantics (although data dependences must still be maintained). For example, the two stores in the first transaction of Thread 0 in Figure 24 can be reordered by the hardware and the results will still be correct. Because of atomicity, all other threads will only observe the result after both stores have been performed.

In practice, transactions may be intermixed with loads and stores to shared memory that do not belong to a transaction. This leads to two varieties of transactional semantics. In one class, transactions are atomic only with respect to other transactions; they do not have to be atomic with respect to loads and stores that are not part of transactions. In the other, atomicity is maintained among both transactions and non-transaction loads and stores. The first class is referred to as *weak atomicity* and the second is referred to as *strong atomicity* [17]. The difference is illustrated in Figure 25. In the figure, the last load and store for Thread 1 are not part of a transaction; they are individual loads and stores. With weak atomicity, these loads and stores are not required to respect the atomicity of a transaction, so they can be interleaved inside a transaction as shown on the right side of the figure. With strong atomicity, this interleaving would not be allowed.

Current instruction sets and hardware provide no special support for transactional memory. Consequently, today, it must be supported entirely by software, and this is inefficient. A software implementation, essentially has to save the contents of any state that is modified by a transaction, then check to see if any other thread writes to any of the addresses accessed by a transaction as it executes, and, if so, the transaction is aborted, the initial state is restored, and the transaction must begin again. The state saving and detection of writes to the transaction addresses adds a significant number of instructions and slows down a transaction considerably.

To make transactional memory efficient, it is clear that some type of ISA (and underlying hardware) support will be needed. On the other hand, this ISA/hardware support alone will probably not be enough, there will also have to be a software component of a transactional memory implementation. This interplay between hardware and software and the definition of the interface (in the ISA) is a topic of significant study. Some of the alternatives are described in Chapter 4 when transactional memory hardware support is discussed.

At the ISA level, one approach is to provide begin_transaction, end_transaction instructions as is sugested in Figure 23. Another approach is to provide more primitive operations to assist with the tracking of instructions that access transaction write addresses. These alternatives will also be discussed in greater detail in Chapter 4.



```
        Initially:
        A=B=C=D=0

Thread0:
Begin_Trans         Thread1:
Store A←2           Begin_Trans
Store B←1           Load A
End_Trans           Load B
Begin_Trans         Store C←3
Load  C             End_Trans
Store C←1           Load  C
Load  D             Store D←4
End_Trans
```

```
                         Initially:
                         A=B=C=D=0

Thread0:                 Store A←2
Begin_Trans              Store B←1        Thread1:
Store A←2                ---------        Begin_Trans
Store B←1                Load  A=2        Load A
End_Trans                Load  B=1        Load B
Begin_Trans              Store C←3        Store C←3
Load  C                  ---------        End_Trans
Store C←1                Load  C=3        Load  C
Load  D                  Store C←1        Store D←4
End_Trans                Load  C=1
                         Store D←4
                         Load  D=4
```

Figure 25. With weak atomicity, loads and stores that are not part of a transaction are not required to respect atomicity within a transaction.

2.4 Message Passing

In the message passing programming model, concurrently executing processes or threads communicate by sending and receiving explicit messages. The model is often implemented with an API formed by appending a set of message passing routines to a conventional procedural programming language. Because messages are passed between explicitly identified buffers using only API-defined routines, there is no need for logically shared memory. Another important feature of the message passing programming model is that communication and synchronization are combined in message semantics; they do not have to be handled separately as is the case in most shared memory APIs (transactional memory is an attempt to combine synchronization and communication, however).

MPI, or Message Passing Interface [6], is a standard set of API routines that support the message passing parallel programming model. This set of routines has been combined with a number of popular programming languages including C and FORTRAN. When describing the message passing model, we will use examples from MPI.

2.4.1 Processes and Threads

In this subsection we will consider the creation and management of processes and threads as used in a typical message passing API. Then, in the next subsection, message passing communication and synchronization are considered.

The message passing programming model can be implemented easily on both shared and distributed memory hardware. Consequently, computation can be performed either by threads sharing an address space or by processes with different address spaces. This leads to three different environments in which an API can support message passing:

1) Threads sharing an address space; threads run as part of a single process;

2) Processes having separate address spaces; processes run under the control of a common operating system;

3) Processes having separate address spaces; processes run under the control of different operating systems.

Environments 1) and 2) are best supported on shared memory hardware and the threads and processes run under a common operating system. Consequently, the API can support creation, management, and termination of concurrent processes and threads in much the same way as described for the shared memory model in Section 2.2.1. For example, the Unix OS `fork()`, `exec()`, and `wait()` system calls can be used. In typical usage, a main application process may create, and then manage a number of cooperating threads or processes through these system calls.

