Chapter 12 Multithreaded Programming and Multiprocessors
(Revision number 16)

Multithreading is a technique for a program to do multiple tasks, perhaps concurrently. Since the early stages of computing, exploiting parallelism has been a quest of computer scientists. As early as the 70's, languages such as *Concurrent Pascal* and *Ada* have proposed features for expressing program-level concurrency. Humans think and do things in parallel all the time. For example, you may be reading this book while you are listening to some favorite music in the background. Often, we may be having an intense conversation with someone on some important topic, while working on something with our hands, may be fixing a car or folding our laundry. Given that computers extend the human capacity to compute, it is only natural to provide the opportunity for the human to express concurrency in the tasks that they want the computer to do on their behalf. Sequential programming forces us to express our computing needs in a sequential manner. This is unfortunate since humans think in parallel but end up coding up their thoughts sequentially. For example, let us consider an application such as video surveillance. We want the computer to gather images from ten different cameras continuously, analyze each of them individually for any suspicious activity, and raise an alarm in case of a threat. There is nothing sequential in this description. If anything, the opposite is true. Yet if we want to write a computer program in C to carry out this task, we end up coding it sequentially.

The intent of this chapter is to introduce concepts in developing multithreaded programs, the operating system support needed for these concepts, and the architectural support needed to realize the operating system mechanisms. It is imperative to learn parallel programming and the related system issues since single chip processors today incorporate multiple processor cores. Therefore, parallel processing is becoming the norm these days from low-end to high-end computing systems. The important point we want to convey in this chapter is that the threading concept and the system support for threading are simple and straightforward.

12.1 Why Multithreading?

Multithreading allows expressing the inherent parallelism in the algorithm. A *thread* is similar to a process in that it represents an active unit of processing. Later in this chapter, we will discuss the semantic differences between a thread and a process. Suffice it to say at this point that a user level process may comprise multiple threads.

Let us understand how multithreading helps at the programming level. First, it allows the user program to express its intent for concurrent activities in a modular fashion, just as the procedure abstraction helps organize a sequential program in a modular fashion. Second, it helps in overlapping computation with inherent I/O activities in the program. We know from Chapter 10, that DMA allows a high-speed I/O device to communicate directly with the memory without the intervention of the processor. Figure 12.1 shows this situation for a program that periodically does I/O but does not need the result of the

I/O immediately. Expressing the I/O activity as a separate thread helps in overlapping computation with communication.

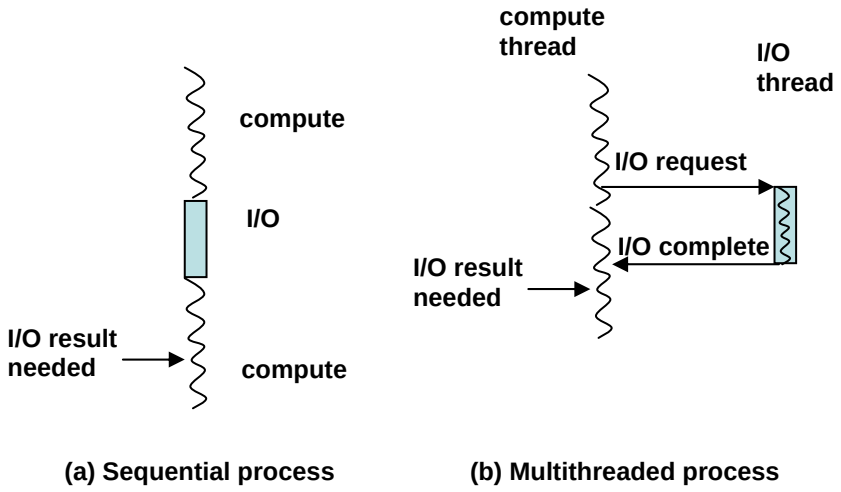


Figure 12.1: Overlapping Computation with I/O using threads

Next, let us see how multithreading helps at the system level. It is common to have multiple processors in the same computer or even in a single chip these days. This is another important reason for multithreading since any expression of concurrency at the user level can serve to exploit the inherent hardware parallelism that may exist in the computer. Imagine what the program does in a video surveillance application. Figure 12.2 shows the processing associated with a single stream of video in such an application. The digitizer component of the program continually converts the video into a sequence of frames of pixels. The tracker component analyzes each frame for any content that needs flagging. The alarm component takes any control action based on the tracking. The application pipeline resembles the processor pipeline, albeit at a much coarser level. Thus if we have multiple processors in the computer that work autonomously, then they could be executing the program components in parallel leading to increased performance for the application.

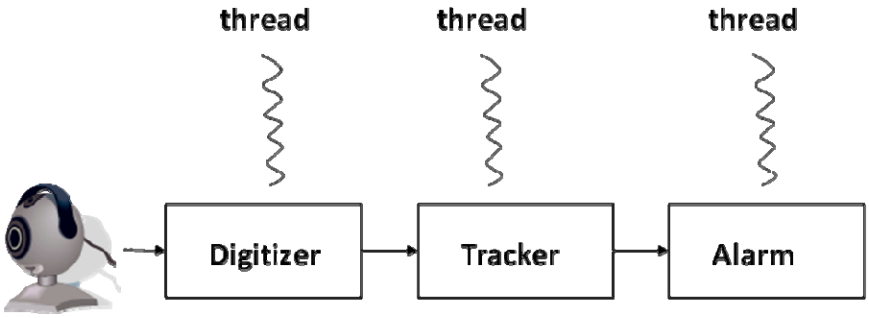


Figure 12.2: Video processing pipeline

Thus, multithreading is attractive from the point of view of program modularity, opportunity for overlapping computation with I/O, and the potential for increased performance due to parallel processing.

We will use the application shown in Figure 12.2 as a running example to illustrate the programming concepts we develop for multithreading in this chapter.

12.2 Programming support for threads

Now that we appreciate the utility of threads as a way of expressing concurrency, let us understand what it takes to support threads as a programming abstraction. We want to be able to dynamically *create* threads, *terminate* threads, *communicate* among the threads, and *synchronize* the activities of the threads,

Just as the system provides a math library for common functions that programmers may need, the system provides a library of functions to support the threads abstraction. We will explore the facilities provided by such a library in the next few subsections. In the discussion, it should be understood that the syntax used for data types and library functions are for illustration purposes only. The actual syntax and supported data types may be different in different implementations of thread libraries.

12.2.1 Thread creation and termination

A thread executes some program component. Consider the relationship between a process and the program. A process starts executing a program at the *entry* point of the program, the *main* procedure in the case of a C program. In contrast, we want to be able to express concurrency, using threads, at *any* point in the program *dynamically*. This suggests that the entry point of a thread is *any user-defined procedure*. We define *top-level* procedure as a procedure name that is visible (through the visibility rules of the programming language) wherever such a procedure has to be used as a target of thread creation. The top-level procedure may take a number of arguments to be passed in.

Thus, a typical thread creation call may look as shown below:

```
thread_create (top-level procedure, args);
```

Essentially, the thread creation call names the top-level procedure and passes the arguments (packaged as args) to initiate the execution of a thread in that procedure. From the user program's perspective, this call results in the creation of a *unit of execution* (namely a *thread*), concurrent with the current thread that made the call (Figure 12.3 before/after picture).

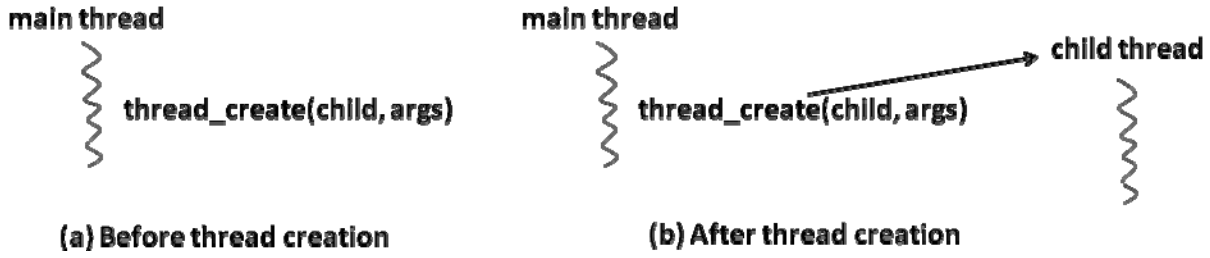


Figure 12.3: Thread creation

Thus, a **`thread_create`** function *instantiates* a new and independent entity called a *thread* that has a life of its own.

This is analogous to parents giving birth to children. Once a child is born, he or she roams around the house doing his/her own thing (within the limits set by the parents) independent of the parent. This is exactly what happens with a thread. Recall that a program in execution is a process. In a sequential program there is only one *thread of control*, namely, the process. By a thread of control, we mean an active entity that is roaming around the memory footprint of the program carrying out the intent of the programmer. Now that it has a life of its own, a thread (within the limits set by the parent process) can do its own thing. For example, in Figure 12.3, once created, "child" can make its own procedure calls programmed into the body of its code independent of what "main" is doing in its code body past the thread creation point.

Let us understand the limits set for the child thread. In the human analogy, a parent may place a gate near the stairs to prevent the child from going up the stairs, or child proof cabinets and so on. For a thread, there are similar limits imposed by both the programming language and the operating system. A thread starts executing in a top-level procedure. In other words, the starting point for a threaded program is a normal sequential program. Therefore, the visibility and scoping rules of the programming language applies to constrain the data structures that a thread can manipulate during its lifetime. The operating system creates a unique and distinct memory footprint for the program called an *address space*, when it instantiates the program as a process. As we saw in earlier chapters, the address space contains the code, global data, stack, and heap sections specific to each program. A process plays in this "sandbox". This is exactly the same "sandbox" for children of this process to play in as well.

The reader may be puzzled as to the difference between a "process" and a "thread". We will elaborate on this difference more when we get to the section on operating system support for threads (Section 12.6). Suffice it to say at this point that the amount of state that is associated with a process is much more than that with a thread. On the other hand, a thread shares the parent process's address space, and in general has lesser state information associated with it than a process. This makes a process a heavier-weight entity compared to a thread. However, both are independent threads of control within the process's address space and have lives of their own.

One fundamental difference between a process and thread is memory protection. The operating system turns each program into a process, each with its own address space that acts as a wall around each process. However, threads execute within a single address space. Thus, they are not protected from each other. In human terms, one cannot walk into a neighbor's house and start scribbling on the walls. However, one's children (if not supervised) can happily go about scribbling on the walls of the house with their crayons. They can also get into fistfights with one another. We will see shortly how we can enforce some discipline among the threads to maintain a sense of decorum while executing within the same address space.

A thread automatically terminates when it exits the top-level procedure that it started in. Additionally, the library may provide an explicit call for terminating a thread in the same process:

```
thread_terminate (tid);
```

Where, **tid** is the system-supplied identifier of the thread we wish to terminate.

Example 1:
Show the code fragment to instantiate the digitizer and tracker parts of the application shown in Figure 12.2

Answer:

```
digitizer()
{
    /* code for grabbing images from camera
     * and share the images with the tracker
     */
}

tracker()
{
    /* code for getting images produced by the digitizer
     * and analyzing an image
     */
}

main()
{
     /* thread ids */
     thread_type digitizer_tid, tracker_tid;


     /* create digitizer thread */
     digitizer_tid = thread_create(digitizer, NULL);

     /* create tracker thread */
     tracker_tid = thread_create(tracker, NULL);

     /* rest of the code of main including
      * termination conditions of the program
      */
}
```

Figure 12.4: Code fragment for thread creation

Note that the shaded box is all that is needed to create the required structure.

12.2.2 Communication among threads

Threads may need to share data. For example, the digitizer in Figure 12.2 shares the buffer of frames that it creates with the tracker.

Let us see how the system facilitates this sharing. As it turns out, this is straightforward. We already mentioned that a threaded program is created out of a sequential program by turning the top-level procedures into threads. Therefore, the data structures that are visible to multiple threads (i.e., top-level procedures) within the scoping rules of the original program become shared data structures for the threads. Concretely, in a programming language such as C, the global data structures become shared data structures for the threads.

Of course, if the computer is a multiprocessor, there are system issues (at both the operating system and hardware levels) that need to be dealt with it to facilitate this sharing. We will revisit these issues in a later section (Section 12.8).

Example 2:

Show the data structure definition for the digitizer and tracker threads of Figure 12.2 to share images.

Answer:

```
#define MAX 100                 /* maximum number of images */

image_type frame_buf[MAX];      /* data structure for
                                 * sharing images between
                                 * digitizer and tracker
                                 */

digitizer()
{
  loop {
    /* code for putting images into frame_buf */
  }
}

tracker()
{
  loop {
    /* code for getting images from frame_buf
     * and analyzing them
     */
  }
}
```

Note: The shaded box shows the data structure created at the global level to allow both the tracker and digitizer threads to share.

12.2.3 Data race and Non-determinism

In a sequential program, we never had to worry about the integrity of data structures since there are no concurrent activities in the program by definition. However, with multiple threads working concurrently within a single address space it becomes essential to ensure that the threads do not step on one another. We define *data race* as a condition in which multiple concurrent threads are simultaneously trying to access a shared variable with at least one of the threads trying to write to the shared variable.

Consider the following code fragment:
Scenario #1:

```
int flag = 0;  /* shared variable initialized to zero */
```

Thread 1

```
while (flag == 0) {
   /* do nothing */
}
```

Thread 2

```
.
.
.
if (flag == 0) flag = 1;
.
.
```

Threads 1 and 2 are part of the same process. Per definition, threads 1 and 2 are in a data race. Thread 1 is in a loop continuously reading shared variable flag, while thread 2 is writing to it in the course of its execution. On the surface, this setup might appear to be a problem, since there is a data race between the two threads. However, most likely, this is an intentional data race, wherein thread 1 is awaiting the value of flag to change by thread 2. Thus, data race does not always imply that the code is erroneous.

On the other hand, consider, the following code fragment:

Scenario #2:

```
int count = 0; /* shared variable initialized to zero */

Thread 1 (T1)          Thread 2 (T2)          Thread 3 (T3)
.                      .                      .
.                      .                      .
count = count+1;       count = count+1;       count = count+1;
.                      .                      .
.                      .                      .
Thread 4 (T4)
.
.
printf("count = %d\n", count);
.
.
```

This is another data race among all the four threads. What value will thread 4 print? Each of the threads 1, 2, and 3, are adding 1 to the current value of count. However, what is the current value of count seen by each of these threads? Depending on the order of execution of the increment statement (count = count+1), the printf statement in thread 4 will result in different values being printed. This is illustrated in Figure 12.5 wherein four different order of execution are captured just to illustrate the point.

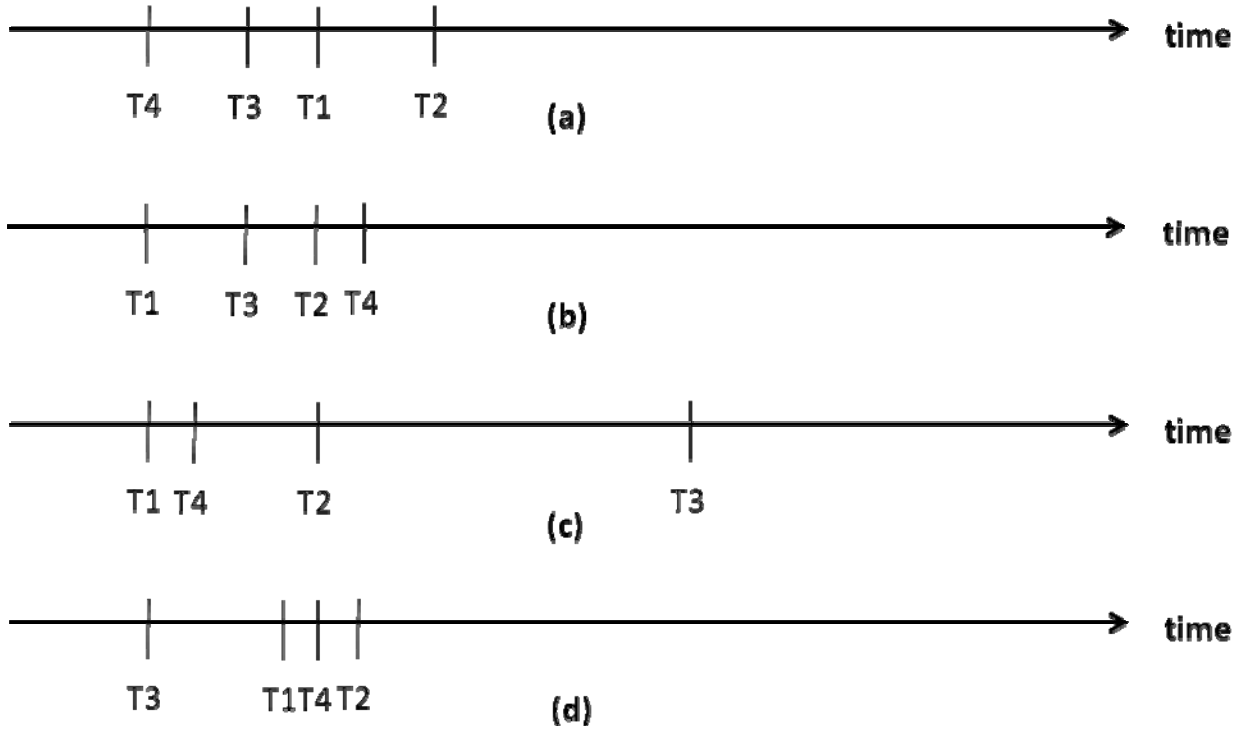


Figure 12.5: Examples of possible executions of Scenario 2 on a uniprocessor using non-preemptive scheduling of threads

The first question that should arise in the mind of the reader is why there are many different possible executions of the code shown in Scenario 2. The answer is straightforward. The threads are concurrent and execute asynchronously with respect to each other. Therefore, once created, the order of execution of these threads is simply a function of the available number of processors in the computer, any dependency among the threads, and the scheduling algorithm used by the operating system.

The timeline of execution shown in Figure 12.5 for scenario #2 assumes that there is a single processor to schedule the threads and that there is no dependency among the threads. That is, once spawned the threads may execute in any order. Further, a non-preemptive scheduling discipline is assumed. The following example illustrates that there are more than 4 possible executions for these threads.

Example 3:

Assume there are 4 threads in a process. Assume that the threads are scheduled on a uniprocessor. Once spawned, each thread prints its thread-id and terminates. Assuming a non-preemptive thread scheduler, how many different executions are possible?

Answer:

As stated in the problem, the threads are independent of one another. Therefore, the operating system may schedule them in any order.

The number of possible executions of the 4 threads = **4!**

Thus, execution of a parallel program is a fundamental departure from the sequential execution of a program in a uniprocessor. The execution model presented to a sequential program is very simple: the instructions of a sequential program execute in the *program* order of the sequential program[1]. We define *program order* as the combination of the textual order in which the instructions of the program appear to the programmer, and the logical order in which these instructions will be executed in every run of the program. The logical order of course depends on the intended semantics for the program as envisioned by the programmer. For example, if you write a high-level language program the source code listing gives you a textual order the program. Depending on the input and the actual logic you have in the program (in terms of conditional statements, loops, procedure calls, etc.) the execution of the program will result in taking a specific path through your source code. In other words, the behavior of a sequential program is *deterministic*, which means that for a given input the output of the program will remain unchanged in every execution of the program.

It is instructive to understand the execution model for a parallel program that is comprised of several threads. The individual threads of a process experience the same execution model as a sequential program. However, there is no guarantee as to the order of execution of the different threads of the same process. That is, the behavior of a

[1] Note that, as we mentioned in Chapter 5 (Section 5.13.2.4), processor implementation might choose to reorder the execution of the instructions, so long as the appearance of program order to the programmer is preserved despite such reordering.

parallel program is *non-deterministic*. We define a non-deterministic execution as one in which the output of the program for a given input may change from one execution of the program to the next.

Let us return to scenario #2 and the 4 possible non-preemptive executions of the program shown in Figure 12.5. What are the values that would be printed by T4 in each of the 4 executions?

Let us look at Figure 12.5-(a). Thread T4 is the first one to complete execution. Therefore, the value it would print for count is zero. On the other hand, in Figure 12.5-(b), thread T4 is the last to execute. Therefore, the value printed by T4 would be 3 (each of the executions of T1, T2, and T3 would have incremented count by 1). In Figures 12.5-(c) and 12.5-(d), T4 would print 1, and 2, respectively.

Further, if the scheduler is preemptive, many more interleavings of the threads of an application are possible. Worse yet, a statement such as

```
count = count + 1
```

would get compiled into a sequence of machine instructions. For example, compilation of this statement would result in series of instructions that includes a load from memory, increment, and a store to memory. The thread could be preempted after the load instructions before the store. This has serious implications on the expected program behavior and the actual execution of a program that has such data races. We will revisit this issues shortly (Section 12.2.6) and demonstrate the need for atomicity of a group of instructions in the context of a programming example.

Figure 12.6 demonstrates how due to the non-determinism inherent in the parallel programming model, instructions of threads of the same program may get arbitrarily interleaved in an actual run of the program on a uniprocessor.

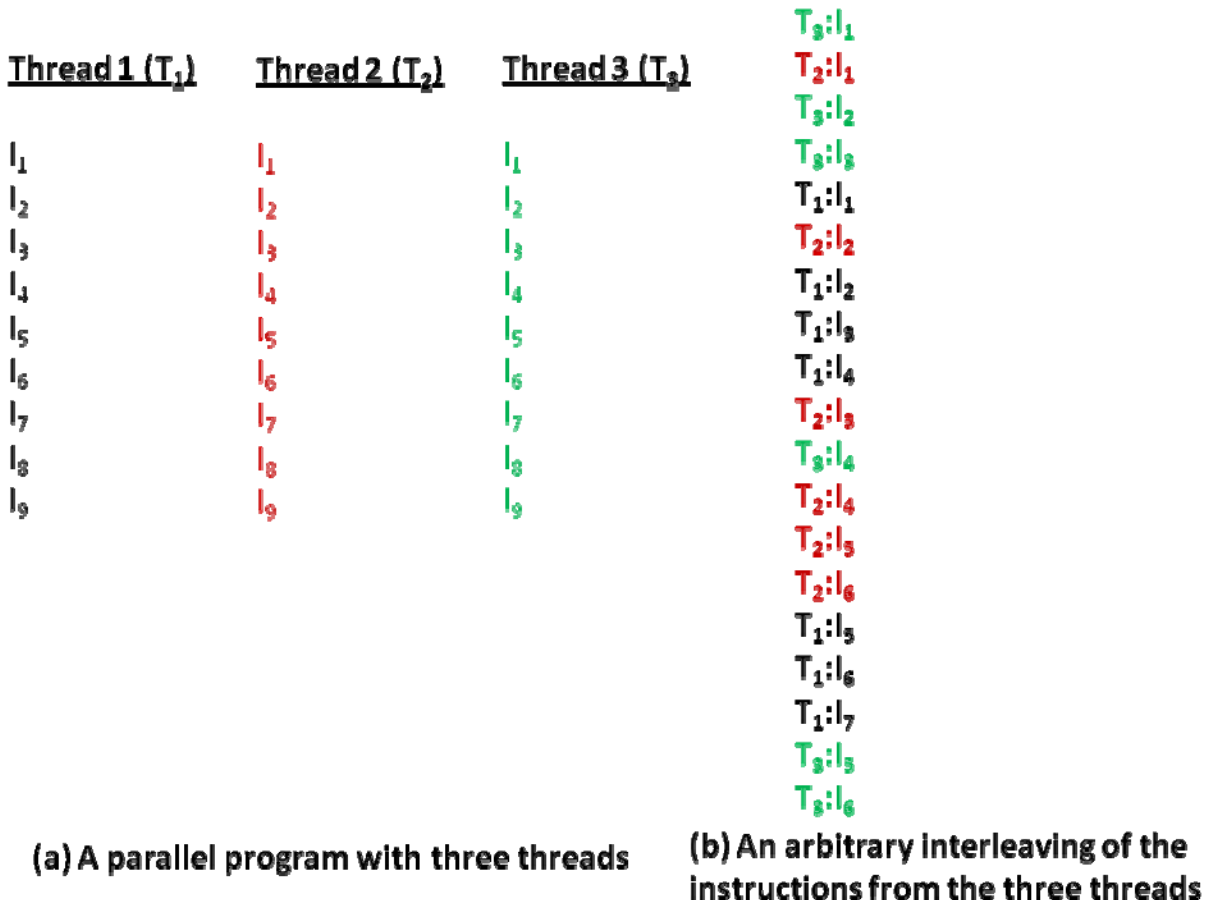


Figure 12.6: A parallel program and a possible trace of its execution on a uniprocessor

There are a few points to take away from Figure 12.6. Instructions of the *same* thread execute in program order (e.g., T_1:I_1, T_2:I_2, T_3:I_3, T_4:I_4, …..). However, instructions of different threads may get arbitrarily interleaved (Figure 12.6-(b)), while preserving the program order of the individual threads. For example, if you observe the instructions of thread 2 (T_2), you will see that they are in the program order for T_2.

Example 4:

Given the following threads and their execution history, what is the final value in memory location x? Assume that the execution of each instruction is atomic. Assume that Mem[x] = 0 initially.

Thread 1 (T1)

```
Time 0: R1 <- Mem[x]

Time 2: R1 <- R1+2

Time 4: Mem[x] <- R1
```

Thread 2 (T2)

```
Time 1: R2 <- Mem[x]

Time 3: R2 <- R2+1

Time 5: Mem[x] <- R2
```

Answer:
Each of the threads T1 and T2, load a memory location, add a value to it, and write it back. With the presence of data race and preemptive scheduling, unfortunately, x will contain the value written to it by the last store operation.

Since T2 is the last to complete its store operation, the final value in x is 1.

To summarize, non-determinism is at the core of the execution model for a parallel program. As an application programmer, it is important to understand and come to terms with this concept to be able to write correct parallel programs. Table 12.1 summarizes the execution models of sequential and parallel programs.

	Execution model
Sequential program	The program execution is deterministic, i.e., instructions execute in program order. The hardware implementation of the processor may reorder instructions for efficiency of pipelined execution so long as the appearance of program order is preserved despite such reordering.
Parallel program	The program execution is non-deterministic, i.e., instructions of each individual thread execute in program order. However, the instructions of the different threads of the same program may be arbitrarily interleaved.

Table 12.1: Execution Models of Sequential and Parallel Programs

12.2.4 Synchronization among threads

Let us understand how a programmer could expect a deterministic behavior for her program given that the execution model for a parallel program is non-deterministic.

As an analogy, let us watch some children at play shown in Figure 12.7. There are activities that they can do independently and concurrently (Figure 12.7-(a)) without stepping on each other. However, if they are sharing a toy, we tell them to take turns so that the other child gets a chance (Figure 12.7-(b)).

(a) Children playing independently[2] **(b) Children sharing a toy[3]**

Figure 12.7: Children playing in a shared "sandbox"

Similarly, if there are two threads, one a *producer* and the other a *consumer* then it is essential that the producer not modify the shared buffer while the consumer is reading it (see Figure 12.8). We refer to this requirement as *mutual exclusion*. The producer and consumer work concurrently except when either or both have to modify or inspect shared data structures. In that case, necessarily they have to execute sequentially to ensure data integrity.

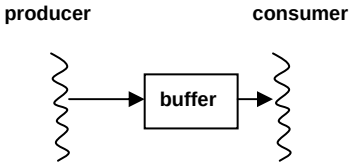


Figure 12.8: Shared buffer between threads

1. Mutual Exclusion Lock and Critical Section

The library provides a *mutual exclusion lock* for this purpose. A *lock* is a data abstraction that has the following semantics. A program can declare any number of these locks just as it declared variables of any other data type. The reader can see the analogy to a physical lock. Only one thread can hold a particular lock at a time. Once a thread *acquires* a lock, other threads cannot get that same lock until the first thread *releases* the lock. The following declaration creates a variable of type *lock*:

```
mutex_lock_type mylock;
```

The following calls allow a thread to acquire and release a particular lock:

```
thread_mutex_lock (mylock);
thread_mutex_unlock(mylock);
```

[2] Picture source: www.fotosearch.com/BNS238/cca021/
[3] Picture source: www.liveandlearn.com/toys/tetherball.html

Successful return from the first function above is an indication to the calling thread that it has successfully acquired the lock. If another thread currently holds the lock, then the calling thread *blocks* until the lock becomes free. In general, we define the *blocked* state of a thread as one in which the thread cannot proceed in its execution until some condition is satisfied. The second function shown above releases the named lock.

Sometimes, a thread may not want to block but go on to do other things if the lock is unavailable. The library provides a non-blocking version of the lock acquisition call:

```
{success, failure} <- thread_mutex_trylock (mylock);
```

This call returns a success or failure indication for the named lock acquisition request.

Example 5:
Show the code fragment to allow the producer and consumer threads in Figure 12.8 to access the buffer in a mutually exclusive manner to place an item and retrieve an item, respectively.

Answer:

```
item_type buffer;
mutex_lock_type buflock;
```

```
int producer()
{
  item_type item;

  /* code to produce item */
  ……

  thread_mutex_lock(buflock);
    buffer = item;
  thread_mutex_unlock(buflock);
  ……
  ……

}
```

```
int consumer()
{
  item_type item;

  ……
  ……

  thread_mutex_lock(buflock);
    item = buffer;
  thread_mutex_unlock(buflock);
  ……

  /* code to consume item */
}
```

Note: Only buffer and buflock are shared data structures. "item" is a local variable within each thread.

In example 5, the producer and consumer execute concurrently for the most part. When either the producer or the consumer executes code in their respective shaded boxes, what is the other thread doing? The answer to this question depends on where the other thread is in its execution. Let us say the producer is executing within its shaded box. Then the consumer is in one of two situations:

- Consumer is also actively executing if it is outside its shaded box
- If the consumer is trying to get into its shaded box, then it has to *wait* until the producer is out of its shaded box; similarly, the producer has to wait until the consumer is already in the shaded box

That is, the execution of the producer and the consumer in their respective shaded box is *mutually exclusive* of each other. Such code that is executed in a mutually exclusive manner is referred to as *critical section.* We define critical section as a region of the program in which the execution of the threads is serialized. That is, exactly one thread can be executing the code *inside* a critical section at a time. If multiple threads arrive at a critical section at the same time, one of them will succeed in entering and executing the code inside the critical section, while the others wait at the entry point. Many of us would have encountered a similar situation with an ATM machine, where we have to wait for our turn to withdraw cash or deposit a check.

As an example of a critical section, we present the code for implementing updates to a shared counter.

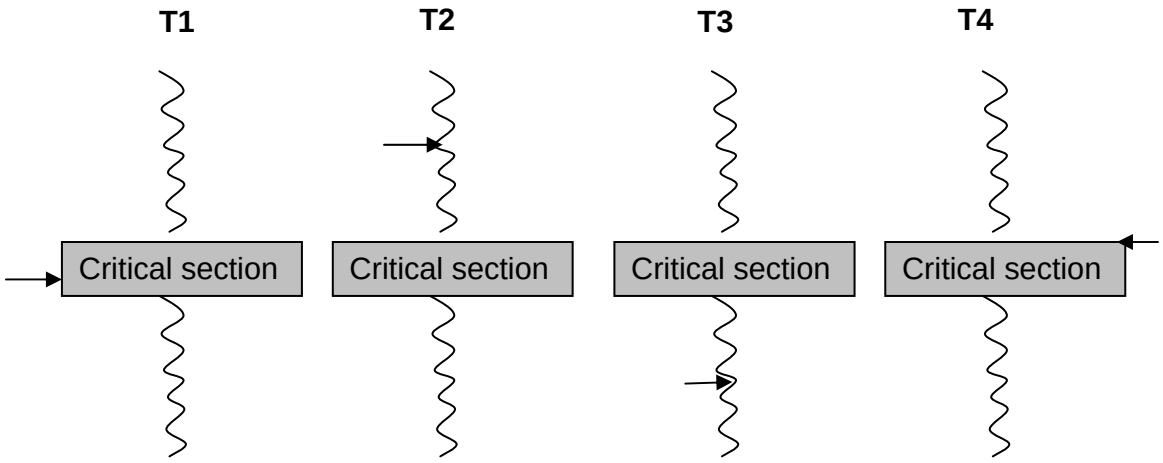
```
mutex_lock_type lock;
int counter; /* shared counter */

int increment_counter()
{
              /* critical section for
               * updating a shared counter
               */
              thread_mutex_lock(lock);
                    counter = counter + 1;
              thread_mutex_unlock(lock);
              return 0;
}
```

Any number of threads may call increment_counter simultaneously. Due to the mutually exclusive nature of thread_mutex_lock, exactly one thread can be inside the critical section updating the shared counter.

Example 6:

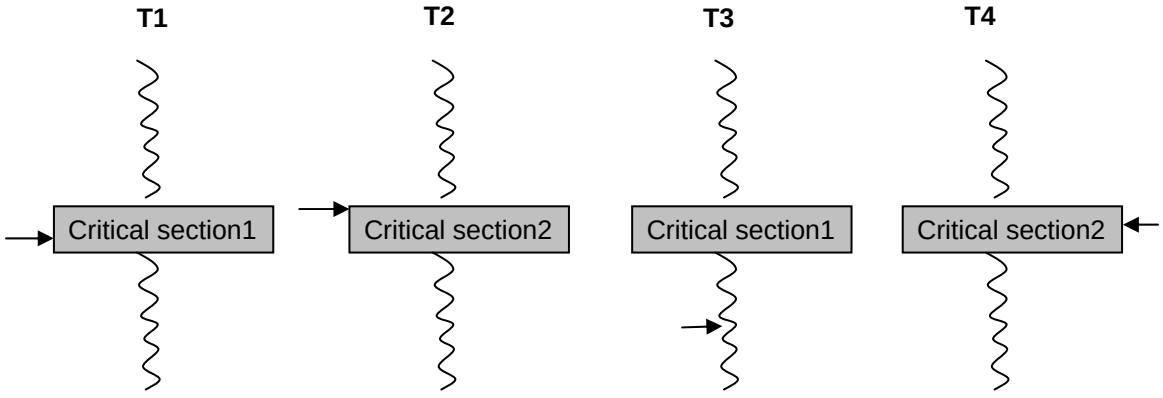
Given the points of execution of the threads (indicated by the arrows) in the figure below, state which ones are active and which ones are blocked and why. Assume that the critical

sections are mutually exclusive (i.e., they are governed by the same lock). T1-T4 are threads of the same process.



Answer:
T1 is **active** and executing code **inside** its **critical section**.
T2 is **active** and executing code **outside** its **critical section**.
T3 is **active** and executing code **outside** its **critical section**.
T4 is **blocked** and **waiting** to get into its **critical section**. (It will get in once the lock is released by T1).

Example 7:
Given the points of execution of the threads (indicated by the arrows) in the figure below, state which ones are active and which ones are blocked and why. Note that distinct locks govern each of critical sections 1 and 2, respectively.



Answer:
T1 is **active** and executing code **inside** its **critical section 1**.
T2 is **blocked** and **waiting** to get into its **critical section 2**. (It will get in once the lock is released by T4).
T3 is **active** and executing code **outside** its **critical section 1**.
T4 is **active** and executing code **inside** its **critical section 2**.

---

2. Rendezvous

A thread may want to wait another thread in the same process. The most common usage of such a mechanism is for a parent to wait for the children that it spawns. Consider for example, a thread is spawned to read a file from the disk while main has some other concurrent activity to perform. Once main completes this concurrent activity, it cannot proceed further until the spawned thread completes its file read. This is a good example where main may wait for its child to terminate which would be an indication that the file read is complete.

The library provides such a *rendezvous* mechanism through the function call:

```
thread_join (peer_thread_id);
```

The function blocks the caller until the named peer thread terminates. Upon the peer thread's termination, the calling thread resumes execution.

More formally, we define *rendezvous* as a meeting point between threads of the same program. A rendezvous requires a minimum of two threads but in the limit may include all the threads of a given program. Threads participating in a rendezvous continue with their respective executions once all the other threads have also arrived at the rendezvous point. Figure 12.9 shows an example of a rendezvous among threads. T1 is the first to arrive at the rendezvous point, awaits the arrival of the other two threads. T3 arrives next; finally, once T2 arrives the rendezvous is complete, and the three threads proceed with their respective executions. As should be evident, rendezvous is a convenient way for threads of a parallel program to coordinate their activities with respect to one another in the presence of the non-deterministic execution model. Further, any subset of the threads of a process may decide to rendezvous amongst them. The most general form of rendezvous mechanism is often referred to in the literature as *barrier synchronization*. This mechanism is extremely useful in parallel programming of scientific applications. *All* the threads of a given application that would like to participate in the rendezvous execute the barrier synchronization call. Once all the threads have arrived at the barrier, the threads are allowed to proceed with their respective executions.

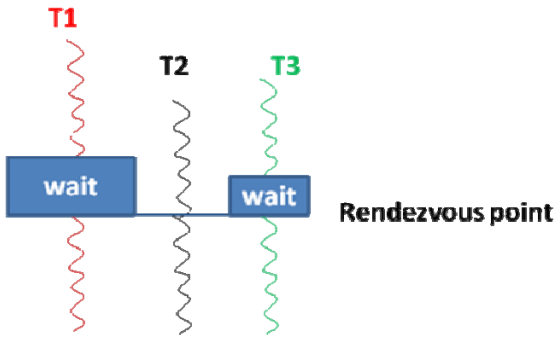


Figure 12.9: Rendezvous among threads

The thread_join call is a special case of the general rendezvous mechanism. It is a one-sided rendezvous. Only the thread that executes this call may experience any waiting (if

the peer thread has not already terminated). The peer thread is completely oblivious of the fact that another thread is waiting on it. Note that this call allows the calling thread to wait on the termination of exactly one thread. For example, if a main thread spawns a set of children threads, and would like to be alive until all of them have terminated, then it has to execute multiple thread_join calls one after the other for each of its children. In Section 12.2.8 (see Example 10), we will show a symmetric rendezvous between two threads using conditional variables.

There is one way in which the real-life analogy breaks down in the context of threads. A child usually outlives the parent in real life. Unfortunately, this is not necessarily true in the context of threads. In particular, recall that all the threads execute within an address space. As it turns out, not all threads are equal in status. There is a distinction between a parent thread and a child thread. In Figure 12.4 (Example 1), when the process is instantiated there is only one thread within the address space, namely, "main". Once "main" spawns the "digitizer" and "tracker" threads, the address space has three active threads: "main", "digitizer", and "tracker". What happens when "main" exits? This is the parent thread and is **synonymous** with the process itself. The usual semantics implemented by most operating systems is that when the parent thread in a process terminates, then the entire process terminates. However, note that if a child spawns its own children, immediate parent does not determine these children's life expectancy; the main thread that is synonymous with the process determines it. This is another reason why the thread_join call comes in handy, wherein the parent thread can wait on the children before exiting.

Example 8:

"main" spawns a top-level procedure "foo". Show how we can ensure that "main" does not terminate prematurely.

Answer:

```
int foo(int n)
{
  .....
  return 0;

}

int main()
{
        int f;
        thread_type child_tid;

        .....

        child_tid = thread_create (foo, &f);

        .....

        thread_join(child_tid);
}
```

Note: "main" by executing thread_join on the child_tid essentially waits for the child to be finished before itself terminating.

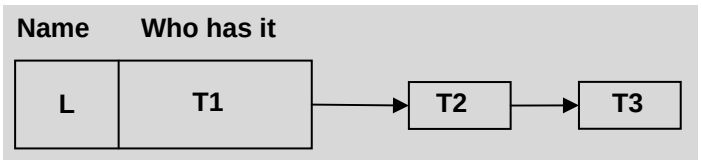
12.2.5 Internal representation of data types provided by the threads library

We mentioned that a thread that blocks when a lock it wants is currently in use by another thread. Let us understand the exact meaning of this statement. As should be evident by now, in contrast to data types (such as "int" and "float") supported by a programming language such as C, the threads library supports the data types we have been introducing in this chapter.

The **`thread_type`** and the **`mutex_lock_type`** are opaque data type (i.e., the user has no direct access to the internal representation of these data types). Internally, the threads library may have some accounting information associated with a variable of the **`thread_type`** data type. The **`mutex_lock_type`** data type is interesting and worth knowing more about from the programmer's perspective. Minimally, the internal representation for a variable of this data type will have two things:

- The thread (if any) that is currently holding the lock
- A queue of waiting requestors (if any) for this lock

Thus, if we have a lock variable **L**, currently a thread **T1** has it, and there are two other threads **T2** and **T3** waiting to get the lock then the internal representation in the threads library for this variable L will look as follows:



When **T1** releases the lock, **T2** gets the lock since that is the first thread in the waiting queue for this lock. Note that every lock variable has a distinct waiting queue associated with it. A thread can on exactly one waiting queue at any point of time.

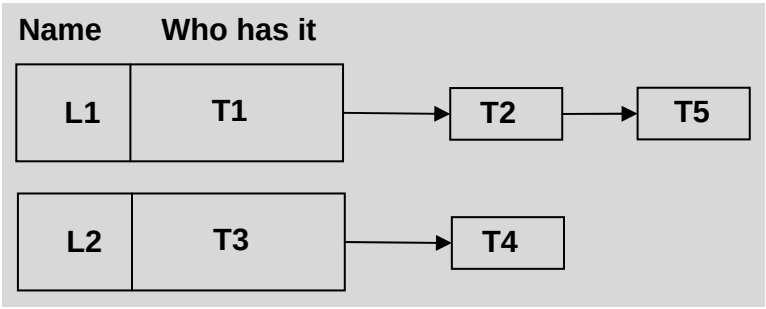
Example 9:

Assume that the following events happen in the order shown (T1-T5 are threads of the same process):

T1 executes **`thread_mutex_lock(L1);`**
T2 executes **`thread_mutex_lock(L1);`**
T3 executes **`thread_mutex_lock(L2);`**
T4 executes **`thread_mutex_lock(L2);`**
T5 executes **`thread_mutex_lock(L1);`**

Assuming there has been no other calls to the threads library prior to this, show the state of the internal queues in the threads library after the above five calls.