Environment 3) is probably the most common situation where message passing is used in practice. In this environment, the concurrently executing processes are under the control of different operating systems. Because there is no common OS, software running under control of one of the operating systems cannot directly perform a system call that creates a process on another system. One solution to this problem is to create processes in a manner that is external to the cooperating user processes. For example, the user may run a shell script that starts processes on the various distributed processors. Then, each of these processes calls a routine that initializes a runtime message passing environment based on OS-provided networking facilities, thereby establishing inter-process communication. This was the approach taken in the first version of MPI, MPI-1, where there are no explicit routines for forking or creating processes; rather, it is assumed that the user first starts a number of separate programs, through means external to MPI. Then each of the programs calls the routine MPI_INIT() which initializes the MPI API environment and sets up the necessary communication linkages; exactly how this is done depends on the actual hardware platform.

MPI extensions, provided in MPI-2 support the creation of processes from inside an MPI program. These extensions employ an interface to an external process manager; the external process manager does the actual process creation. The routine MPI_COMM_SPAWN () creates a number of processes, all executing the same executable binary; MPI_COMM_SPAWN_MULTIPLE() can create processes running different executables.

Once processes are created and communication channels are set up, then the other process management functions are relatively straightforward. For example, in MPI there is a routine to terminate running processes. An important part of a message passing API is the ability to name processes and to direct message sending and receiving to/from specific processes, or groups of processes. Typically when a process is created, the creating process is returned the name of the created process. A process may also get its own name through an API call. In MPI, the process identifier is called the process's *rank*.

There are often cases where inter-process communication or other actions should be confined to a subset of processes. This suggests support for giving groups of processes a common name that can be used, for example, when a message is to be broadcast to all members of the group. MPI provides such support for grouping processes into collections that can communicate with each other. This is illustrated in Figure 26. An MPI *group* contains an ordered set of processes, and an MPI *communicator* is an object that defines a "universe" for communication. The group MPI_COMM_WORLD contains all the processes. In practice, it is common to first define a group, and then create a communicator so that the processes within the group can communicate.

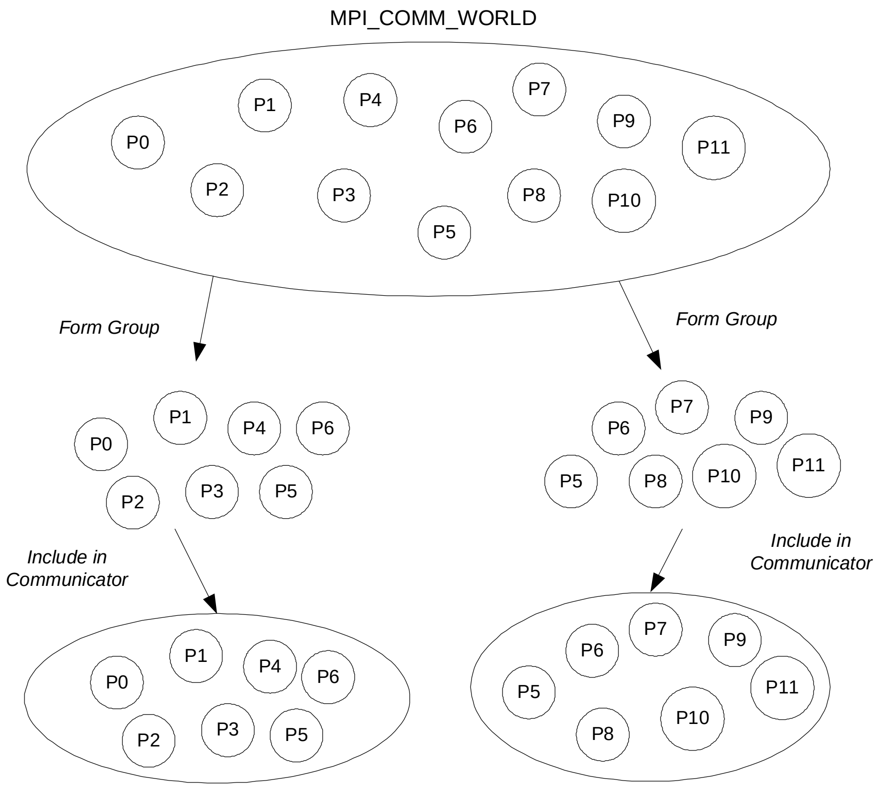


Figure 26. Processes in a message passing API may be organized into groups for collective communication.

2.4.2 Communication and Synchronization

In a pure message passing programming model (see Figure 27), multiple processes each have there own private memory. There are no shared variables, and all data communication takes place via explicit messages. In the simplest case, *point-to-point communication*, one process executes a "send" operation, which creates a message that contains of data items to be sent. Then, another process executes a "receive" operation to read the contents of the sent message. There are also more complex forms of message passing, *collective communication*, where, for example, one process may broadcast a message to all the other processes that are running in parallel.