Answer:



12.2.6 Simple programming examples

1. Basic code with no synchronization

First, let us understand why we need the synchronization constructs. Consider the following sample program (#1) to show the interaction between the digitizer and the tracker threads of Figure 12.2. Suffice it to say at this point that we will progressively refine the sample program to ensure that it delivers the desired semantics for this application. For the benefit of advanced readers, the sample program #5 appearing a little later in the text delivers the desired semantics (see Section 12.2.9).

```
/*
 * Sample program #1:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];

digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail mod MAX] =
          dig_image;
      bufavail = bufavail - 1;
      tail = tail + 1;
    }
  } /* end loop */
}

tracker()
{
  image_type track_image;
  int head = 0;

  loop { /* begin loop */
    if (bufavail < MAX) {
      track_image =
          frame_buf[head mod MAX];
      bufavail = bufavail + 1;
      head = head + 1;
      analyze(track_image);
    }
  } /* end loop */
}
```

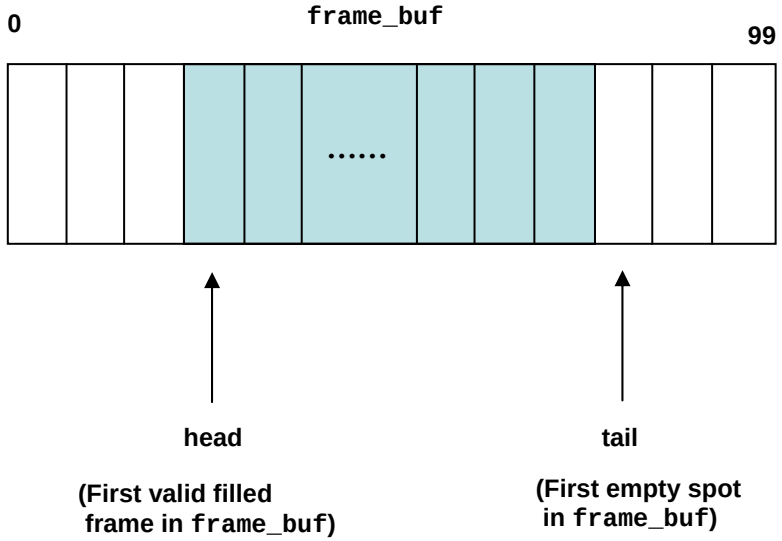


Figure 12.10: frame_buf implemented as a circular queue with head and tail pointers

In this sample program, **`bufavail`** and **`frame_buf`** are the shared data structures between the digitizer and the tracker threads. The sample program shows an implementation of the **`frame_buf`** as a circular queue with a **`head`** and a **`tail`** pointer,

with insertion at the tail and deletion at the head (see Figure 12.10; the shaded region contains valid items in **`frame_buf`**). The availability of space in the buffer is indicated by the **`bufavail`** variable.

The **`head`** and **`tail`** pointers themselves are local variables inside the digitizer and tracker, respectively. The digitizer code continuously loops grabbing an image from the camera, putting it in the frame buffer, advancing its **`tail`** pointer to point to the next open slot in **`frame_buf`**. Availability of space in the frame buffer (**`bufavail > 0`**) predicates this execution within the loop. Similarly, the tracker code continuously loops getting an image from the frame buffer (if one is available), advancing its **`head`** pointer to the next valid frame in **`frame_buf`**, and then analyzing the frame for items of interest. The two threads are independent of one another **except** for the interaction that happens in the **shaded boxes** in the two threads shown in sample program (#1). The code in the shaded boxes manipulates the shared variables, namely, **`frame_buf`** and **`bufavail`**.

2. Need for atomicity for a group of instructions

The problem with the above code fragment is that the digitizer and the tracker are concurrent threads and they could be reading and writing to the shared data structures *simultaneously*, if they are each executing on a distinct processor. Figure 12.11 captures this situation. Both the digitizer and the tracker are modifying **`bufavail`** at the same time.

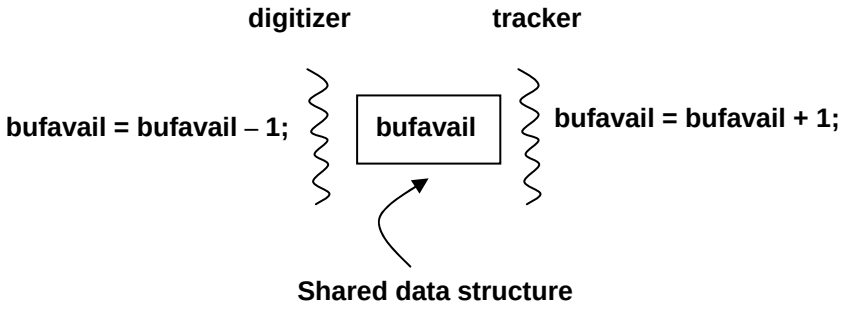


Figure 12.11: Problem with unsynchronized access to shared data

Let us drill down into this situation a little more. The statement

```
bufavail = bufavail – 1;          (1)
```

is implemented as a set of instructions on the processor (load **`bufavail`** into a processor register; do the decrement; store the register back into **`bufavail`**).

Similarly, the statement

```
bufavail = bufavail + 1;          (2)
```

is implemented as a set of instructions on the processor (load **`bufavail`** into a processor register; do the increment; store the register back into **`bufavail`**).

A correct execution of the program requires the *atomic* execution of each of the two statements (1 and 2). That is, either statement (1) executes and then (2), or vice versa. Interleaved execution of the instruction sequences for (1) and (2) could lead to erroneous and unanticipated behavior of the program. This is what we referred to as data race in Section 12.2.3. As we already mentioned in Section 12.2.3, such an interleaving could happen even on a uniprocessor due to context switches (see Example 4). The processor guarantees atomicity of an instruction at the level of the instruction-set architecture. However, the system software (namely, the operating system) has to guarantee atomicity for a group of instructions.

Therefore, to ensure atomicity we need to encapsulate accesses to shared data structures within *critical sections* to ensure mutually exclusive execution. However, we have to be careful in deciding how and when to use synchronization constructs. Indiscriminate use of these constructs, while ensuring atomicity could lead to restricting concurrency and worse yet introduce incorrect program behavior.

3. Code refinement with coarse grain critical sections

We will now proceed to use the synchronization constructs **`thread_mutex_lock`** and **`thread_mutex_unlock`** in the sample program to achieve the desired mutual exclusion.

Sample program #2 is another attempt at writing a threaded sample program for the same example in Figure 12.2. This program illustrates the use of mutual exclusion lock. The difference from the earlier one (sample program #1) is the addition of the synchronization constructs inside the shaded boxes in sample program #2. In each of digitizer and tracker, the code between lock and unlock is the work done by the respective threads to access shared data structures. Note that the use of synchronization constructs ensures atomicity of the entire code block between lock and unlock calls for the digitizer and tracker, respectively. This program is "correct" in terms of the desired semantics but has a serious performance problem that we elaborate next.

```
/*
 * Sample program #2:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
    thread_mutex_lock(buflock);
      if (bufavail > 0) {
        grab(dig_image);
        frame_buf[tail mod MAX] =
          dig_image;
        tail = tail + 1;
        bufavail = bufavail - 1;
      }
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

```
tracker()
(
  image_type track_image;
  int head = 0;

  loop { /* begin loop */
    thread_mutex_lock(buflock);
      if (bufavail < MAX) {
        track_image =
          frame_buf[head mod MAX];
        head = head + 1;
        bufavail = bufavail + 1;
        analyze(track_image);
      }
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

---

4. Code refinement with fine grain critical sections

A close inspection of sample program #2 will reveal that it has no synchronization problem but there is no concurrency in the execution of the digitizer and the tracker. Let us analyze what needs mutual exclusion in this sample program. There is no need for mutual exclusion for either grabbing the image by the digitizer or for analyzing the image by the tracker. Similarly, once the threads have ascertained the validity of operating on **`frame_buf`** by checking **`bufavail`**, insertion or deletion of the item can proceed concurrently. That is, although **`frame_buf`** is a shared data structure, the way it is used in the program obviates the need for serializing access to it. Therefore, we modify the program as shown below to increase the amount of concurrency between the tracker and the digitizer. We limit the mutual exclusion to the checks and modifications done to **`bufavail`**. Unfortunately, the resulting code has a serious problem that we elaborate next.

```
/*
 * Sample program #3:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
    grab(dig_image);
    thread_mutex_lock(buflock);
      while (bufavail == 0) do nothing;
    thread_mutex_unlock(buflock);
    frame_buf[tail mod MAX] =
          dig_image;
    tail = tail + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail - 1;
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop { /* begin loop */
    thread_mutex_lock(buflock);
      while (bufavail == MAX) do nothing;
    thread_mutex_unlock(buflock);
    track_image =
          frame_buf[head mod MAX];
    head = head + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail + 1;
    thread_mutex_unlock(buflock);
    analyze(track_image);
  } /* end loop */
}
```

12.2.7 Deadlocks and livelocks

Let us dissect sample program #3 to see if it has any problems. Consider the **`while`** statement in the digitizer code. It is checking **`bufavail`** for an empty spot in the **`frame_buf`**. Let us assume that **`frame_buf`** is full. In this case, the digitizer is continuously executing the **`while`** statement waiting for space to free up in **`frame_buf`**. The tracker has to make space in the **`frame_buf`** by removing an image and incrementing **`bufavail`**. However, the digitizer has **`buflock`** and hence the tracker is stuck trying to acquire **`buflock`**. A similar situation arises when **`frame_buf`** is empty (the **`while`** statement in the tracker code).

The problem we have described above, called *deadlock* is the bane of all concurrent programs. Deadlock is a situation wherein a thread is waiting for an event that will never happen. For example, the digitizer is waiting for **`bufavail`** to become non-zero in the **`while`** statement, but that event will not happen since the tracker cannot get the lock. The situation captured above is a special case of deadlock, often referred to as a *livelock*. A thread is involved in a deadlock may be waiting actively or passively. Livelock is the situation wherein a thread is actively checking for an event that will never happen. Let us say in this case, the digitizer is holding the **`buflock`** and checking for **`bufavail`** to become non-zero. This is a livelock since it is wasting processor resource to check for an event that will never happen. On the other hand, the tracker is waiting for the lock to be released by the digitizer. Tracker's waiting is passive since it is blocked in the operating system for the lock release. Regardless of whether the waiting is passive or active, the threads involved in a deadlock are stuck forever. It should be evident to the reader that deadlocks and livelocks are yet another manifestation of the basic non-deterministic nature of a parallel program execution.

One could make a case that the **`while`** statements in the above code do not need mutual exclusion since they are only inspecting buffer availability. In fact, removing mutual exclusion around the **`while`** statements will remove the *deadlock* problem. Sample program #4 takes this approach. The difference from sample program #3 is that the lock around the while statement has been removed for both the digitizer and the tracker.

```
/*
 * Sample program #4:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

 digitizer()
 {
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
   grab(dig_image);
   while (bufavail == 0) do nothing;
   frame_buf[tail mod MAX] =
         dig_image;
   tail = tail + 1;
   thread_mutex_lock(buflock);
    bufavail = bufavail - 1;
   thread_mutex_unlock(buflock);
  } /* end loop */
 }
```

```
tracker()
{
 image_type track_image;
 int head = 0;

 loop { /* begin loop */
  while (bufavail == MAX) do nothing;
  track_image =
        frame_buf[head mod MAX];
  head = head + 1;
  thread_mutex_lock(buflock);
   bufavail = bufavail + 1;
  thread_mutex_unlock(buflock);
  analyze(track_image);
 } /* end loop */
}
```

This solution is correct and has concurrency as well for the digitizer and the tracker. However, the solution is grossly inefficient due to the nature of waiting. We refer to the kind of waiting that either the digitizer or the tracker does in the while statement as *busy waiting*. The processor is busy doing nothing. This is inefficient since the processor could have been doing something more useful for some other thread or process.

12.2.8 Condition variables

Ideally, we would like the system to recognize that the condition that the digitizer is waiting for (**`bufavail > 0`**) is not satisfied, and therefore release the lock on behalf of the digitizer, and reschedule it later on when the condition is satisfied.

This is exactly the semantics of another data abstraction commonly provided by the library, called a *condition variable*.

The following declaration creates a variable of type of *condition variable*:

```
cond_var_type buf_not_empty;
```

The library provides calls to allow threads to *wait* and *signal* one another using these condition variables:

```
thread_cond_wait(buf_not_empty, buflock);
thread_cond_signal(buf_not_empty);
```

The first call allows a thread (tracker in our example) to wait on a condition variable. *Waiting* on a condition variable amounts to the library de-scheduling the thread that made that call. Notice that the second argument to this call is a mutual exclusion lock variable. Implicitly, the library performs **`unlock`** on the named lock variable before de-scheduling the calling thread. The second call *signals* any thread that may be waiting on the named condition variable. A *signal* as the name suggests is an indication to the waiting thread that it may resume execution. The library knows the specific lock variable associated with the *wait* call. Therefore, the library performs an implicit **`lock`** on that variable prior to scheduling the waiting thread. Of course, the library treats as a NOP any signal on a condition variable for which there is no waiting thread. Multiple threads may wait on the same condition variable. The library picks one of them (usually First-Come-First-Served) among the waiting threads to deliver the signal. One has to be very careful in using wait and signal for synchronization among threads. Figure 12.12 (a) shows a correct use of the primitives. Figure 12.12 (b) shows an incorrect use, creating a situation wherein T1 starts waiting after T2 has signaled, leading to a deadlock.

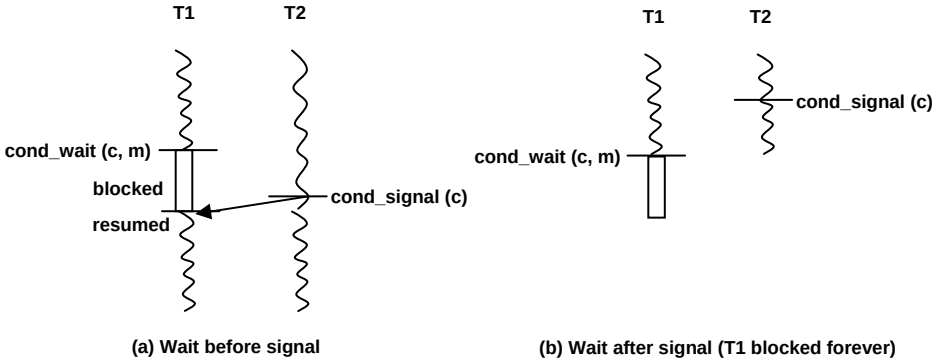


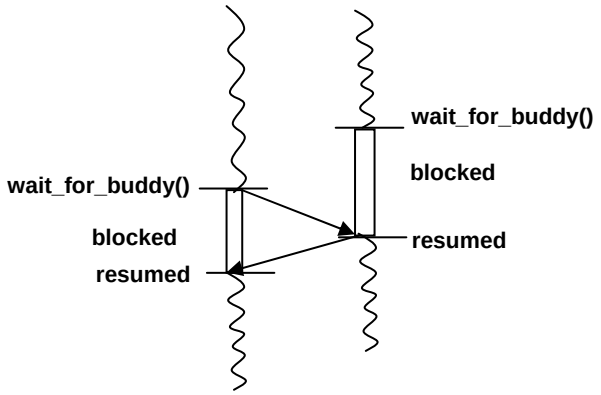
Figure 12.12: Wait and Signal with condition variable: "c" is a condition variable, and "m" is a mutex lock with which is "c" is associated through the cond_wait call

Figure 12.12(b) illustrates the situation that a signal sent prematurely will land a peer in a deadlock. The following example shows how one can devise a rendezvous mechanisms between two peers independent of the order of arrival.

Example 10:

Write a function **wait_for_buddy()** to be used by EXACTLY 2 threads to rendezvous with each other as shown in the figure below. The order of arrival of the two thread

should be immaterial. Note that this is a general example of accomplishing a rendezvous (described earlier in Section 12.2.4) among independent threads of the same process.



Answer:

The solution uses a boolean (buddy_waiting), a mutex lock (mtx), and a condition variable (cond). The basic idea is the following:

- Whichever thread arrives first (the "if" section of the code below), sets the buddy_waiting flag to be true and waits
- The second arriving thread (the "else" section of the code below), sets the buddy_waiting to be false, signals the first thread, and waits
- The first arriving thread is unblocked from its conditional wait, signals the second thread, unlocks the mutex and leaves the procedure
- The second arriving thread is unblocked from its conditional wait, unlocks the mutex and leaves the procedure
- It is important to observe the wait-signal ordering in the "if" and "else" code blocks; not following the order shown will result in deadlock.

```
boolean buddy_waiting = FALSE;
mutex_lock_type mtx; /* assume this has been initialized properly */
cond_var_type cond;  /* assume this has been initialized properly */

wait_for_buddy()
{
  /* both buddies execute the lock statement */
  thread_mutex_lock(mtx);

  if (buddy_waiting == FALSE) {
    /* first arriving thread executes this code block */
    buddy_waiting = TRUE;

    /* the following order is important */
    /* the first arriving thread will execute a wait statement */
    thread_cond_wait (cond, mtx);

    /* the first thread wakes up due to the signal from the second
     * thread, and immediately signals the second arriving thread
     */
    thread_cond_signal(cond);
  }
```

```
  else {
    /* second arriving thread executes this code block */
    buddy_waiting = FALSE;

    /* the following order is important */
    /* signal the first arriving thread and then execute a wait
     * statement awaiting a corresponding signal from the first thread
     */
    thread_cond_signal (cond);
    thread_cond_wait (cond, mtx);
  }

  /* both buddies execute the unlock statement */
  thread_mutex_unlock (mtx);
}
```

12.2.8.1 Internal representation of the condition variable data type

It is instructive from a programming standpoint to understand the internal representation within the threads library of the **`cond_var_type`** data type. Minimally, a variable of this data type has:

- a queue of threads (if any) waiting for signal on this variable
- for each thread waiting for a signal, the associated mutex lock

A thread that calls **`thread_cond_wait`** names a mutex lock. The threads library unlocks this lock on behalf of this thread before placing it on the waiting queue. Similarly, upon receiving a signal on this condition variable, when a thread is unblocked from the waiting queue, it is the responsibility of the threads library to re-acquire the lock on behalf of the thread before resuming the thread. This is the reason why the threads library remembers the lock associated with a thread on the waiting queue.

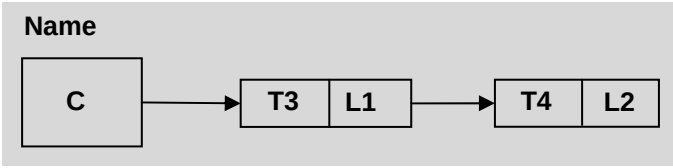
Thus, for instance if two threads T3 and T4 execute conditional wait calls on a condition variable C; let T3's call be

thread_cond_wait (C, L1)

and let T4's call be

thread_cond_wait (C, L2)

The internal representation of C after the above two calls will look as follows:



Note that it is not necessary that all wait calls on a given condition variable name the same lock variable.

Example 11:

Assume that the following events happen in the order shown (T1-T7 are threads of the same process):

T1 executes **`thread_mutex_lock(L1);`**
T2 executes **`thread_cond_wait(C1, L1);`**
T3 executes **`thread_mutex_lock(L2);`**
T4 executes **`thread_cond_wait(C2, L2);`**
T5 executes **`thread_cond_wait(C1, L2);`**

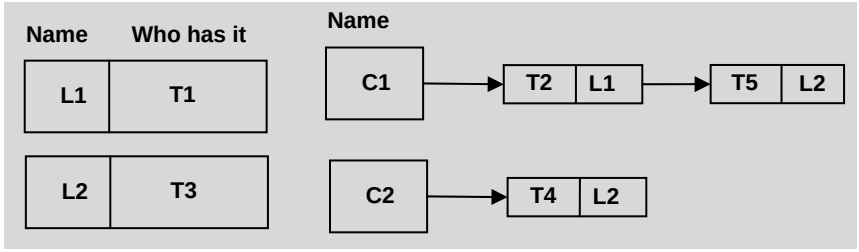
(a) Assuming there has been no other calls to the threads library prior to this, show the state of the internal queues in the threads library after the above five calls.
(b) Subsequently the following event happens:

T6 executes **`thread_cond_signal(C1);`**
T7 executes **`thread_cond_signal(C2);`**

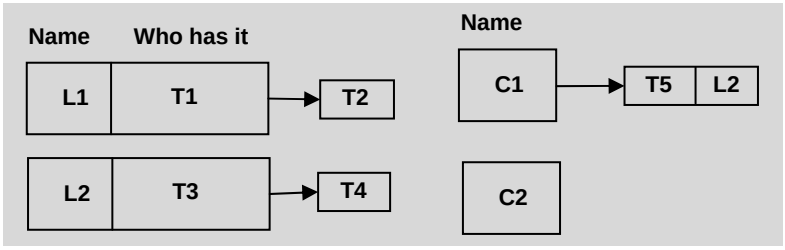
Show the state of the internal queues in the threads library after these two calls.