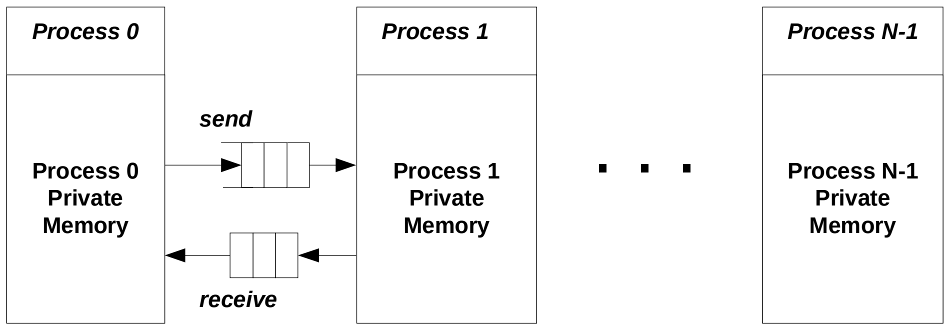


Figure 27. Message Passing

POINT-TO-POINT COMMUNICATION

In point-to-point communication, the sending process calls a routine of the form `send(RecProc, SendBuf,…)` and a receiving process calls `receive(SendProc, RecBuf…)`. For a send routine, the parameter `RecProc` names destination of the message and may indicate a wildcard if a number of processors are valid receivers. The data to be sent is in sending processor's memory space and is identified as `SendBuf`. Other parameters in the send routine may include the size and type of the data to be sent. Similarly, for the receive routine, `SendProc` identifies the process from whom a message is expected and `RecBuf` identifies the memory location in the receivers address space where the message should be delivered.

An example of a point-to-point message passing routine taken from MPI performs a basic send and is of the form: `MPI_Send(buffer, count, type, dest, tag, comm)` , where the arguments are:

`buffer` – data buffer holding data to be sent
`count` – number of data items to be sent
`type` – type of data items to be sent
`dest` – rank (identifier) of the receiving process
`tag` – arbitrary programmer-defined identifier; the tag of a send and the corresponding receive must match. The tag may be a wildcard.
`comm` – communicator number

A basic MPI receive `MPI_Recv (buffer, count, type, source, tag, comm, status)`; its arguments are:

`buffer` – address where received data is to be placed
`count` – number of data items
`type` – type of data items
`source` – rank (identifier) of the sending process
`tag` – arbitrary programmer-defined identifier tag of send and receive must match
`comm` – communicator number
`status` – the source, tag, and number of bytes transferred.

Message sends and receives not only communicate data; they also may provide implicit synchronization. Consequently, a variety of message send/receive routines are supported; they have similar communication semantics but differ in the timing and ordering of sends and receives. This is analogous to memory ordering semantics in the shared memory model. As with memory ordering models, message sequencing is very closely connected with underlying implementations.

In MPI, messages may be synchronous or asynchronous, and they may be blocking or non-blocking. These are pairs of properties are similar, but they are not the same. The distinction between synchronous and asynchronous is a natural distinction that is defined at higher level. The distinction between blocking and non-blocking is more closely related to underlying implementations. These distinctions will be elaborated upon in following paragraphs.

A send routine is *synchronous* if it returns only when a matching receive is called by the receiving process. In essence, there is an acknowledgment from the receiver back to the sender indicating that the message has been received. While this implied acknowledgment is pending, the send routine stalls[1], and no further instructions in the sending process are executed. A synchronous receive routine stalls until a message becomes available.

A send routine is *asynchronous* if the sending process can proceed immediately after executing the send. Similarly, an asynchronous receive routine simply posts an "intent" to receive a message, and, if the message happens to be available it is copied into the receive buffer. If the message is not yet available, the receiving process continues computation, anyway. Both the sender and receiver are given a *request handle* that identifies that particular point-to-point communication. Then, there is a routine that allows the sending and receiving process to test to see whether the message has been received. Given the request handle as in argument, the test routine returns a status flag indicating whether the message is available or has been received. The advantage of asynchronous routines is that they allow the sender/receiver to do useful work while waiting for a message rather than stalling.

The blocking nature of sends and receives depends on availability of buffering that is part of the implementation. A *blocking* send returns as soon as its send buffer has been completely read (as part of the send implementation); otherwise it blocks (stalls) until the send buffer has been read. A *non-blocking* send does not wait for the send buffer to be read; it can go ahead with computation. For the non-blocking send a test instruction allows the sending process to determine if the buffer is empty. A blocking receive blocks until data is available in the receive buffer. A non-blocking receive can continue executing in a manner similar to an asynchronous receive.