Answer:

(a)



(b)



The library moves T2 to the waiting queue of L1, and T4 to the waiting queue of L2 upon receiving the signals on C1 and C2, respectively.

12.2.9 A complete solution for the video processing example

Now we will return to our original video processing example of Figure 12.2. Shown below is a program sample using wait and signal semantics for the same example. Note that each thread waits after checking a condition that is not true currently; the other thread enables this condition eventually ensuring that there will not be a deadlock. Note also that each thread performs the signaling while holding the mutual exclusion lock. While this is strictly not necessary, nevertheless, it is a good programming practice, and results in less erroneous parallel programs.

```
/*
 * Sample program #5:  This solution delivers the expected
 *                     semantics for the video processing
```

```
 *                      pipeline shown in Figure 12.2, both
 *                      in terms of performance and
 *                      correctness for a single digitizer
 *                      feeding images to a single tracker.
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
cond_var_type buf_not_full;
cond_var_type buf_not_empty;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop { /* begin loop */
    grab(dig_image);
    thread_mutex_lock(buflock);
      if (bufavail == 0)
          thread_cond_wait(buf_not_full,
              buflock);
    thread_mutex_unlock(buflock);
    frame_buf[tail mod MAX] = dig_image;
    tail = tail + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail - 1;
      thread_cond_signal(buf_not_empty);
    thread_mutex_unlock(buflock);
  } /* end loop */
}
```

(1)

(2)

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop { /* begin loop */
    thread_mutex_lock(buflock);
      if (bufavail == MAX)
          thread_cond_wait(buf_not_empty,
              buflock);
    thread_mutex_unlock(buflock);
    track_image = frame_buf[head mod MAX];
    head = head + 1;
    thread_mutex_lock(buflock);
      bufavail = bufavail + 1;
      thread_cond_signal(buf_not_full);
    thread_mutex_unlock(buflock);
    analyze(track_image);
  } /* end loop */
}
```

(3)

(4)

The key point to note in this program sample is the *invariant* maintained by the library on behalf of each thread. An invariant is some indisputable truth about the state of the program. At the point of making the **`thread_cond_wait`** call, the invariant is that the calling thread is holding a lock. The library implicitly releases the lock on behalf of the calling thread. When the thread resumes it is necessary to re-establish the invariant. The library re-establishes the invariant on behalf of the blocked thread prior to resumption by implicitly re-acquiring the lock.

12.2.9.1 Discussion of the solution

1. Concurrency

Let us analyze the solution presented in program sample #5 and convince ourselves that there is no lack of concurrency.

- First, notice that code blocks (1) and (3) hold the lock only for checking the value in **`bufavail`**. If the checking results in a favorable outcome, the release the lock and

go on to either putting an image or getting an image, respectively. What if the checking results in an unfavorable outcome? In that case, the code blocks execute a conditional wait statement on **`buf_not_full`** and **`buf_not_empty`**, respectively. In either case, the library releases the associated lock immediately.

- Second, notice that code blocks (2) and (4) hold the lock only for updating the **`bufavail`** variable, and signaling to unblock the other thread (if it is waiting).

Given the above two points, there is **no lack of concurrency** since the lock is never held for any extended period by either thread.

Example 12:

Assume that the digitizer is in code block (2) and is about to signal on the **`buf_not_empty`** condition variable in Sample program #5.

Explain if the following statement is **True** or **False** with justification:

The tracker is guaranteed to be waiting for a signal on `buf_not_empty` inside code block (3).

Answer:

False. Tracker could be waiting but not always. Note that the signaling in code block (2) is unconditional. Therefore, we do not know what the value of **`bufavail`** is. The only way for the tracker to be blocked inside code block (2) is if **`bufavail = MAX.`** We know it is non-zero since the digitizer is able to put a frame in, but we do not know that it is = MAX.

---

2. Absence of deadlock

Next, let us convince ourselves that the solution is correct and does not result in deadlock. First, we will informally show that at any point of time **both** the threads do not block leading to a deadlock.

- Let the digitizer be waiting inside code block (1) for a signal. Given this, we will show that the tracker will not also block leading to a deadlock. Since the digitizer is blocked, we know the following to be true:
  - **`bufavail = 0`**
  - digitizer is blocked waiting for a signal on **`buf_not_full`**
  - **`buflock`** has been implicitly released by the thread library on behalf of the digitizer.

  There are three possible places for the tracker to block:
  - Entry to code block (3): Since the digitizer does not hold **`buflock`** tracker will not block at the entry point.
  - Entry to code block (4): Since the digitizer does not hold **`buflock`** tracker will not block at the entry point.
  - Conditional wait statement inside code block (3): The digitizer is blocked waiting for a signal inside code block (1). Therefore, **`bufavail`** = 0; hence, the "if" statement inside code block (3) will return a favorable result and the tracker is guaranteed not to block.

- With a similar line of argument as above, we can establish that if the tracker is waiting inside code block (3) for a signal, the digitizer will not also block leading to a deadlock.

Next, we will show that if one thread is blocked it will eventually unblock thanks to the other.

- Let us say that the digitizer is blocked waiting for a signal inside code block (1). As we argued earlier, the tracker will be able to execute its code without blocking. Therefore, eventually it will get to the signal statement inside code block (4). Upon receiving this signal, the digitizer waits for re-acquiring the lock (currently held by the tracker inside code block (4)). Note that the thread library does this lock re-acquisition implicitly for the digitizer. Tracker leaves code block (4) releasing the lock; digitizer re-acquires the lock and moves out of code block (1) releasing the lock on its way out.
- With a similar line of argument as above, we can establish that if the tracker is waiting inside code block (3) for a signal, the digitizer will issue the signal that will unblock the tracker.

Thus, we have shown informally that the solution is correct and does not suffer from lack of concurrency.

12.2.10 Rechecking the predicate

The program sample #5 works correctly for the specific example where there is one tracker and one digitizer. However, in general, programming with condition variables needs more care to avoid synchronization errors. Consider the program fragment shown below for using a shared resource. Any number of threads can execute the procedure **`use_shared_resource`**.

```
/*
 * Sample program #6:
 */
enum state_t {BUSY, NOT_BUSY} res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

/* helper procedure for acquiring the resource */
acquire_shared_resource()
{
  thread_mutex_lock(cs_mutex);                          T3 is here
    if (res_state == BUSY)
      thread_cond_wait (res_not_busy, cs_mutex);        T2 is here
    res_state = BUSY;
  thread_mutex_unlock(cs_mutex);
}
```

```
/* helper procedure for releasing the resource */
release_shared_resource()
{
  thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;                          T1 is here
    thread_cond_signal(res_not_busy);
  thread_mutex_unlock(cs_mutex);
}

/* top level procedure called by all the threads */
use_shared_resource()
{
  acquire_shared_resouce();
    resource_specific_function();
  release_shared_resource();
}
```

As shown above,

- T1 has just finished using the resource and has set the **res_state** as **NOT_BUSY**;
- T2 is in conditional wait; and
- T3 is waiting to acquire the **cs_mutex**.

Figure 12.13 shows the state of the waiting queues in the library for **cs_mutex** and **res_not_busy**:

- T2 is in the **res_not_busy** queue while T3 is in the **cs_mutex** queue, respectively (Figure 12.13 (a)).
- T1 signals, resulting in the library moving T2 to the **cs_mutex** queue since the library has to re-acquire the **cs_mutex** before resuming T2 (Figure 12.13 (b)).

When T1 releases **cs_mutex**:

- The lock is given to T3, the first thread in the waiting queue for **cs_mutex**.
- T3 tests **res_state** to be **NOT_BUSY** and releases **cs_mutex**, and goes on to use the resource.
- T2 resumes from the **thread_cond_wait** (since **cs_mutex** is now available for it), releases **cs_mutex** and goes on to use the resource as well.

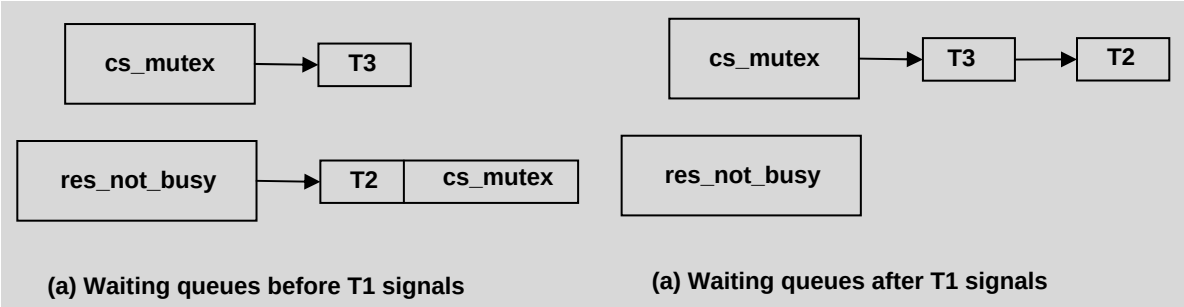


Figure 12.13: State of the waiting queues

Now we have violated the mutual exclusion condition for using the shared resource. Let us investigate what led to this situation. T1 enables the condition that T2 is waiting on prior to signaling, but T3 negates it before T2 resumes execution. Therefore, re-checking the predicate (i.e., the condition that needs to be satisfied in the program) upon resumption is a defensive coding technique to avoid such synchronization errors.

The program fragment shown below fixes the above problem by changing the **`if`** statement associated with the **`thread_cond_wait`** to a **`while`** statement. This ensures that a thread tests the predicate again upon resumption and blocks again on **`thread_cond_wait`** call if necessary.

```
/*
 * Sample program #7:
 */
enum state_t {BUSY, NOT_BUSY} res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

acquire_shared_resource()
{
  thread_mutex_lock(cs_mutex);                        T3 is here
   while (res_state == BUSY)
      thread_cond_wait (res_not_busy, cs_mutex);      T2 is here
    res_state = BUSY;
  thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
  thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;                             T1 is here
    thread_cond_signal(res_not_buys);
  thread_mutex_unlock(cs_mutex);
}
```

```
use_shared_resource()
{
  acquire_shared_resouce();
    resource_specific_function();
  release_shared_resource();
}
```

Example 13:
Rewrite sample program #5 to allow multiple digitizers and multiple trackers to work together. This is left as an exercise for the reader.
[Hint: Rechecking the predicate after awakening from a conditional wait becomes important. In addition, now the instances of digitizers share the head pointer and instances of trackers share the tail pointer. Therefore, modifications of these pointers require mutual exclusion among the digitizer instances and tracker instances, respectively. To ensure concurrency and reduce unnecessary contention among the threads, use distinct locks to provide mutual exclusion for the head and tail pointers, respectively.]

12.3 Summary of thread function calls and threaded programming concepts

Let us summarize the basic function calls we proposed in the earlier sections for programming multithreaded applications. Note that this is just an illustrative set of basic calls and not meant to be comprehensive. In Section 12.6, we give a comprehensive set of function calls proposed as an IEEE POSIX[4] standard thread library.

- **`thread_create (top-level procedure, args);`**
  creates a new thread that starts execution in the top-level procedure with the supplied args as actual parameters for the formal parameters specified in the procedure prototype.

- **`thread_terminate (tid);`**
  terminates the thread with id given by **`tid`**.

- **`thread_mutex_lock (mylock);`**
  when the thread returns it has **`mylock`**; the calling thread blocks if the lock is in use currently by some other thread.

- **`thread_mutex_trylock (mylock);`**
  call does not block the calling thread; instead it returns **`success`** if the thread gets **`mylock`**; **`failure`** if the lock is in use currently by some other thread.

- **`thread_mutex_unlock(mylock);`**
  if the calling thread currently has **`mylock`** it is released; error otherwise.

[4] IEEE is an international society and stands for **Institute of Electrical and Electronics Engineers, Inc.**, and POSIX stands for **Portable Operating System Interface (POSIX®).**

- **`thread_join (peer_thread_tid);`**
  the calling thread blocks until thread given by **`peer_thread_tid`** terminates.

- **`thread_cond_wait(buf_not_empty, buflock);`**
  calling thread blocks on the condition variable **`buf_not_empty`**; the library implicitly releases the lock **`buflock`**; error if the lock is not currently held by the calling thread.

- **`thread_cond_signal(buf_not_empty);`**
  a thread (if any) waiting on the condition variable **`buf_not_empty`** is woken up; the awakened thread either is ready for execution if the lock associated with it (in the wait call) is currently available; if not, the thread is moved from the queue for the condition variable to the appropriate lock queue.

Concept	Definition and/or Use
Top level procedure	The starting point for execution of a thread of a parallel program
Program order	This is the execution model for a sequential program that combines the textual order of the program together with the program logic (conditional statements, loops, procedures, etc.) enforced by the intended semantics of the programmer.
Execution model for a parallel program	The execution model for a parallel program preserves the program order for individual threads, but allows arbitrary interleaving of the individual instructions of the different threads.
Deterministic execution	Every run of a given program results in the same output for a given set of inputs. The execution model presented to a sequential program has this property.
Non-deterministic execution	Different runs of the same program for the same set of inputs could result in different outputs. The execution model presented to a parallel program has this property.
Data race	Multiple threads of the same program are simultaneously accessing a shared variable without any synchronization, with at least one of the accesses being a write to the variable.
Mutual exclusion	Signifies a requirement to ensure that threads of the same program execute serially (i.e., not concurrently). This requirement needs to be satisfied in order to avoid data races in a parallel program.
Critical section	A region of a program wherein the activities of the threads are serialized to ensure mutual exclusion.
Blocked	Signifies the state of a thread in which it is simply waiting in a queue for some condition to be satisfied to make it runnable.
Busy waiting	Signifies the state of a thread in which it is continuously checking for a condition to be satisfied before it can proceed further in its execution.
Deadlock	One or more threads of the same program are blocked awaiting a condition that will never be satisfied.
Livelock	One or more threads of the same program are busy-waiting for a condition that will never be satisfied.
Rendezvous	Multiple threads of a parallel program use this mechanism to coordinate their activities. The most general kind of rendezvous is barrier synchronization. A special case of rendezvous is the thread_join call.

Table 12.2: Summary of Concepts Relating to Threads

Table 12.2 summarizes the important concepts we have introduced in the context of multithreaded programming as a quick reference.

12.4 Points to remember in programming with threads

There are several points to take care of while programming with threads:

1. It is important to design the data structures in such a way to enhance concurrency among threads.

2. It is important to minimize both the granularity of data structures that need to be locked in a mutually exclusive manner as well as the duration for which such locks need to be held.
3. It is important to avoid busy waiting since it is wasteful of the processor resource.
4. It is important to carefully understand the invariant that is true for each critical section in the program and ensure that this invariant is preserved while in the critical section.
5. It is important to make the critical section code as simple and concise as possible to enable manual verification that there are no deadlocks or livelocks.

12.5 Using threads as software structuring abstraction

Figure 12.14 shows some of the models for using threads as a structuring mechanism for system software.

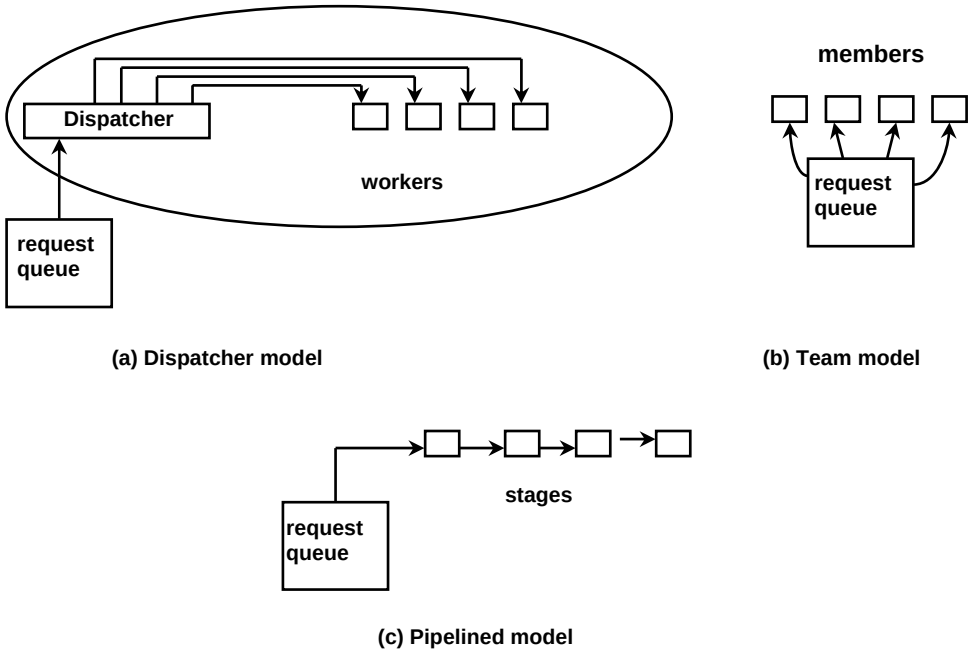


Figure 12.14: Structuring servers using threads

Software entities such as file servers, mail servers, and web servers typically execute on multiprocessors. Figure 12.14 (a) shows a dispatcher model for such servers. A *dispatcher* thread dispatches requests as they come in to one of a pool of *worker* threads. Upon completion of the request, the worker thread returns to the free pool. The *request queue* serves to smooth the traffic when the burst of requests exceeds server capacity. The dispatcher serves as a workload manager as well, shrinking and growing the number of worker threads to meet the demand. Figure 12.14 (b) shows a *team* model in which all the members of the team directly access the request queue for work. Figure 12.14 (c) shows a pipelined model that is more appropriate for continuous applications such as video surveillance that we discussed earlier in the chapter. Each stage of the pipeline handles a specific task (e.g. digitizer, tracker, etc.).

Client programs benefit from multithreading as well. Threads increase modularity and simplicity of client programs. For example, a client program may use threads to deal with exceptions, handling signals, and for terminal input/output.

12.6 POSIX pthreads library calls summary

IEEE has standardized the Application Programming Interface (API) for threads with the POSIX *pthreads* library. Every flavor of Unix operating system implements this standard. Program portability is the main intent of such standardization efforts. Microsoft Windows does not follow the POSIX standard for its thread library[5]. Summarized below are some of the most commonly used pthread library calls with a brief description of their purpose. For more information, see appropriate documentation sources (e.g. man pages on any flavor of Unix systems[6]).

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutex-attr_t *mutexattr);
```

Arguments

mutex	address of the mutex variable to be initialized
mutexattr	address of the attribute variable used to initialize the mutex. see pthread_mutexattr_init for more information

Semantics

Each mutex variable must be declared (pthread_mutex_t) and initialized.

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

Arguments

cond	address of the condition variable being initialized
cond_attr	address of the attribute variable used to initialize the condition variable. Not used in Linux.

Semantics

Each condition variable must be declared (pthread_cond_t) and initialized.

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Arguments

thread	Address of the thread identifier (tid)
attr	Address of the attributes to be applied to the new thread
start_routine	Function that new thread will start executing
arg	address of first argument passed to start_routine

Semantics

This function will create a new thread; establish the starting address (passed as the name of a function) and pass arguments to be passed to the

[5] Though Microsoft does not directly support the POSIX standard, the thread library available in WIN32 platforms for developing multithreaded applications in C have, for the most part, a semantically equivalent function call for each of the POSIX standard thread calls presented in this section.

[6] For example see: http://www.penguin-soft.com/penguin/manpages.jsp

function where the thread starts. The thread id (*tid*) of the newly created thread will be placed in the location pointed to by thread.

```
int pthread_kill(pthread_t thread,
                 int signo);
```

Arguments

thread	thread id of the thread to which the signal will be sent
signo	signal number to send to thread

Semantics

Used to send a signal to a thread whose *tid* is known.

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

Arguments

th	tid of thread to wait for
thread_return	If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return. The return value of th is either the argument it gave to pthread_exit(3), or PTHREAD_CANCELED if th was cancelled.

Semantics

pthread_join suspends the execution of the calling thread until the thread identified by th terminates, either by calling pthread_exit(3) or by being cancelled.

```
pthread_t pthread_self(void);
```

Arguments

None

Semantics

pthread_self returns the thread identifier for the calling thread.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Arguments

mutex	address of mutex variable to be locked

Semantics

Waits until the specified mutex is unlocked and then locks it and returns.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Arguments

mutex	address of mutex variable to be unlocked

Semantics

Unlocks the specified mutex if, the caller is the thread that locked the mutex.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Arguments

cond address of condition variable to wait upon.
mutex address of mutex variable associated with cond

Semantics

pthread_cond_wait atomically unlocks the mutex (as per pthread_unlock_mutex) and waits for the condition variable cond to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to pthread_cond_wait. Before returning to the calling thread, pthread_cond_wait re-acquires mutex (as per pthread_lock_mutex).

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Arguments

cond address of condition variable

Semantics

Wakes up one thread waiting for a signal on the specified condition variable.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Arguments

cond address of condition variable

Semantics

Variant of the previous call, wherein the signal wakes up *all* threads waiting for a signal on the specified condition variable.

```
void pthread_exit(void *retval);
```

Arguments

retval address of the return value of the thread

Semantics

Terminates the execution of the calling thread.

12.7 OS support for threads

In operating systems such as MS-DOS, an early bare bones OS for the PC, there is no separation between the user program and the kernel (Figure 12.15). Thus, the line between the user program and the kernel is imaginary and hence it costs very little (in terms of time, equivalent to a procedure call) to go between user and kernel spaces. The downside to this structure is the fact that there is no memory protection among user programs, and an errant or malicious program can easily corrupt the memory space of the kernel.

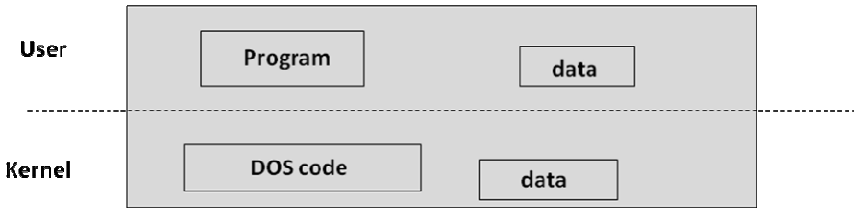


Figure 12.15: MS-DOS user-kernel boundary. MS-DOS was an early OS for the PC. The shading indicates that there is no enforced separation between user and kernel

Modern operating systems such as MS-Windows[xp], Linux, Mac OS X, and Unix provide true memory protection through virtual memory mechanisms we have discussed in earlier chapters. Figure 12.16 shows the memory spaces of user processes and the kernel in such operating systems.

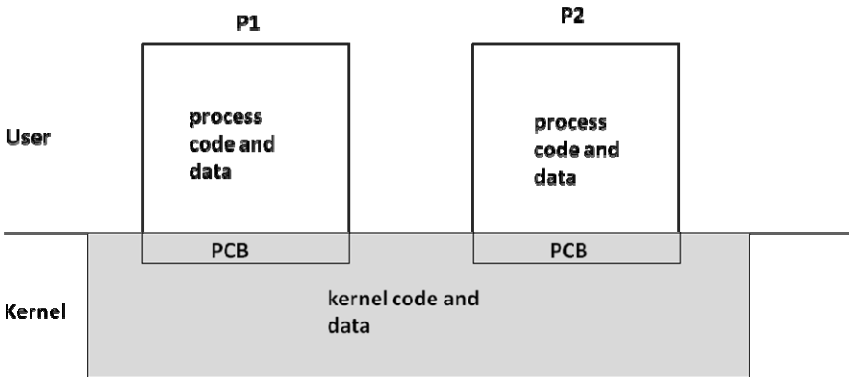


Figure 12.16: Memory protection in traditional operating systems. There is an enforeced separation between the user and kernel

Every process is in its own address space. There is a clear dividing line between the user space and kernel space. System calls result in a change of protection domain. Now that we are familiar with memory hierarchy, we can see that the working set (which affects all the levels of the memory hierarchy from virtual memory to processor caches) changes on each such address space crossing. Consequently, frequent crossing of the boundary is detrimental to performance. A process control block (PCB) defines a particular process. In previous chapters, we have seen how the various components of the operating system such as the scheduler, memory system, and the I/O subsystem use the PCB. In a traditional operating system (i.e. one that is not multithreaded), the process is single-threaded. Hence, the PCB contains information for fully specifying the activity of this single thread on the processor (current PC value, stack pointer value, general-purpose register values, etc.). If a process makes a system call that blocks the process (e.g. read a file from the disk), then the program as a whole does not make any progress.

Most modern operating systems (Windows[xp], Sun Solaris, HP Tru64, etc.) are multithreaded. That is, the operating system recognizes that the state of a running program is a composite of the states of all its constituent threads. Figure 12.17 shows the distribution of memory spaces in modern operating systems supporting both single-threaded as well as multithreaded programs. All threads of a given process share the address space of the process.

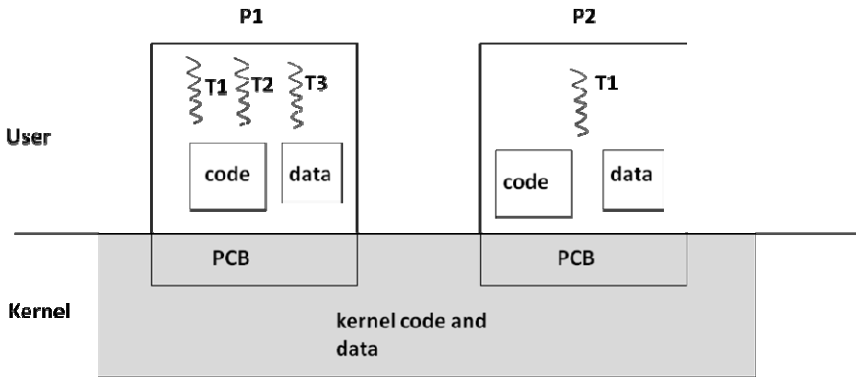
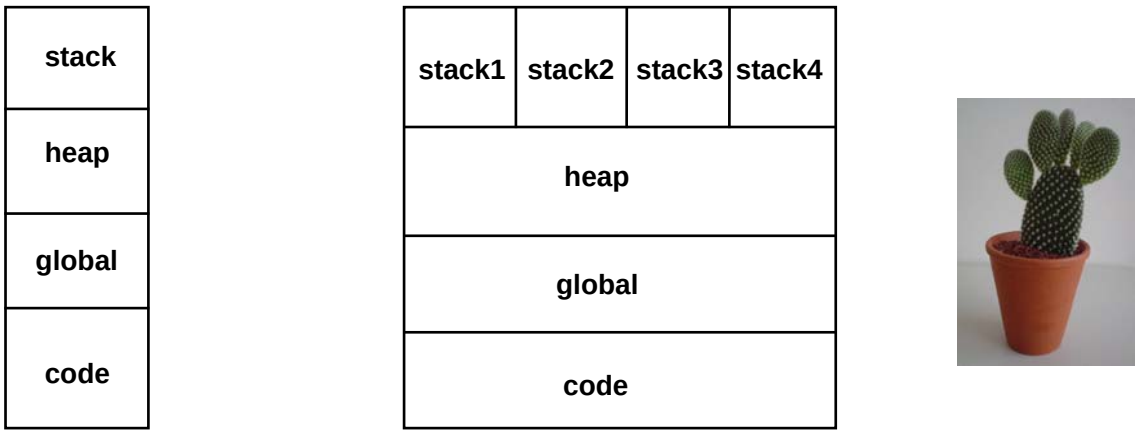


Figure 12.17: Memory protection in modern operating systems. A process may encompass multiple threads.

A given process may have multiple threads but since all the threads share the same address space, they share a common page table in memory. Let us elaborate on the computational state of a thread. A *thread control block (TCB)* contains all the state information pertaining to a thread. However, the information contained in a TCB is minimal compared to a PCB. In particular, the TCB contains the PC value, stack pointer value, and the GPR contents.

It is interesting to contrast the memory layout of multithreaded processes and single threaded processes (Figure 12.18). As we mentioned earlier, all the threads of a given process share the code, global data, and the heap. Thus, the stack is the only portion of memory that is unique for a particular thread. Due to the similarity in the visual picture of the stack layout of a multithreaded process to the cactus plant (Figure 12.18 (c)), we refer to this stack as a *cactus stack*.



(a) Single threaded process **(b) Multi threaded process** **(c) Cactus Plant**

Figure 12.18: Memory layout of single threaded and multithreaded processes[7]

Next, we will see what it takes to implement a threads library: first at the user level and then at the kernel level.

[7] Picture source for cactus plant: unknown.

12.7.1 User level threads

First, we will consider the implementation to be entirely at the user level. That is, the operating system only knows the existence of processes (which are single threaded). However, we could still have threads implemented at the user level. That is, the threads library exists as functionality above the operating system just as you have math libraries available for use in any program. The library provides thread creation calls we discussed earlier and supports data types such as **`mutex`** and **`cond_var`**, and operations on them. A program that wishes to use threads links in the threads library that becomes part of the program as shown in Figure 12.19.

The operating system maintains the traditional ready queue with the set of schedulable processes, which is the unit of scheduling at the operating system level. The threads library maintains a list of ready to run *threads* in each process with information about them in their respective *thread control blocks (TCBs)*. A TCB contains minimal information about each thread (PC value, SP value, and GPR values). Processes at the user level may be single threaded (e.g. P3) or multithreaded (P1 and P2).

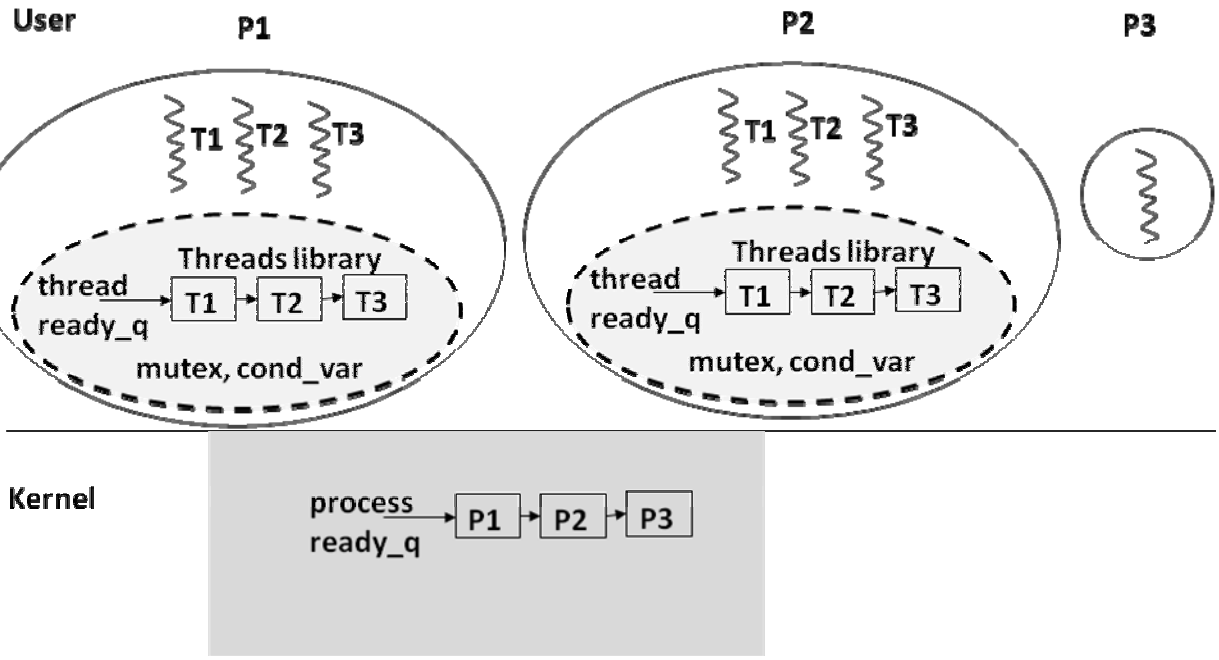


Figure 12.19: User level threads. Threads library is part of the application process's address space. In this sense, there is no enforced memory protection between the application logic and the threads library functionalities. On the other hand, there is an enforced separation between the user and the kernel.

The reader may wonder as to the utility of user level threads if process is the unit of scheduling by the operating system. Even if the underlying platform is a multiprocessor, the threads at the user level in a given process cannot execute concurrently. Recall, that thread is a structuring mechanism for constructing software. This is the primary purpose

served by the user level threads. They operate as *co-routines*, that is, when one thread makes a thread synchronization call that blocks it, the thread scheduler picks some other thread within the same process to run. The operating system is oblivious to such thread level context switching that the thread scheduler performs using the TCBs. It is cheap to switch threads at the user level since it does not involve the operating system. The cost of a context switch approximates making a procedure call in a program. Thus, user level threads provide a structuring mechanism without the high cost of context switch involving the operating system. User level threads incur minimal direct and indirect cost for context switching (see Chapter 9.14 for details on direct and indirect costs). Further, thread level scheduler is customizable for a specific application.

It is interesting to understand what happens when one of the threads in a multithreaded process makes a blocking system call. In this case, the operating system blocks the whole process since it has no knowledge of other threads within the same process that are ready to run. This is one of the fundamental problems with user level threads. There are a number of different solution approaches to this problem:

1. One possibility is to wrap all OS calls with an envelope (for e.g. *fopen* becomes *thread_fopen*) that forces all the calls to go through the thread library. Therefore, when some thread (e.g. T1 of P1 in Figure 12.19) makes such a call, the thread library recognizes that issuing this call to the OS will block the whole process. Therefore, it defers making the call until all the threads in the process are unable to run anymore. At that point the library issues the blocking call to the OS on behalf of the thread.
2. A second approach is for an *upcall* mechanism (Figure 12.20) from the operating system that warns the thread scheduler that a thread in that process is about to make a blocking system call. This warning allows the thread scheduler (in the library) to perform a thread switch and/or defer the blocking call by the thread to a later more opportune time. Of course, this approach requires extension to the operating system for supporting such upcalls.

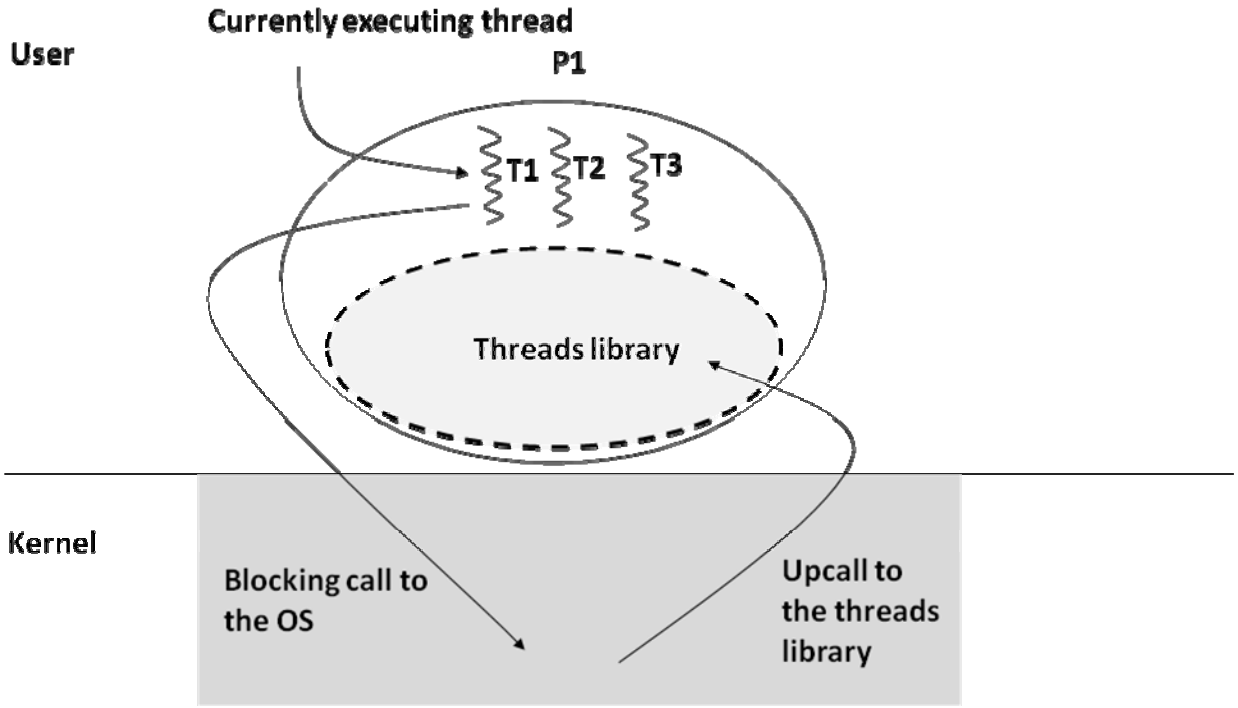


Figure 12.20: Upcall mechanism. Thread library registers a handler with the kernel. Thread makes a blocking call to the OS. The OS makes an upcall to the handler, thus alerting the threads library of the blocking system call made by a thread in that process.

In the chapter on CPU scheduling, we explored different CPU scheduling policies. Given that background, let us see how the thread scheduler switches among user level threads. Of course, if a thread makes a synchronization call into the threads library that is an opportune moment for the threads scheduler to switch among the threads of this process. Similarly, threads library may provide a *thread_yield* call to let a thread voluntarily give up the processor to give a chance for other threads in the same process to execute. One of the scheduling disciplines we studied in Chapter 6 is preemptive scheduling. If it is necessary to implement a preemptive thread scheduler at the user level, the thread scheduler can request a timer interrupt from the kernel and use that as a trigger to perform preemptive scheduling among the threads.

12.7.2 Kernel level threads

Let us understand the operating system support needed for implementing threads at the kernel level:

1. All the threads of a process live in a single address space. Therefore, the operating system should ensure that the threads of a process share the same page table.
2. Each thread needs its own stack, but share other portions of the memory footprint.
3. The operating system should support the thread level synchronization constructs discussed earlier.

First, let us consider a simple extension to the process level scheduler to support threads in the kernel. The operating system may implement a two-level scheduler as shown in Figure 12.21. A process level scheduler manages PCBs with information that is common

to all the threads in that process (page table, accounting information, etc.). A thread level scheduler manages TCBs. The process level scheduler allocates *time quanta* for processes, and performs preemptive scheduling among processes. Within a time quantum, the thread-level scheduler schedules the threads of that process in a round robin fashion or in a co-routine mode with the threads voluntarily yielding the processor. Since the operating system recognizes threads, it can switch among the threads of a process when the currently executing thread makes a blocking system call for I/O or thread synchronization.

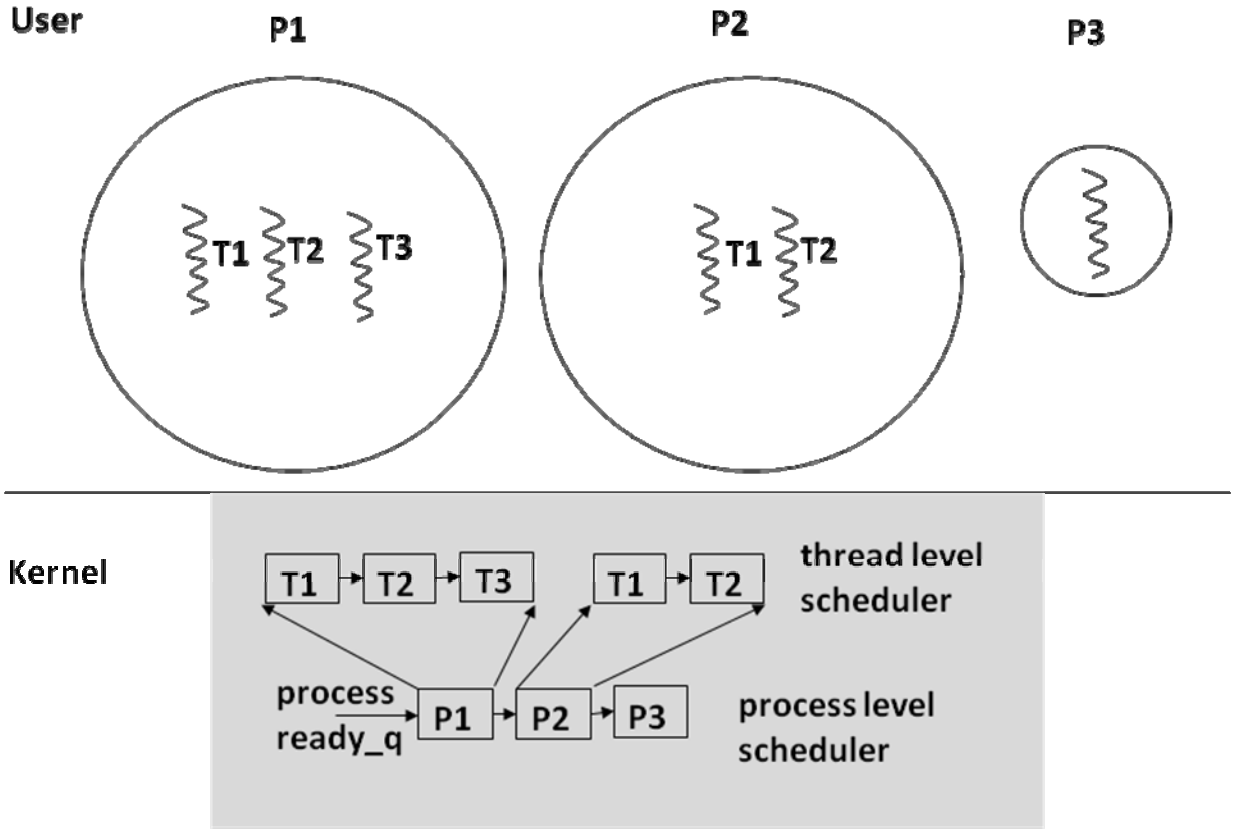


Figure 12.21: Kernel level threads. The process level scheduler uses the process ready_q. Even if a thread in the currently scheduled process makes a blocking system call, the OS uses the thread level scheduler to pick a ready thread from the currently running process to run for the remainder of the time quantum.