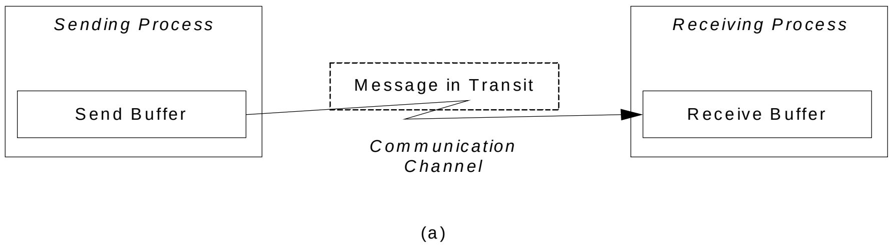
An example of non-blocking communication in MPI is given in Figure 28. This example, adapted from an MPI tutorial [6], performs neighbor communication among processes that are logically connected in a ring topology. The identifier of a given process is rank, the name of the previous process in the ring is prev, and the name of the next process in the ring is next. After initialization routines, each process computes its prev and next, modulo the number of processes, numprocs. Then the process perform a pair of non-blocking receive routines, MPI_Irecv, directed at both the prev and next neighbors; this is followed by a matching pair of a non-blocking send routines MPI_Isend. The two buffers, one for each neighbor are buf[1] and buf[2]. Request handles are in the array reqs[4]. The sent message is a "1" flag. The MPI_Waitall is a routine that waits for all the non-blocking communication to complete.

[1] The term "stall" is used rather than "block" to avoid confusion with blocking/non-blocking messages.

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
int numprocs, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0) prev = numprocs - 1;
if (rank == (numprocs - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
 }
```

Figure 28. Example of non-blocking communication in MPI.

There are at least two ways that buffering can be implemented, and specific blocking semantics depend on this implementation. Figure 29 illustrates the two models. The model in Figure 29a consists of the user defined buffer that holds data at the sending end, some kind of communication channel over which the message travels, and the user-defined buffer for holding data at the receiving end. A send signals that valid data is available in the send buffer, but the data is maintained there until a receive copies the message from the send buffer to the receive buffer. In the second implementation (Figure 29b), there is additional system-defined buffering; this buffering is often associated with the processor at the receiving end, but does not have to be. A send causes the message to be copied into the system buffer. Then, a receive copies the message into the receive buffer. An advantage of the first implementation is that there is no storage for system buffering, and there is less data copying so it is potentially faster. However, it means that a send must block until a receive has been executed at the other end. In this sense, it is similar to a synchronous send. In the second implementation, the send may continue processing before the receive has taken place; in this case, it is similar to an asynchronous send. In either of the two buffering models, a receive blocks (stalls) until data is available to be copied into its receive buffer. In summary, the stalling for synchronous operations is implementation dependent and is written into the semantics of the send and receive routines. The stalling for blocking operations is dependent on the buffering in an implementation.



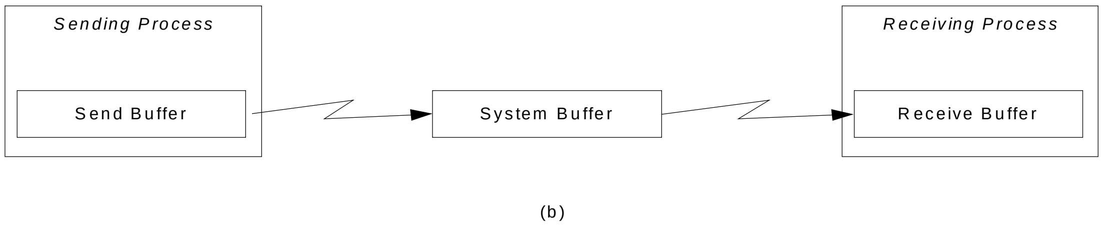


Figure 29. Semantic models for MPI message passing; a) messages are passed directly from the send buffer to the receive buffer; b) there is intermediate system buffering between the send and receive buffers.

Blocking sends and receives are especially prone to deadlock, depending on the implementation. Consider an implementation as in Figure 29a. In Figure 30a, two processes swap data via blocking sends and receives; Process 0 performs a send to process 1 followed by a receive from process 1, and process 1 does the opposite. This would seem to be a natural way for a pair of processes to exchange information, but it deadlocks. Because a send blocks until it buffer is emptied, both processes will block on their send, waiting for the other process to perform a receive, which will cause the deadlock. Deadlock may also occur in systems as illustrated in Figure 29b if the implementation should run out of system buffer space, thereby blocking the copy from the send buffer into the system buffer. To avoid deadlock, a sequence such as that given in Figure 30b should be used. This approach, however, serializes the two pairs of send/receives; i.e., they cannot be overlapped.