With computers and chips these days becoming multiprocessors, it is a serious limitation if the threads of a process cannot take advantage of the available hardware concurrency. The above structure allows threads of a given process to overlap I/O with processing, a definite step forward from user level threads. However, to fully exploit available hardware concurrency in a multiprocessor, a thread should be the unit of scheduling in the operating system. Next, we will discuss Sun Solaris threads as a concrete example of kernel level threads.

12.7.3 Solaris threads: An example of kernel level threads

Figure 12.22 shows the organization of threads in the Sun Solaris operating system.

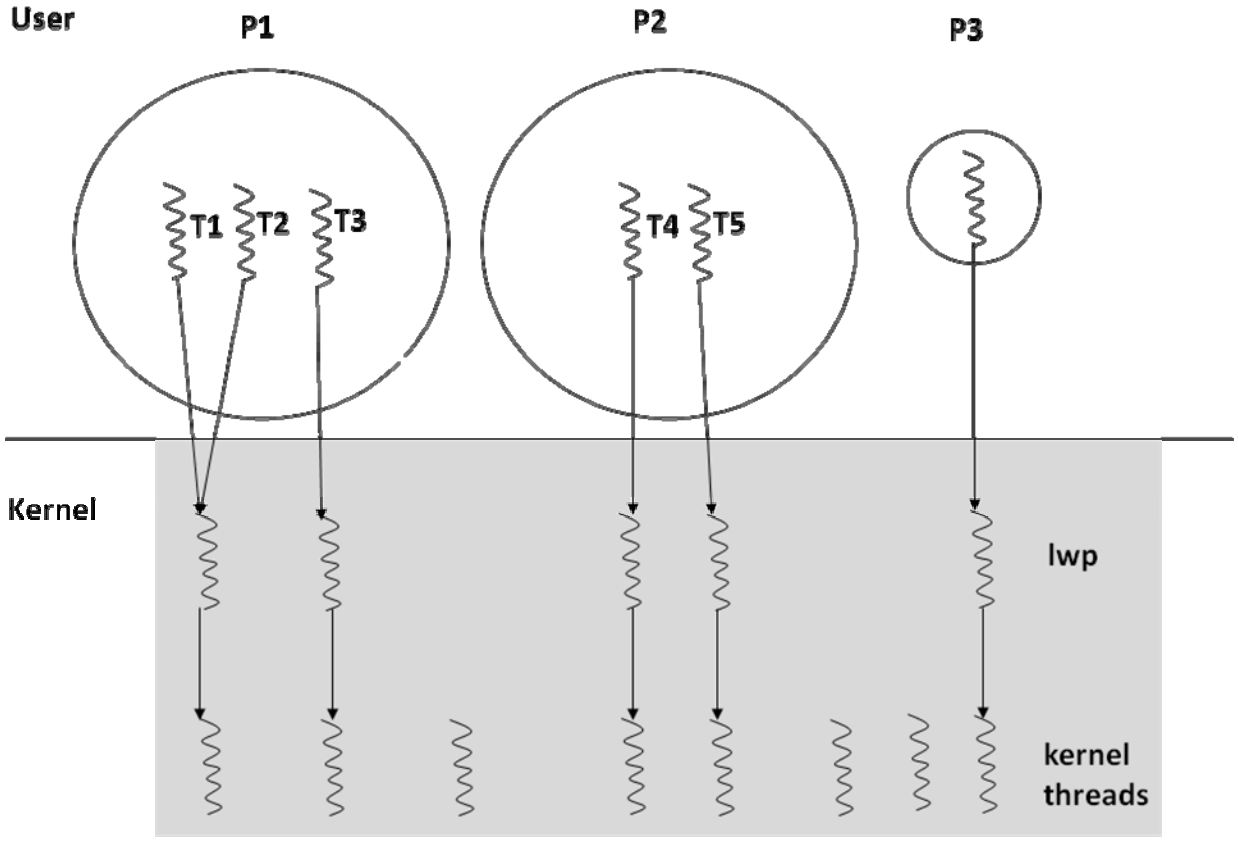


Figure 12.22: Sun Solaris threads structure. A thread in a process is bound to a light-weight process (lwp). Multiple threads of the same process may be bound to the same lwp. There is a one-to-one mapping between an lwp and a kernel thread. The unit of processor scheduling is a kernel thread.

A process is the entity that represents a program in execution. A process may create any number of user level threads that are contained within that process. The operating system allows the creator of threads to have control on the scheduling semantics of threads within that process. For example, threads may be truly concurrent or execute in a co-routine manner. To support these different semantics, the operating system recognizes three kinds of threads: *user*, *lightweight process (lwp)*, and *kernel*.

1. **Kernel:** The kernel threads are the unit of scheduling. We will see shortly its relation to the lwp and user threads.
2. **lwp:** An lwp is the representation of a process within the kernel. Every process upon startup is associated with a distinct lwp. There is a one-to-one association between an lwp and a kernel thread as shown in Figure 12.22. On the other hand, a kernel thread may be unattached to any lwp. The operating system uses such threads for functions that it has to carry out independent of user level processes. For example, kernel threads serve as vehicles for executing device specific functions.

3. **User:** As the name suggests, these threads are at the user level.

Thread creation calls supported by the operating system result in the creation of user level threads. The thread creation call specifies if the newly thread should be attached to an existing lwp of the process or allocated to a new lwp. See for example the threads T1 and T2 in process P1. They both execute in a co-routine manner due to being bound to the same lwp. Any number of user level threads may bind themselves to an lwp. On the other hand, thread T3 of P1 executes concurrently with one of T1 or T2. Threads T4 and T5 of P2 execute concurrently as well.

The ready queue of the scheduler is the set of kernel threads that are ready to run. Some of them, due to their binding to an lwp result in executing a user thread or a process. If a kernel thread blocks, the associated lwp and therefore the user level thread blocks as well.Since the unit of scheduling is a kernel thread, the operating system has the ability to schedule these threads concurrently on parallel processors if the underlying platform is a multiprocessor.

It is interesting to understand the inherent cost of a thread switch given this structure. Remember that every context switch is between one kernel thread to another. However, the cost of the switch changes dramatically depending on what is bound to a kernel thread. The cheapest form of context switch is between two user level threads bound to the same lwp (T1 and T2 in Figure 12.22). Presumably, the process has a thread library available at the user level. Thus, the thread switch is entirely at the user level (similar to user level threads we discussed earlier), Switching between lwps in the same process (for example T1 and T3 in Figure 12.22) is the next higher cost context switch. In this case, the direct cost is a trip through the kernel that performs the switch (in terms of saving and loading TCBs). There are no hidden costs in the switch due to memory hierarchy effects since both the threads are in the same process. The most expensive context switch is between two lwps that are in different processes (for example T3 and T4 in Figure 12.22). In this case, there are both direct and indirect costs involved since the threads are in different processes.

12.7.4 Threads and libraries

Irrespective of the choice of implementing threads at the user or the kernel level, it is imperative to ensure the safety of libraries that multithreaded programs use. For example, all the threads of a process share the heap. Therefore, the library that supports dynamic memory allocation needs to recognize that threads may request memory from the heap simultaneously. It is usual to have *thread-safe* wrappers around such library calls to ensure atomicity for such calls in anticipation of concurrent calls from the threads. Figure 12.23 shows an example. The library call implicitly acquires a mutual exclusion lock on behalf of the thread that makes that call.

```
/* original version */   |  /* thread safe version */
                         |
                         |  mutex_lock_type cs_mutex;
void *malloc(size_t size)|  void *malloc(size_t size)
{                        |  {
                         |   thread_mutex_lock(cs_mutex);
                         |
  ......                 |   ......
  ......                 |   ......
                         |
                         |   thread_mutex_unlock(cs_mutex);
                         |
  return(memory_pointer);|   return (memory_pointer);
}                        |  }
```

Figure 12.23: Thread safe wrapper for library calls

12.8 Hardware support for multithreading in a uniprocessor

Let us understand what is required in hardware for supporting multithreading. There are three things to consider:

1. Thread creation and termination
2. Communication among threads
3. Synchronization among threads

12.8.1 Thread creation, termination, and communication among threads

First, let us consider a uniprocessor. Threads of a process share the same page table. In a uniprocessor each process has a unique page table. On a thread context switch within the same process, there is no change to the TLB or the caches since all the memory mapping and contents of the caches remain relevant for the new thread. Thus, creation and termination of threads, or communication among the threads do not require any special hardware support.

12.8.2 Inter-thread synchronization

Let us consider what it takes to implement the *mutual exclusion lock*. We will use a memory location **`mem_lock`** initialized to zero. The semantics are as follows. If **`mem_lock`** is zero, then the lock is available. If **`mem_lock`** is 1, then some thread currently has the lock. Here are the algorithms for **`lock`** and **`unlock`**.

```
Lock:
     if (mem_lock == 0)
               mem_lock = 1;
     else
               block the thread;

Unlock:
     mem_lock = 0;
```

The lock and unlock algorithms have to be *atomic*. Let us examine these algorithms. Unlock algorithm is a single memory store instruction of the processor. Assuming that each instruction is atomic, unlock is atomic.

The datapath actions necessary to implement the lock algorithm are as follows:

- Read a memory location
- Test if the value read is 0
- Set the memory location to 1
-

12.8.3 An atomic test-and-set instruction

We know that LC-2200 (see Chapter 2) does not provide any single instruction that performs the above datapath actions. Therefore, to make the lock algorithm atomic, we introduce a new instruction:

`Test-And-Set memory-location`

The semantics of this instruction is as follows:

- Read the current value of memory-location into some processor register
- Set the memory-location to a 1

The key point of this instruction is that if a thread executes this instruction, the above two actions (getting the current value of the memory location and setting the new value to 1) happen *atomically*, i.e., no other instruction (by any other thread) intervenes the execution of **`Test-and-set`**.

Example 14:

Given the following procedure called **`binary-semaphore`**:

```
static        int shared-lock =  0; /* global variable to
                                       both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int L)
{
              int X;

              X = test-and-set (L);

              /* X = 0 for successful return */
              return(X);
}
```

Two threads **T1** and **T2** execute the following statement simultaneously:

```
              MyX = binary_semaphore(shared-lock);
```

where **`MyX`** is a local variable in each of **T1** and **T2**.

What are the possible values returned to T1 and T2?

Answer:
Note that the instruction **`test-and-set`** is atomic. Therefore, although T1 and T2 execute the procedure simultaneously, the semantics of this instruction ensures that one or the other (whichever happens to be the winner) gets to execute the instruction first.

So, possible outcomes:
1) T1 is the winner.
T1's `MyX` = 0; T2's `MyX` = 1
2) T2 is the winner.
T1's `MyX` = 1; T2's `MyX` = 0

Note that it will never be the case that **both** T1 and T2 will get a 0 or 1 as the return value.

You may have seen and heard of the *semaphore* signaling system used extensively in railroads. In olden times (and even to this day in some developing countries), mechanical arms (see Figure 12.24) on a high pole cautioned an engine driver in a train to either stop or proceed with caution when approaching shared railroad tracks.

Figure 12.24: Railroad Semaphore[8]

Computer scientists have borrowed that term, semaphore, into computer science parlance. The procedure shown in Example 13 is a *binary semaphore*, i.e., it signals one among many threads that it is safe to proceed into a critical section.

Edsger Dijkstra, a well-known computer scientist of Dutch origin, was the proponent of semaphore as a synchronization mechanism for coordinating the activities of concurrent threads. He proposed two versions of this semaphore. Binary semaphore is the one we just saw wherein the semaphore grants or denies access to a single resource to a set of competing threads. *Counting semaphore* is a more general version wherein there are *n*

[8] Picture source: http://www.stuorg.iastate.edu/railroad/images/FortDodge03/120703-UP2085nose.JPG

instances of a given resource; the semaphore grants or denies access to **an** instance of these *n* resources to competing threads. At most *n* threads can enjoy these resources **simultaneously** at any point of time.

12.8.4 Lock algorithm with test-and-set instruction

Having introduced the atomic **`test-And-Set`** instruction, we are now ready to review the implementation of the mutual exclusion lock primitive, which is at the core of programming support for multithreaded applications. As it turns out, we can implement the lock and unlock algorithm building on the binary semaphore as follows.

```
#define SUCCESS 0
#define FAILURE 1


int lock(int L)
{
     int X;
     while ( (X = test-and-set (L)) == FAILURE ) {
               /* current value of L is 1
                * implying that the lock is
                * currently in use
                */
               block the thread;

               /* the threads library puts the
                * the thread in a queue; when
                * lock is released it allows
                * this thread to check the
                * availability of the lock again
                */
     }

     /* falling out of the while loop implies that
      * the lock attempt was successful
      */

     return(SUCCESS);

}

int unlock(int L)
{
     L = 0;
     return(SUCCESS);
}
```

By design, a thread calling the lock algorithm has the lock when it returns. Using this basic lock and unlock algorithm, we can build the synchronization primitives we discussed in Section 12.3 and the POSIX thread library in Section 12.6.

Therefore, the minimal hardware support for multithreading is an atomic **`Test-And-Set`** (**`TAS`** for short) instruction. The key property of this instruction is that it *atomically reads, modifies, and writes* a memory location. There are other instructions that implement the same property. The point is that most modern processor architectures include one or more instructions in its repertoire that have this property.
Note that if the operating system deals with threads directly, then it can simply turn off interrupts while executing the lock algorithm to ensure atomicity. The **`TAS`** instruction allows implementing locks at the **user** level.

12.9 Multiprocessors

As the name suggest, a multiprocessor consists of multiple processors in a single computer sharing all the resources such as memory, bus, and input/output devices (see Figure 12.25). This is a *Symmetric multiprocessor (SMP)* since all the processors have an identical view of the system resources. An SMP is a cost effective approach to increasing the system performance at a nominal increase in total system cost. Many servers that we use on an everyday basis (web server, file server, mail server) run on 4-way or 8-way SMPs.

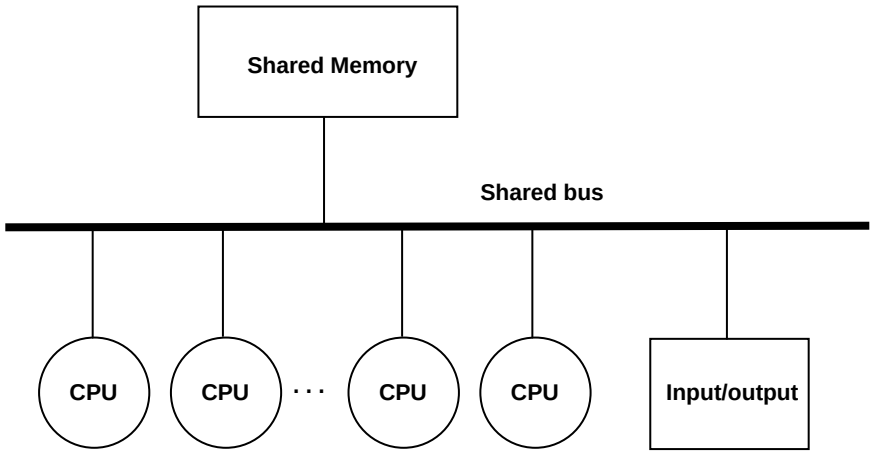


Figure 12.25: A Symmetric Multiprocessor (SMP)

Complication arises at the system level if the program is running on a multiprocessor. In this case, the threads of a given program may be on different physical processors. Thus, the system software (meaning the operating system and runtime libraries) and the hardware have to work in partnership to provide the semantics expected at the user program level for data structures shared by the threads.

We have seen that there are several intricacies associated with preserving the semantics of a sequential program with the hardware and software entities even in a single processor: TLB, page tables, caches, and memory manager just to name a few. The reader

can envision the complications that arise when the system software and hardware have to preserve the semantics of a multithreaded program. We will discuss these issues in this section.

The system (hardware and the operating system together) has to ensure three things:

1. Threads of the same process share the same page table.
2. Threads of the same process have identical views of the memory hierarchy despite being on different physical processors.
3. Threads are guaranteed atomicity for synchronization operations while executing concurrently.

12.9.1 Page tables

The processors share physical memory as shown in Figure 12.25. Therefore, the operating system satisfies the first requirement by ensuring that the page table in shared memory is the same for all the threads of a given process. However, there are a number of issues associated with an operating system for a multiprocessor. In principle, there is a copy of the operating system in each processor executing independently. However, they coordinate some of their decisions for overall system integrity. In particular, they take specific coordinated actions to preserve the semantics of multithreaded programs. These include scheduling threads of the same process simultaneously on different processors, page replacement, and maintaining the consistency of the TLB entries in each CPU. Such issues are beyond the scope of this discussion. Suffice it to say, these are fascinating issues and the reader is encouraged to take advanced courses in operating systems to study them.

12.9.2 Memory hierarchy

Each CPU has its own TLB and cache. As we said earlier, the operating system worries about the consistency of TLBs to ensure that all threads have the same view of the shared process address space. The hardware manages the caches. Each per-processor cache may currently be encaching the same memory location. Therefore, the hardware is responsible for maintaining a consistent view of shared memory that may be encached in the per-processor caches (see Figure 12.26). We refer to this problem as *multiprocessor cache coherence*.

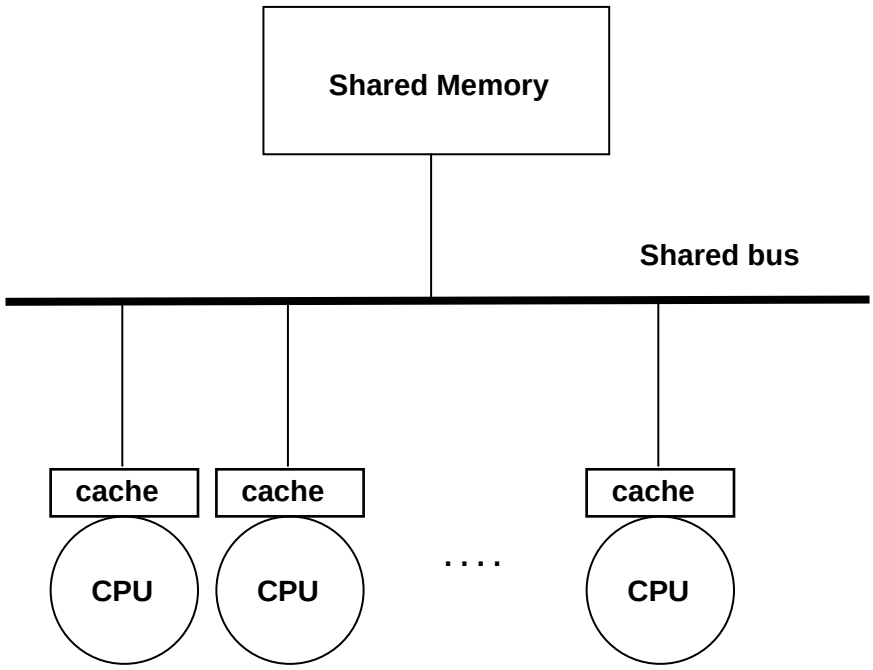


Figure 12.26: SMP with per-processor caches

Figure 12.27 illustrates the cache coherence problem. Threads T1, T2, and T3 (of the same process) execute on processors P1, P2, and P3, respectively. All three of them currently have location X cached in their respective caches (Figure 12.27-a). T1 writes to X. At this point, the hardware has one of two choices:

- It can *invalidate* copies of X in the peer caches as shown in Figure 12.27-b. This requires enhancement to the shared bus in the form of an *invalidation line*. Correspondingly, the caches *monitor* the bus by *snooping* on it to watch out for invalidation requests from peer caches. Upon such a request on the bus, every cache checks if this location is cached locally; if it is then it invalidates that location. Subsequent misses for the same location is either satisfied by the cache that has the most up to date copy (for example P1 in this example) or by the memory, depending on the write policy used. We refer to this solution as the *write-invalidate* protocol.
- It can *update* copies of X in the peer caches as shown in Figure 12.27-c. This may manifest simply as a memory write on the bus. The peer caches that *observe* this bus request update their copies of X (if present in the cache). We refer to this solution as the *write-update* protocol.

Snoopy caches is a popular term used for bus-based cache coherence protocols. In this section, we have presented a very basic intuition to the multiprocessor cache coherence problem and snoopy cache solution approaches to the problem. Snoopy caches do not work if the processors do not have a shared bus (a broadcast medium) to snoop on. In Section 12.10.2.4, we will discuss a different solution approach called *directory-based scheme* that does not rely on a shared bus for communication among the processors. Investigation of scalable solution approaches to the cache coherence problem was a fertile area of research in the mid to late 80's and resulted in several doctoral

dissertations. The reader is encouraged to take advanced courses in computer architecture to learn more on this topic.

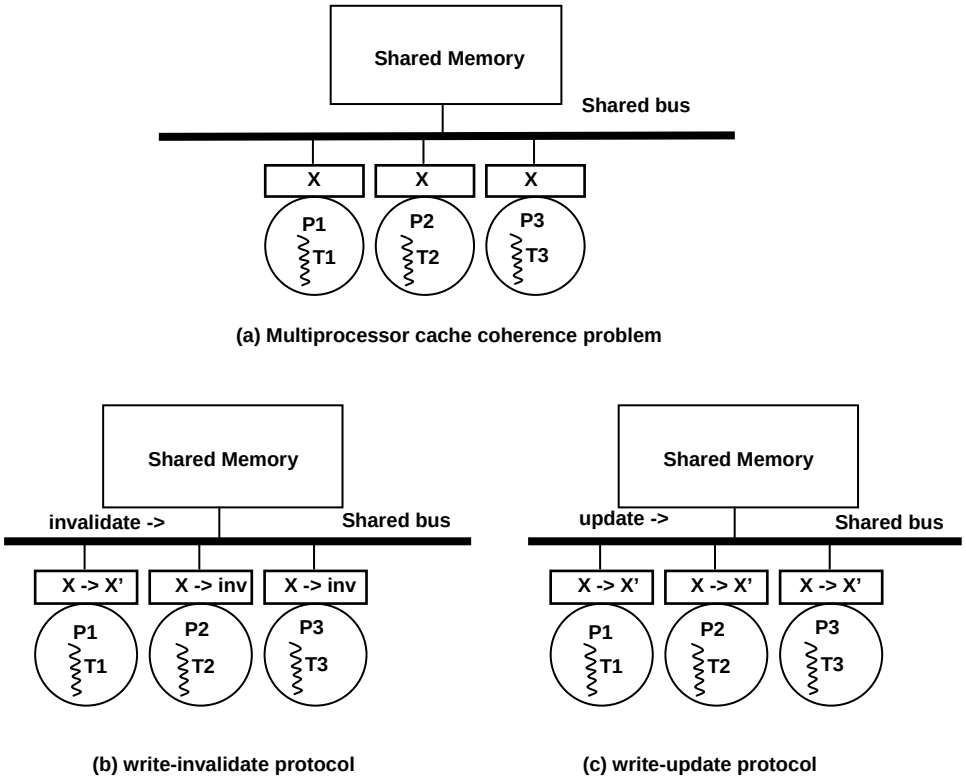


Figure 12.27: Multiprocessor cache coherence problem and solutions

---

Example 15:

Given the following details about an SMP (symmetric multiprocessor):

Cache coherence protocol: **write-invalidate**
Cache to memory policy: **write-back**

Initially:

The caches are empty
Memory locations:

A contains 10
B contains 5

Consider the following timeline of memory accesses from processors P1, P2, and P3.

Time (in increasing order)	Processor P1	Processor P2	Processor P3
T1	Load A		
T2		Load A	
T3			Load A
T4		Store #40, A	
T5	Store #30, B		

Using the table below, summarize the activities and the values in the caches.

Answer:

(**I indicates the cache location is invalid. NP indicates not present**)

Time	Variables	Cache of P1	Cache of P2	Cache of P3	Memory
T1	**A**	**10**	**NP**	**NP**	**10**
T2	**A**	**10**	**10**	**NP**	**10**
T3	**A**	**10**	**10**	**10**	**10**
T4	**A**	**I**	**40**	**I**	**10**
T5	**A** **B**	**I** **30**	**40** **NP**	**I** **NP**	**10** **5**

12.9.3 Ensuring atomicity

Since the CPUs share memory, the lock and unlock algorithms, we presented in Chapter 12.8 work fine in a multiprocessor. The key requirement is ensuring atomicity of these algorithms in the presence of concurrent execution of the threads in different processors. An instruction such as TAS introduced in Chapter 12.8 that has the ability to atomically *read-modify-write* a shared memory location serves this purpose.

12.10 Advanced Topics

We will introduce the reader to some advanced topics of special relevance in the context of multiprocessors and multithreading.

12.10.1 OS topics

12.10.1.1 Deadlocks

In Section 12.2.7, we introduced the concept of deadlocks and livelocks. We will generalize and expand on the concept of deadlocks. We gave a very simple and intuitive definition of deadlock as a situation where in a thread is waiting for an event that will never happen. There are several reasons that lead to deadlocks in a computer system. One reason is that there are concurrent activities in the system, and there are finite resources be they hardware or software. For example, let us consider a uniprocessor running multiple applications. If the scheduler uses a non-preemptive algorithm, and the application currently running on the processor goes into an infinite loop, all the other applications are deadlocked. In this case, the processes are all waiting for a physical resource, namely, a processor. This kind of deadlock is usually referred to as a *resource deadlock*. The conditions that led to the deadlock are two-fold: the need for *mutual exclusion* for accessing the shared resource (i.e., the processor) and the *lack of preemption* for yanking the resource away from the current user.

A similar situation arises due to locks governing different data structures of a complex application. Let us first consider a fun analogy. Nick and his partner arrive in the recreation center to play squash. There is only one court and exactly two rackets in the recreation center. There is one service desk to check out rackets and another to claim the court. Nick and his partner go first to the former and check out the rackets. Alex and his partner have similar plans. However, they first claim the court and then go to the service desk to check out the rackets. Now Nick and Alex are deadlocked. This is also a resource deadlock. In addition to the two conditions we identified earlier, there are two conditions that led to the deadlock in this case: *circular wait* (Alex is waiting for Nick to give up the rackets, and Nick is waiting for Alex to give up the court), and the fact that each of them can *hold a resource and wait for another one*. Complex software systems use fine-grain locking to enhance the opportunity for concurrent execution of threads. For example, consider payroll processing. A check issuing process may be locking the records of all the employees to generate the paychecks; while a merit-raise process may be sweeping through the database locking all the employee records to add raises to all the employees at the same time. *Hold and wait*, and *circular wait* by each of these processes will lead to a deadlock.

To sum up there are four conditions that *must hold simultaneously* for processes to be involved in resource deadlock in a computer system:

- **Mutual exclusion**: a resource can be used only in a mutually exclusive manner
- **No preemption:** the process holding a resource has to give it up voluntarily
- **Hold and wait:** a process is allowed to hold a resource while waiting for other resources
- **Circular wait:** there is a cyclic dependency among the processes waiting for resources (A is waiting for resource held by B; B is waiting for a resource held by C; C….X; X is waiting for a resource held by A)

These are called *necessary* conditions for a deadlock. There are three strategies for dealing with deadlocks. Deadlock *avoidance*, *prevention*, and *detection*. A reader interested in detailed treatment of these strategies is referred to advanced textbooks in Operating Systems[9]. Here we will give the basic intuition behind these strategies. Deadlock avoidance algorithm is ultra-conservative. It basically assumes that the request pattern for resources are all known *a priori*. With this global knowledge, the algorithm will make resource allocation decisions that are guaranteed to never result in a deadlock. For example, if you have $100 in hand, and you know that in the worst case you need a maximum of $80 to see you through the rest of the month, you will be able to loan $20 to help out a friend. If your friend needs $30, you will say no since you could potentially get into a situation where you don't have enough to see you through the rest of the month. However, it could turn out that this month you get some free lunches and dinners and may end up not needing $80. So, you are making a conservative decision as to how much you can loan a friend based on the worst case scenario. As you probably guessed, deadlock avoidance will lead to poor resource utilization due to its inherent conservatism.

More importantly, deadlock avoidance is simply not practical since it requires prior knowledge of future resource requests. A better strategy is deadlock prevention, which goes after each of the four *necessary* conditions for deadlock. The basic idea is to break one of the necessary conditions and thus *prevent* the system from deadlocking. Using the same analogy, you could loan your friend $30. However, if it turns out that this month you do need $80 to survive, you go to your friend and get back $10 out of the $30 you loaned him. Basically, this strategy breaks the necessary condition "no preemption." Of course, the same prevention strategy may not be applicable for all types of resources. For example, if processes need to access a single shared physical resource in a mutually exclusive manner, then one way to avoid deadlock is to pretend as though there are as many instances of that shared resource as the number of requestors. This may seem like a crazy idea but if you think about it, this is precisely how a departmental printer is shared. Basically, we spool our print jobs that are then buffered awaiting the physical printer. Essentially, "spooling" or "buffering" is a way to break the necessary condition "mutual exclusion". Similarly, to break the necessary condition, "hold and wait," we can mandate that all resources need to be obtained simultaneously before starting the process. Returning to our squash player analogy, Nick (or Alex) would have to get both the court and the rackets together, not one after another. Lastly, to break the necessary condition "circular wait" we could order the resources and mandate that the requests be made *in order*. For example, we could mandate that you first have to claim the squash court (resource #1), before you request the rackets (resource #2). This would ensure that there will not be any circular wait.

Deadlock prevention leads to better resource utilization compared to avoidance. However, it is still conservative. For example, consider mandating that a process get all the resources *a priori* before it starts. This for sure prevents deadlock. However, if the process does not need all the resources for the entire duration, then the resources are getting under-utilized. Therefore, a more liberal policy is deadlock detection and recovery. The idea is to be liberal with resource requests and grants. However, if a

[9] A. S. Tannenbaum, "Modern Operating Systems," Prentice-Hall.

deadlock does occur, have a mechanism for detecting it and recovering from it. Returning to our squash players analogy, when the desk clerk at the court claim counter notices the deadlock, she takes the racket from Nick, calls up her cohort at the racket claim counter, and tells her to send Alex over to her counter to resolve the deadlock.

A topic that is closely related to deadlocks in the context of resource allocation is *starvation*. This is the situation wherein some process is indefinitely blocked awaiting a resource. For example, if the resource allocation uses some sort of a priority scheme, then lower priority processes may be starved, if there is a steady stream of higher priority processes making requests for the same resource. Returning to our squash court example, consider the situation where faculty member are given priority over students. This could lead to starvation of students. We give another example of starvation when we discuss classic problems in synchronization (see Section 12.10.1.4).

Beyond resource deadlocks, computer systems are susceptible to other kinds of deadlocks as well. Specifically, the kinds of deadlocks we discussed earlier in this chapter have to do with errors in writing correct parallel program that could lead to deadlocks and livelocks. This problem gets exacerbated in distributed systems (see Chapter 13), where messages may be lost in transit due to a variety of reasons, leading to deadlocks. All of these can be lumped under the heading of *communication deadlocks*.

12.10.1.2 Advanced synchronization algorithms

In this chapter, we studied basic synchronization constructs. Such constructs have been incorporated into IEEE standards such as POSIX threading library. As we observed earlier, these libraries or their variants have been incorporated into almost all modern operating systems. Most application software for parallel systems is built using such multithreading libraries.

Programming mutual exclusion locks and condition variables is error-prone. The primary reason is that the logic of synchronized access to shared data structures is strewn all over the program and makes it hard from a software-engineering point of view. This has implications on the design, development, and maintenance of large complex parallel programs.

We can boil down the needs of concurrent programming to three things:

(1) an ability for a thread to execute some sections of the program (which we called critical sections in Section 12.2.4) in a mutually exclusive manner (i.e., serially),
(2) an ability for a thread to wait if some condition is not satisfied, and
(3) an ability for a thread to notify a peer thread who may be waiting for some condition to become true.

Monitor is a programming construct proposed by Brinch Hansen and Tony Hoare in the 70's that meet the above needs. Essentially, a monitor is abstract data type that contains the data structures and procedures for manipulating these data structures. In terms of modern programming languages such as Java or C++, one can think of the monitor as syntactically similar to an object. Let us understand the difference between a Java object

and a monitor. The principal difference is that there can be exactly one active thread inside a monitor at any point of time. In other words, if you have a need for a critical section in your program, you will write that portion of the program as a monitor. A program structure that requires multiple independent critical sections (as we saw in Example 7), will be constructed using multiple monitors, each with a distinct name for each of the critical sections. A thread inside the monitor may have to block for a resource. For this purpose the monitor provides *condition variables*, with two operations *wait* and *notify* on such a variable. The reader can see an immediate parallel to the condition variable inside a monitor and the condition variable available in the pthreads library. The monitor construct meets all the three needs for writing concurrent programs that we laid out above. To verify this is true, we will do an example.

Example 16:
Write a solution using monitors for the video processing example developed in this chapter.

Answer:

The digitizer and tracker codes are written assuming the existence of a monitor called FrameBuffer. The procedures grab and analyze used by the digitizer and tracker, respectively, are outside the monitor.

```
digitizer()
{
  image_type dig_image;

  loop {
    grab(dig_image);
    FrameBuffer.insert(dig_image);
  }
}

tracker()
{
  image_type track_image;

  loop {
    FrameBuffer.remove_image(&track_image);
    analyze(track_image);
  }
}
```

```
monitor FrameBuffer
{

#define MAX 100

image_type frame_buf[MAX];
int bufavail = MAX;
int head = 0, tail = 0;

condition not_full, not_empty;

    void insert_image(image_type image)
    {
       if (bufavail == 0)
           wait(not_full);
       frame_buf[tail mod MAX] = image;
       tail = tail + 1;
       bufavail = bufavail - 1;
       if (bufavail == (MAX-1)) {
         /* tracker could be waiting */
         notify(not_empty);
       }
     }

     void remove_image(image_type *image)
     {
       if (bufavail == MAX)
           wait(not_empty);
       *image = frame_buf[head mod MAX];
       head = head + 1;
       bufavail = bufavail + 1;
       if (bufavail == 1) {
         /* digitizer could be waiting */
         notify(not_full);
       }
     }

} /* end monitor */
```

A number of things are worth commenting on the solution presented in Example 16. The most important point to note is that all the details of synchronization and buffer management that were originally strewn in the digitizer and tracker procedures in the pthreads version are now nicely tucked away inside the monitor named **`FrameBuffer`**. This simplifies the tracker and digitizer procedures to be just what is intuitively needed for those functions. This also ensures that the resulting program will be less error-prone

compared to sprinkling synchronization constructs all over the program. Another elegant aspect of this solution is that there can be any number of digitizer and tracker threads in the application. Due to the semantics of the monitor construct (mutual exclusion), all the calls into the monitor from these various threads get serialized. Overall, the monitor construct adds significantly to the software-engineering of parallel programs.

The reader may be wondering if the monitor is such a nice construct why we are not using it. The main catch is that it is a programming construct. In the above example, we have written the monitor using a C-style syntax to be compatible with the earlier solutions of the video processing application developed in this chapter. However, C does not support the monitor construct. One could of course "simulate" the monitor construct by writing the monitor using facilities available in the operating system (e.g., pthreads library, please see Exercise 16).

Some programming languages have adopted the essence of the monitor idea. For example, Java is an object-oriented programming language. It supports user-level threads and allows methods (i.e., procedures) to be grouped together into what are called classes. By prefixing a method with the keyword "synchronized," the Java runtime will ensure that exactly one user-level thread is able to execute a synchronized method at any time in a given object. In other words, as soon as a thread starts executing a synchronized method, no other thread will be allowed to another synchronized method within the same object. Other methods that do not have the keyword synchronized may of course be executed concurrently with the single execution of a synchronized method. While Java does not have an in-built data type similar to the monitor's condition variable, it does provide wait and notify functions that allow blocking and resumption of a thread inside a synchronized method (please see Exercise 17).

12.10.1.3 Scheduling in a Multiprocessor

In Chapter 6, we covered several processor scheduling algorithms. All of these apply to parallel systems as well. However, with multiple processors the scheduler has an opportunity to run multiple threads of the same application or threads of different applications. This gives rise to some interesting options.

The simplest way to coordinate the scheduling of the threads on the different processors is to have a single data structure (a run queue) that is shared by the scheduler on each of the processors (Figure 12.28). This also ensures that all the processors share the computational load (in terms of the runnable threads) equally.

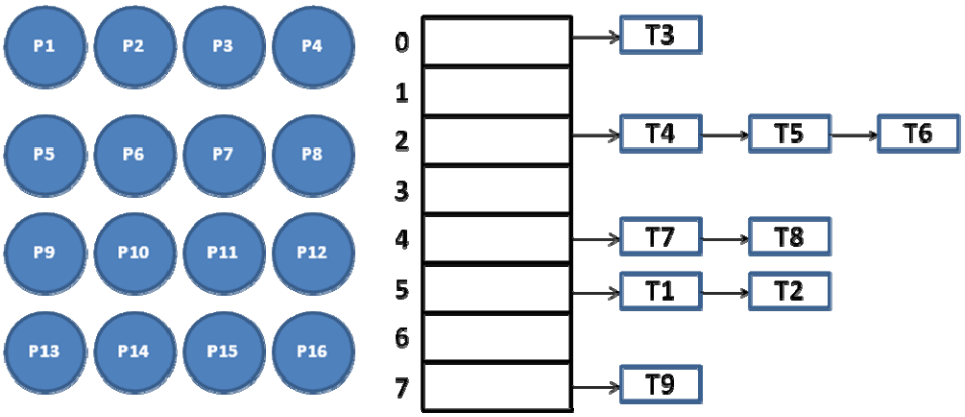


Figure 12.28: Shared scheduling queue with different priority levels

There are some downsides to this approach. The first issue is pollution of the memory hierarchy. On modern processors with multiple levels of caches, there is significant time penalty for reaching into levels that are farther away from the processor (see Chapter 9 for more details). Consider a thread T1 that was running on processor P2. Its time quantum ran out and it was context switched out. With a central queue, T1 may be picked up by some other processor, say P5, the next time around. Let us understand why this may not be a desirable situation. Well, perhaps most of the memory footprint of T1 is still in the nearer levels of processor P2's memory hierarchy. Therefore, T1 would have incurred less cache misses if it was run on P2 the second time around. In other words, T1 has an *affinity* for processor P2. Thus, one embellishment to the basic scheduling algorithm when applied to multiprocessors is to use cache affinity, a technique first proposed by Vaswani and Zahorjan. A per-processor scheduling queue (shown in Figure 12.29) would help in managing the threads and their affinity for specific processors than a shared queue. For load balancing, processors who find themselves out of work (due to their queues empty) may do what is usually referred to as *work stealing* from the scheduling queues of other processors. Another embellishment is to *lengthen the time quantum* for a thread that is currently holding a mutual exclusion lock. The intuition behind this is the fact that other threads of the same application cannot really run until this thread relinquishes the lock. Zahorjan proposed this technique as well. Figure 12.29 shows how a conceptual picture of a per processor scheduling queue.

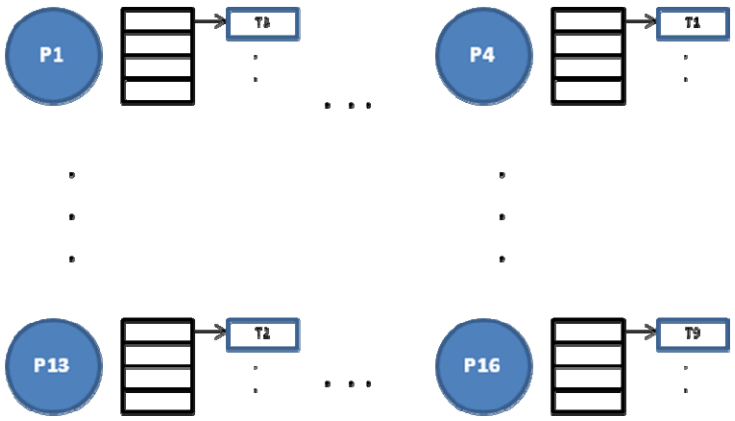


Figure 12.29: Per processor scheduling queue

We will introduce the reader to two additional techniques for making multiprocessor scheduling more effective, especially for multithreaded applications. The first one is called *space sharing*. The idea here is to dedicate a set of processors for an application for its lifetime. At application start up, as many processors as there are threads in the application are allocated to this application. The scheduler will delay starting the application until such time as that many idle processors are available. Since a processor is dedicated to a thread, there is no context switching overhead (and affinity is preserved as well). If a thread is blocked on synchronization or I/O, then the processor cycles are simply wasted. This technique provides excellent service to applications at the risk of wasting resources. Modifications to this basic idea of space sharing is for the application to have the ability to scale up or down its requirements for CPUs depending on the load on the system. For example, a web server may be able to shrink the number of threads to 10 instead of 20 if the system can provide only 10 processors. Later on, if the system has more processors available, the web server can claim them and create additional threads to run on those processors. A space sharing scheduler would divide up the total number of processors in the system into different sized partitions (see Figure 12.30) and allocate partitions at a time to applications instead of individual processors. This reduces the amount of book-keeping data structures that the scheduler needs to keep. This should remind the reader of fixed sized partition memory allocation we studied in Chapter 7. Similar to that memory management scheme, space sharing may result in internal fragmentation. For example, if an application requires 6 processors, it would get a partition with 8 processors. Two of the 8 processors will remain idle for the duration of the application.