```
<Process 0>                        <Process 1>
Send(Process1, Message);           Send(Process0, Message);
Receive(Process1, Message)    ;    Receive(Process0, Message);
                             (a)


<Process 0>                        <Process 1>
Send(Process1, Message);           Receive (Process0, Message);
Receive(Process1, Message)    ;    Send (Process0, Message);
                             (b)
```

Figure 30. Two processes swap messages via blocking sends and receives; a) if both send, then receive, there is deadlock b) deadlock can be avoided by carefully ordering the sends and receives.

To summarize, Figure 31 illustrates four cases; in all four, the send is performed before the receive. In in Figure 31a, the send and receive are synchronous, so the send stalls until the receive executes and provides an acknowledgement. The actual copying of data happens at some point between the send and receive, but does not directly affect the timing. Figure 31b illustrates the asynchronous blocking case where there is no system buffering. The timing is similar to the synchronous timing, except it is the emptying of the sender's buffer that causes the sender to stop stalling. Figure 31c illustrates the asynchronous blocking case where there is intermediate system buffering. In this case, the send only waits until data is copied into the system buffer. Finally, Figure 31d illustrates the asynchronous non-blocking case. The sending process is able to proceed immediately (as with the blocking case and intermediate buffering, although in this case there is no assurance that the sender's buffer has been emptied, so it should first test to see if the buffer has been emptied before performing another send.

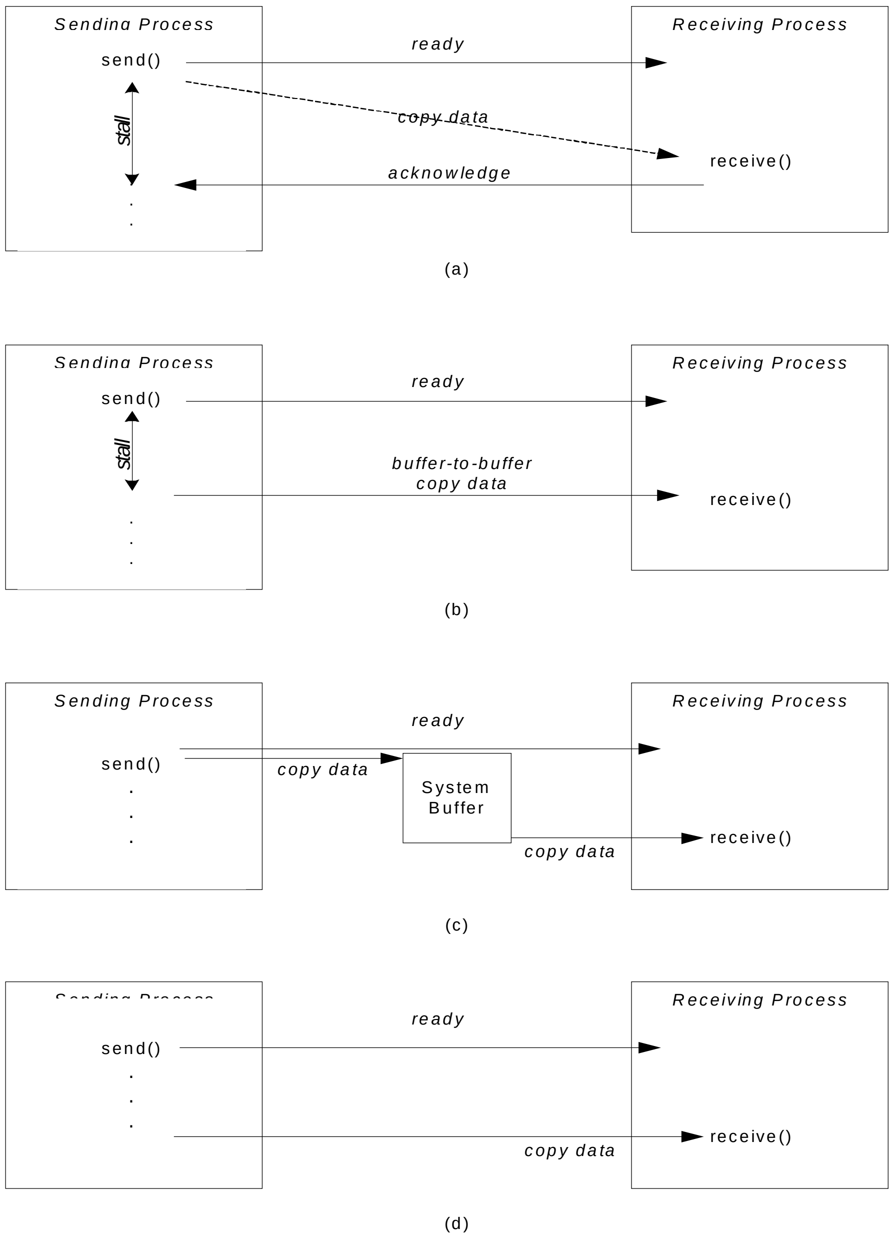


Figure 31. Send and receive timing when the send is performed before the receive a) synchronous, b) asynchronous, blocking with no intermediate buffer; c) blocking with an intermediate buffer, d) asynchronous and non-blocking.

MPI defines message passing to be *synchronous* or *asynchronous*, and *blocking* and *non-blocking*, as we have just defined (MPI was used as a guide in formulating these definitions). The examples of MPI_send and MPI_recv given at the beginning of this section are asynchronous and blocking. The routine MPI_Ssend is the synchronous version of the MPI_send. In a sense the synchronous version is "stronger" than the blocking version, and some implementations may implement the blocking version with the synchronous version.

MPI also contains non-blocking sends and receives, these are MPI_Isend and MPI_Irecv. Because of their non-blocking nature, these send/receives need additional API support, as pointed out above.

MPI provides an MPI_test routine that checks the status of a pending send or receive operation. MPI also provides an MPI_Wait, which causes a sender to block on an already full send buffer. An MPI_Wait was used in the example given in Figure 28.

COLLECTIVE COMMUNICATION

Besides point-to-point communication, the message passing model also includes collective communication, for example, when one process broadcasts the same message to all the other processes. A broadcast is illustrated in Figure 32a. The routine `broadcast(SendBuf, SendProc, Communicator, …)` sends a message from SendProc's send buffer to all the processors in the communicator group; in the example, all the processors shown are in the communicator group. Other more elaborate patterns of group communication are also possible. For example, a scatter operation distributes a sequence of messages to a number of different processes. The routine `scatter(SendBuf, RcvBuf, SendProc, Communicator)` is illustrated in Figure 32b. A gather is a collective receive operation that is gathers messages from a number of `processes` into a process's receive buffer. The routine `gather(SendBuf, RcvBuf, SendProc, Communicator)` is illustrated in Figure 32c.

In MPI, a communicator is associated with collective communication and identifies the processes that will participate. A broadcast, for example, is MPI_`Bcast(*buffer, count, datatype, root, communicator)`; here, "root" is the identifier of the broadcasting process.

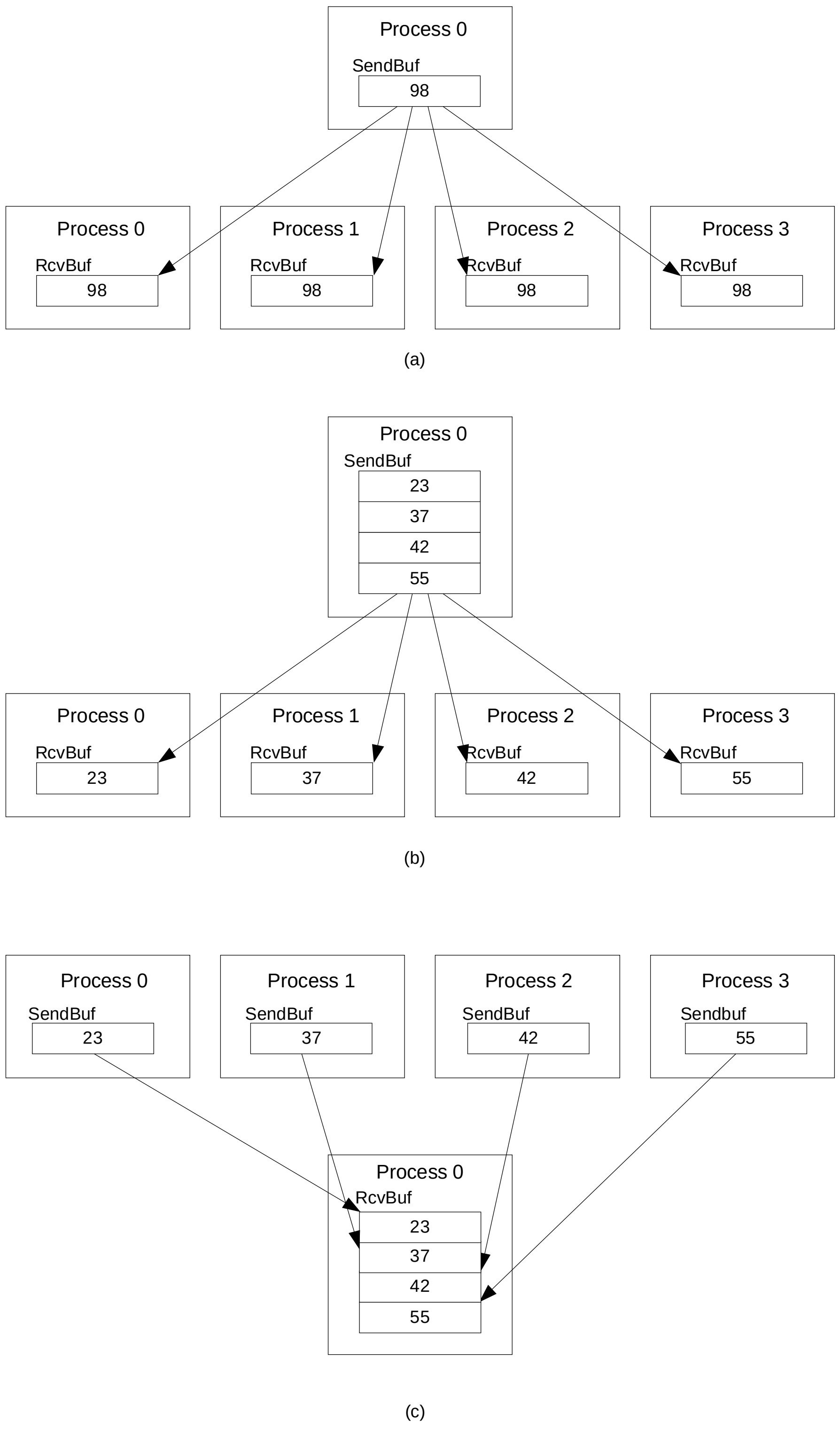


Figure 32. Collective communication a) broadcast, b) scatter, c) gather.

2.5 Message Passing Implementations

Even though message passing can be performed between processes with private memory address spaces, the message passing programming model does not *require* hardware memory that uses distributed (non-shared) memory. It can be built on either shared memory hardware or distributed memory hardware. Implementations on hardware shared memory are not uncommon. On the other hand, if the hardware uses distributed memory, then the message passing programming model is used predominantly (as compared with the shared memory model).

It is also possible to mix the shared memory model and message passing model on the same shared memory hardware platform. For example, threads belonging to the same process may pass messages for some types of data communication, and use shared memory for others. Which is used, and in what combination, is up to the programmer or software development team.

There is typically no special ISA or ABI level support for message passing in most real systems. This is one of the big advantages of the message passing model – the implementer can just lump together existing hardware platforms with a network and get them to work together with neither OS changes (or minimal changes) nor specific ISA support (for example, any memory ordering model will do). Some compute servers where program parallelism is a primary application area may provide additional system software support, however.

Its not that there haven't been proposals for ISA level support – there have. However, these proposals have been made in the context of special purpose parallel computer systems, where the predominant application is program parallelism (to make a single program run very fast) and, in that context, message passing support leads to greater efficiency. The main objective of ISA (and hardware) support is reduction of communication overheads.

To describe implementations of message passing APIs in more detail, we return to the three API-supported models listed at the beginning of the previous section and consider ways in which they can be implemented.

1) Threads sharing an address space on a shared memory platform. Here, much of the implementation can be done at user level with the assistance of runtime software. Because of the common, shared address space, message passing can be managed at the user level with normal load and store instructions. For example, a send routine would call the API runtime which would post the presence of a message in a shared memory runtime table, and then either return immediately (for a non-blocking or asynchronous send) or enter a wait loop (or be placed into an runtime-managed thread queue). A receive routine would also call the runtime. If the runtime finds that a matching message has been posted, then it would copy the message from the sender's buffer to the receiver's, using load and store instructions. Otherwise, the runtime would either return to the receiver (for a non-blocking or asynchronous receive) or wait until a message is sent. The key point is that in this implementation, user-level runtime code can implement the message passing API. Consequently, it would be relatively fast and efficient when compare with a distributed memory implementation which would require operating system intervention. One disadvantage of this shared memory implementation is that the message buffers are not protected from accidental overwriting by a "rogue" thread.

2) Processes running on shared memory hardware communicate via messages in shared regions of their address space. For the most part the implementation would be similar to the one given above, with user level runtime code providing the implementation. Here, there are a couple of alternatives. In one,

the programmer could make sure that all message buffers are in the shared regions of memory. Then, message passing would be managed by the runtime, and messages would pass directly from the sender's buffer to the receiver's as in Figure 29b. Alternatively, the implementation might set aside a special shared memory region for message buffering and use intermediate buffering as illustrated in Figure 29b; although in this case the buffering would be in user memory space rather than system space. In this case, the memory regions of the parallel processes would be protected from over-writing due to a bug in another process.

3) Processes on distributed memory hardware communicate via network hardware; the processes have no shared memory regions and are under control of different operating systems. If message passing is used among processes in a distributed memory system with separate operating systems on each compute node, then the API runtime software must convert message sends (or receives) into system calls that use networking routines to communicate from one process to another (the direct communication is among the runtimes that support the processes. This method is substantially slower, because of the need for OS intervention to pass individual messages. There are techniques for reducing the overheads. For example, the message send may require OS intervention with the receiving OS placing the message into a user level buffer at the receiver's end. Then, the receive can be done in the runtime, without a system call. This is an example as shown in Figure 29b where the system buffer is in the receiver's runtime space.

2.6 Summary

Figure 33 illustrates the architectures and programming models by giving commonly used examples. This is the only chapter where we will discuss the upper levels of the architecture stack (API and ABI) in any detail. The API and ABI are discussed in this chapter primarily to provide background for the remainder of the book. They are of interest, however, because certain hardware tradeoffs are influenced by characteristics of the API and ABI levels. As Figure 33 suggests, all the popular APIs and ABIs run on conventional ISAs, which very similar features for supporting multi-threading.

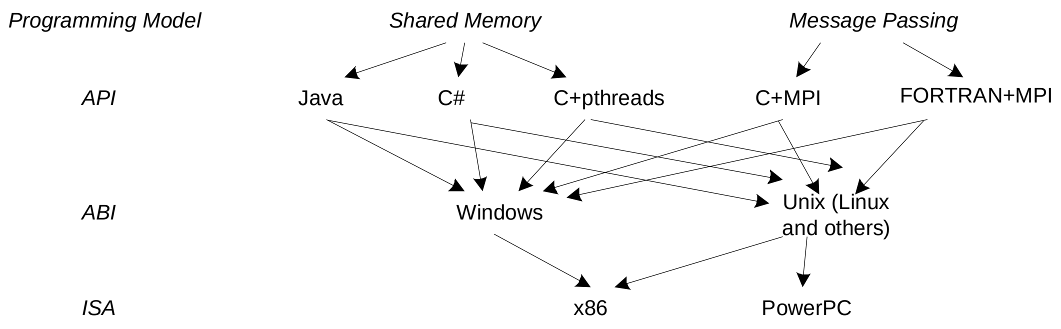


Figure 33. Examples of commonly used programming models and architectures.

The high level material is for background and we do not discuss it further, except, perhaps in examples. The ISA level issues are important for some of the multi-core hardware discussions. We know what must be supported, now in the remainder of the book, we will discuss how it is done. OS issues will be discussed more in the context of clustered architectures (where communication and synchronization are implemented via the OS).

2.7 References

1. A. H. Karp, "Programming for Parallelism," *IEEE Computer*, pp. 43-57, May 1987.
2. A. Muys, A Pthreads Tutorial.
3. P. E. McKenney, Selecting Locking Designs for Parallel Programs PLoPD-II, 1996.
4. H. Sutter and J. Larus, Software and the Concurrency Revolution, ACM Queue, September 2005.
5. J. Larus, Software Challenges in Nanoscale Technology, talk at CRA workshop, Dec. 2005.
6. Maui High Performance Computing Center, SP Parallel Programming Workshop -- Message Passing Interface, 2003.
7. MPI Forum, MPI-2: Extensions to the Message Passing Interface, 2003.
8. B. Barney, Posix Threads Programming, Lawrence Livermore National Laboratory.
9. S. Dobrev, CSI4140 Lecture notes on Message Passing Communication Patterns and Programming, University of Ottawa.
10. S. Oaks and H. Wong, Advanced Synchronization in Java Threads, excerpt from *Java Threads*, O'reilly Publishers.
11. D. B. Skillicorn, J. M. D. Hill and W. F. McColl, "Questions and answers about BSP", *Journal of Scientific Programming*, Fall 1997.
12. Leslie Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers*, September 1979, pp. 690-691.
13. M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors", IEEE Computer, vol. 21, pp. 9-21, February 1988.
14. Sarita V. Adve and Kourosh Gharachorloo, Shared Memory Consistency Models: A Tutorial, IEEE Computer, 29(12):66-76, December 1996.
15. S. V. Adve and M. D. Hill, 'Weak Ordering- A New Definition", Proc. 17th Annual International Symposium on Computer Architecture, pp. 2-14, June 1990.
16. K. Gharachorloo et al., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," Proc. ASPLOS IV, pp. 245--257, Apr. 1991.
17. C. Blundell, et al., "Deconstructing Transactional Sematics: The Subtleties of Atomicity," *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.
18. L. Hammond, et al., "Programming with Transactional Coherence and Consistency (TCC)", *Proc. ASPLOS,* October 2004.