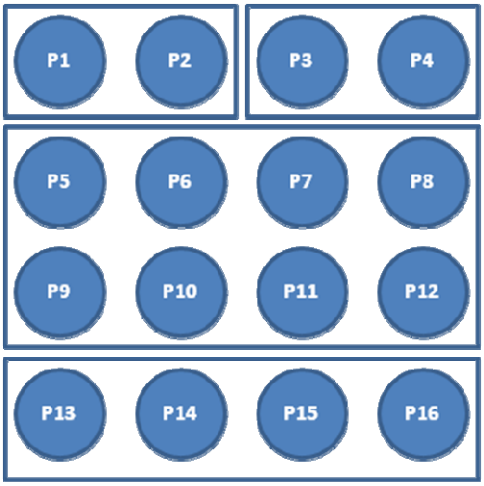


Figure 12.30: The scheduler has created 4 partitions to enable space sharing. There are two 2-processor partitions; and one each of a 4-processor and 8-processor partitions.

The last technique we introduce is *gang scheduling*. It is a complementary technique to space sharing. Consider the following. Thread T1 is holding a lock; T2 is waiting for it. T1 releases the lock but T2 is not currently scheduled to take advantage of the lock that just became available. The application may have been designed to use fine-grain locks such that a thread holds a lock only for a very short amount of time. In such a situation, there is tremendous loss of productivity for this application due to the fact that the scheduler is unaware of the close coupling that exists among the threads of the application. Gang scheduling alleviates this situation. Related threads of an application are scheduled as a group, hence the name gang scheduling. Each thread runs on a different CPU. However, in contrast to space sharing a CPU is not dedicated to a single thread. Instead, each CPU is time-shared.

Gang scheduling works as follows:

- Time is divided into fixed size quanta
- All the CPUs are scheduled at the beginning of each time quantum
- The scheduler uses the principle of gangs to allocate the processors to the threads of a given application
- Different gangs may use the same set of processors in different time quanta
- Multiple gangs may be scheduled at the same time depending on the availability of the processors
- Once scheduled the association of a thread to a processor remains until the next time quantum even if the thread blocks (i.e., the processor will remain idle)

Gang scheduling may incorporate cache affinity in its scheduling decision. Figure 12.31 shows how three gangs may share 6 CPUs both in space and time using the gang scheduling principle.

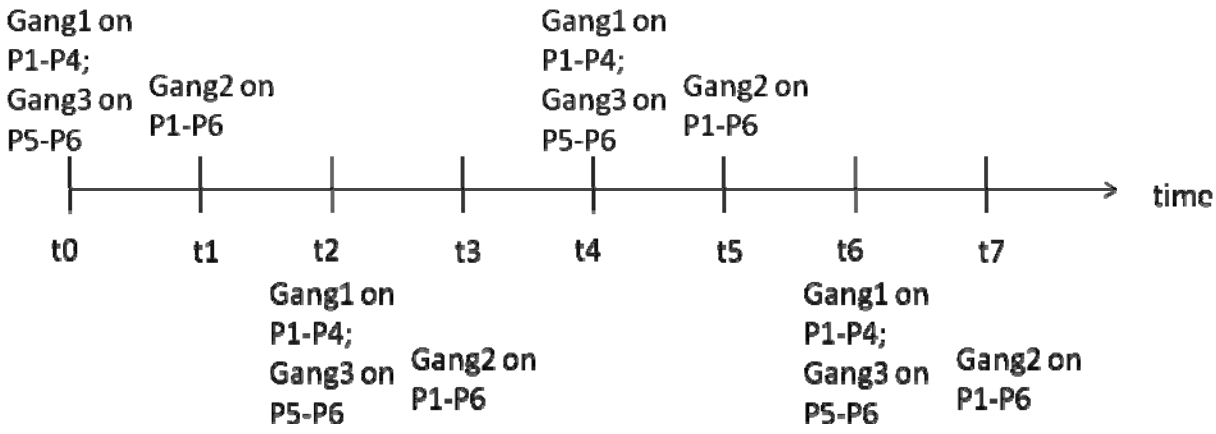


Figure 12.31: Gang1 needs 4 processors; Gang2 needs 6 processors; Gang3 needs 2 processors. Timeline of gang scheduling these three different gangs.

To summarize, we introduced the reader to four techniques that are used to increase the efficiency of CPU scheduling in a multiprocessor:

- Cache affinity scheduling
- Lock-based time quantum extension
- Space sharing
- Gang scheduling

A multiprocessor scheduler might use a combination of these above techniques to maximize efficiency depending on the workload for which the system is intended. Finally, these techniques are on top of and work in concert with the short-term scheduling algorithm that is used in each processor (see Chapter 6 for coverage of short-term scheduling algorithms).

12.10.1.4 Classic problems in concurrency

We will conclude this discussion by mentioning some classic problems in concurrency that have been used as a way of motivating the synchronization issues in parallel systems.

(1) Producer-consumer problem: This is also referred to as the bounded-buffer problem. The video processing application that has been used as running example in this chapter is an instance of the producer-consumer problem. Producers keep putting stuff into a shared data structure and consumers keep taking stuff out of it. This is a common communication paradigm that occurs in several applications. Any application that can be modeled as a pipeline uses this communication paradigm.

(2) Readers-writers problem: Let us say you are trying to get tickets to a ball game. You would go to a website such as Ticketmaster.com. You would go to the specific date on which you want to get the tickets and check availability of seats. You may look at different options in terms of pricing and seating before you finally decide to lock in on a specific set of seats and purchase the tickets. While you are doing this, there are probably 100's of others who are also looking to purchase tickets for the same game on the same

day. Until you actually purchase the seats you want, those seats are up for grabs by anyone. Basically, a database contains all the information concerning availability of seats on different dates, etc. Looking for availability of seats is a read operation on the database. Any number of concurrent readers may browse the database for availability. Purchasing tickets is a write operation on the database. This requires exclusive access to the database (or at least part of the database), implying that there should be no readers present while during the writing.

The above example is an instance of the classic readers-writers problem first formulated in 1971 by Courtois, et al. A simple minded solution to the problem would go like this. So long as readers are in the database, allow new readers to come in since none of them needs exclusive access. When a writer comes along, make him wait if there are readers currently active in the database. As soon as all the readers have exited the database, let the writer in exclusively. Similarly, if a writer in the database, block the readers until the writer exist the database. This solution using mutual exclusion locks is shown in Figure 12.32.

```
mutex_lock_type readers_count_lock, database_lock;
int readers_count = 0;

void readers()
{
   lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
     if (readers_count == 0) {
       /* first reader in a new group,
        * obtain lock to the database
        */
       lock(database_lock);
       /* note only first reader does this,
        * so in effect this lock is shared by
        * all the readers in this current set
        */
     }
     readers_count = readers_count + 1;
   unlock(read_count_lock);
   read_dabatase();
   lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
     readers_count = readers_count - 1;
     if (readers_count == 0) {
       /* last reader in current group,
        * release lock to the database
        */
       unlock(database_lock);
     }
   unlock(read_count_lock);
}

void writer()
{
   lock(database_lock);   /* get exclusive lock */
   write_dabatase();
   unlock(database_lock); /* release exclusive lock */
}
```

Figure 12.32: Solution to readers-writers problem using mutual exclusion locks

Pondering on this simple solution, one can immediately see some of its shortcomings. Since readers do not need exclusive access, so long as at least one reader is currently in the database, the solution will continue to let new readers in. This will lead to starvation of writers. The solution can be fixed by checking if there are waiting writers when a new reader wants to join the party, and simply enqueueing the new reader behind the waiting writer (please see Exercises 18-21).

(3) Dining Philosophers problem: This is a famous synchronization problem due to Dijkstra (1965). Since the time Dijkstra posed and solved this problem in 1965, any new synchronization construct that is proposed used this problem as a litmus test to see how efficiently the proposed synchronization construct solves it. Five philosophers are sitting around a round table. They alternate between eating and thinking. There is a bowl of spaghetti in the middle of the table. Each philosopher has his individual plate. There are totally five forks, one in between every pair of philosophers as shown in Figure 12.33. When a philosopher wants to eat, he picks up the two forks nearest to his on either side, gets some spaghetti from the bowl on to his plate and eats. Once done eating, he puts down his forks and continues thinking[10].

Figure 12.33: Dining Philosophers[11]

[10] It would appear that these philosophers did not care much about personal hygiene reusing forks used by neighbors!

[11] Picture source: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6a/Dining_philosophers.png/578px-Dining_philosophers.png, and http://www.dkimages.com

The problem is to make sure each philosopher does his bit of eating and thinking without bothering anyone else. In other words, we want each philosopher to be an independent thread of execution with maximal concurrency for the two tasks that each has to do, namely, eating and thinking. Thinking requires no coordination since it is an individual effort. On the other hand, eating requires coordination since there is a shared fork between every two philosophers.

A simple solution is for each philosopher to agree up front to pick up one fork first (say the left fork), and then the right fork and start eating. The problem with this solution is that it does not work. If every philosopher picks up the left fork simultaneously, then they are each waiting for the right fork, which is held by their right neighbor and this leads to the circular wait condition of deadlock.

Let us sketch a possible correct solution. We will introduce an intermediate state between *thinking* and *eating*, namely, *hungry*. In the hungry state, a philosopher will try to pick up both the forks. If successful, he will proceed to the eating state. If he cannot get both forks simultaneously, he will keep trying until he succeeds in getting both forks. What would allow a philosopher to pick up both forks simultaneously? Both his immediate neighbors *should not be* in the *eating* state. Further, he will want to ensure that his neighbors do not change states while he is doing the testing to see if he can pick up the forks. Once done eating, he will change his state to thinking, and simply notify his immediate neighbors so that they may attempt to eat if they are hungry. Figure 12.34 gives a solution to the dining philosophers' problem using monitors. Note that when a philosopher tries to get the forks using take_forks, monitor due to its guarantee that there is only one active thread in it at any point of time, ensures that no other philosopher can change his state.

```
void philosopher(int i)
{
  loop {/* forever */
    do_some_thinking();
    DiningPhilosphers.take_forks(i);
    eat();
    DiningPhilosophers.put_down_forks(i);
  }
}
```

Figure 12.34-(a): Code that each philosopher thread executes.

```
monitor DiningPhilosophers
{
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT  ((i+N-1) mod N)
#define RIGHT ((i+1) mod N)

condition phil_waiting[N]; /* one for each philosopher */
int phil_state[N]; /* state of each philosopher */

        void take_forks (int i)
        {
          phil_state[i] = HUNGRY;
          repeat
            if ( (phil_state[LEFT] != EATING) &&
                 (phil_state[RIGHT] != EATING) )
                phil_state[i] = EATING;
            else
                wait(phil_waiting[i]);
          until (phil_state[i] == EATING);
        }

        void put_down_forks (int i)
        {
          phil_state[i] = THINKING;
          notify(phil_waiting[LEFT]);  /* left neighbor
                                          notified */
          notify(phil_waiting[RIGHT]); /* right neighbor
                                          notified */
        }

        /* monitor initialization code */
        init:
        {
          int i;
          for (i = 1; i < N; i++) {
                phil_state[i] = THINKING;
          }
        }

} /* end monitor */
```

Figure 12.34-(b): Monitor for Dining Philosophers problem

12.10.2 Architecture topics

In Section 12.9, we introduced multiprocessors. We will delve a little deeper into advanced topics relating to parallel architectures.

12.10.2.1 Hardware Multithreading

A pipelined processor exploits instruction level parallelism (ILP). However, as we have discussed in earlier chapters (see Chapters 5 and 9), there are limits to exploiting instruction-level parallelism due to a variety of factors including branches, limited functional units, and the growing gap between processor cycle time and memory cycle time. For example, cache misses result in pipeline stalls and the severity grows with the level at which cache misses occur. A cache miss that requires off-chip servicing could result in the CPU waiting for several tens of clock cycles. With multithreaded programs, there is another avenue to use the processor resources efficiently, namely across all the active threads that are ready to run. This is referred to as *thread level parallelism (TLP)*. M*ultithreading* at the hardware level comes in handy to help reduce the ill effects of pipeline stalls due to ILP limitations by exploiting TLP.

A simple analogy will help here. Imagine a queue of people awaiting service in front of a single bank teller. A customer walks up to the teller. The teller finds that the customer needs to fill out a form before the transaction can be completed. The teller is smart. She asks the customer to step aside and fill out the form, and starts dealing with the next customer. When the first customer is ready again, the teller will deal with him to finish the original transaction. The teller may temporarily sideline multiple customers for filling out forms, and thus keep the waiting line moving efficiently.

Hardware multithreading is very much like this real life example. The uniprocessor is the teller. The customers are the independent threads. A long latency operation that could stall the processor is akin to filling out a form by the customer. Upon a thread stall on a long latency operation (e.g., a cache miss that forces main memory access), the processor picks the next instruction to execute from another ready thread, and so on. Thus, just like the bank teller, the processor utilizes its resources efficiently even if one or more ready threads are stalled on long latency operations. Naturally, this raises several questions. How does the processor know there are multiple threads ready to run? How does the operating system know that it can schedule multiple threads simultaneously to run on the same physical processor? How does the processor keep the *state* (PC, register file, etc.) of each thread distinct from one another? Does this technique work only for speeding up multithreaded applications or does it also work for sequential applications? We will answer these questions in the next couple of paragraphs.

Multithreading in hardware is yet another example of a contract between hardware and system software. The processor architecture specifies how many concurrent threads it can handle in hardware. In the Intel architecture, this is referred to as the number of *logical processors*. Essentially, this represents the level of *duplication* of the hardware resources needed to keep the states of the threads distinct. Each logical processor has its own distinct PC, and register file. The operating system allows an application to bind a thread to a logical processor. Earlier we discussed the operating system support for

thread level scheduling (Section 12.7). With multiple logical processors at its disposal, the operating system can simultaneously schedule multiple ready threads for execution on the processor. The physical processor maintains distinct persona for each logical processor through duplicate sets of register files, PC, page tables, etc. Thus, when an instruction corresponding to a particular logical processor goes through the pipeline, the processor knows the thread-specific hardware resource that have to be accessed for the instruction execution.

A multithreaded application stands to gain with hardware support for multithreading, since even if one thread is blocked due to limitations of ILP, some other thread in the same application may be able to make forward progress. Unfortunately, if an application is single-threaded then it cannot benefit from hardware multithreading to speed up its execution. However, hardware multithreading would still help improve the *throughput* of the system as a whole. This is because hardware multithreading is agnostic as to whether the threads it is pushing through the pipeline belong to independent processes or part of the same process.

This discussion begs another question: can exploitation of ILP by the processor using superscalar design coexist with exploitation of TLP? The answer is yes, and this is precisely what most modern processors do to enhance performance. The basic fact is that modern multiple-issue processors have more functional units than can be gainfully used up by a single thread. Therefore, it makes perfect sense to combine ILP exploitation using multiple-issue with TLP exploitation using the logical processors.

Each vendor gives a different name for this integrated approach to exploiting ILP and TLP. Intel calls it *hyperthreading* and is pretty much a standard feature on most Intel line of processors. IBM calls is *simultaneous multithreading (SMT)* and uses it in the IBM Power5 processor.

12.10.2.2 Interconnection Networks

Before we look at different types of parallel architectures, it is useful to get a basic understanding of how to interconnect the elements inside a computer system. In the case of a uniprocessor, we already learned in earlier chapters (see Chapter 4, 9, and 10) the concept of buses that allow the processor, memory, and peripherals to be connected to one another. The simplest form of interconnection network for a parallel machine is a shared bus (Section 12.9). However, large-scale parallel machines may have on the order of 1000's of processors. We will call each node in such a parallel machine a *processing element – PE* for short. A shared bus would become a bottleneck for communication among the PEs in such a large-scale machine. Therefore, such machines use more sophisticated interconnection networks such as a *mesh* (each processor is connected to its four neighbors, north, south, east, and west), or a *tree*. Such sophisticated interconnection networks allow for simultaneous communication among different PEs. Figure 12.35 shows some examples of such sophisticated interconnection networks. Each PE may locally have buses to connect to the memory and other peripheral devices.

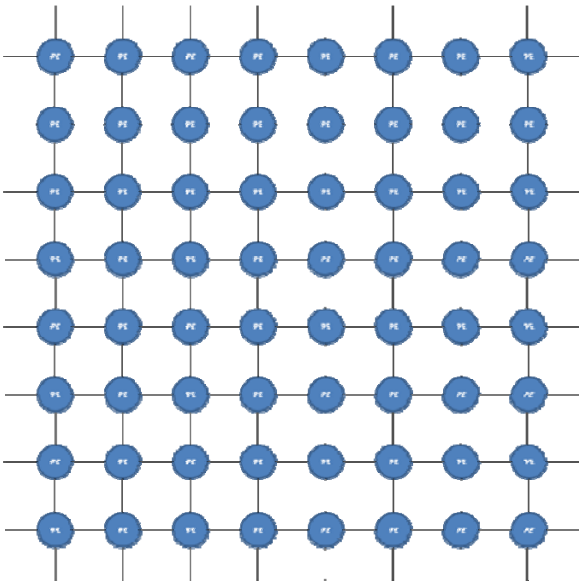


Figure 12.35-(a): A Mesh Interconnection Network

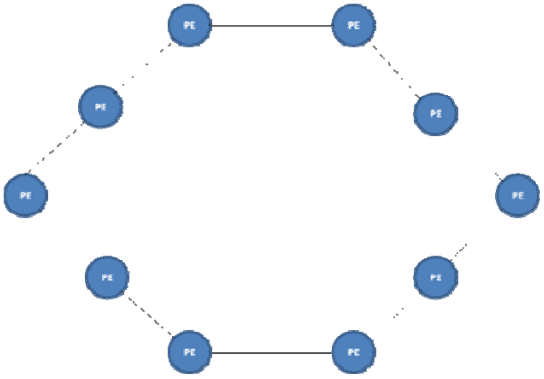


Figure 12.35-(b): A Ring Interconnection Network

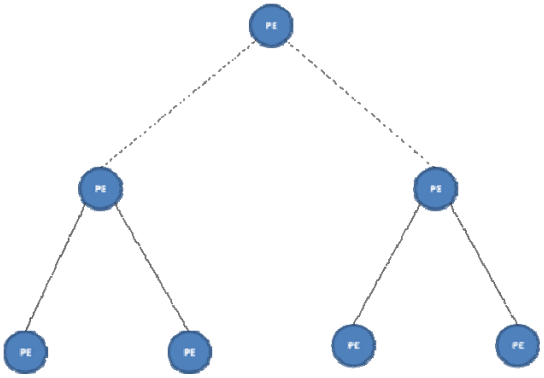


Figure 12.35-(c): A Tree Interconnection Network

12.10.2.3 Taxonomy of Parallel Architectures

Flynn in 1966 proposed a simple taxonomy for characterizing all architectures based on the number of independent instruction and data streams concurrently processed. The taxonomy, while useful for understanding the design space and the architectural choices, also aided the understanding of the kinds of applications that are best suited for each architectural style. Figure 12.36 shows the taxonomy graphically.

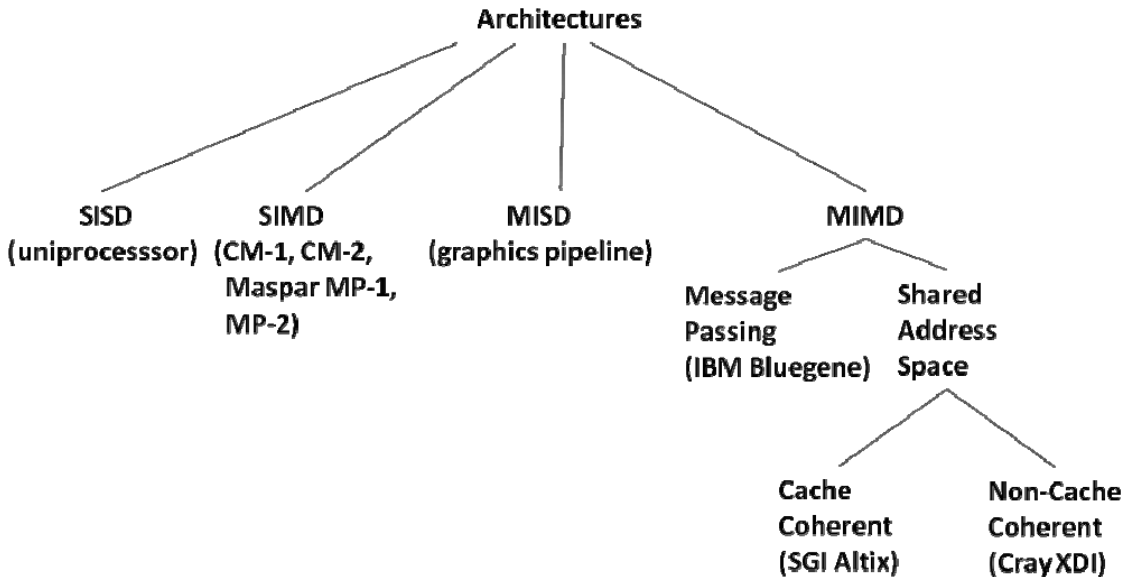


Figure 12.36: A Taxonomy of Parallel Architectures

Single Instruction Single Data (SISD): This is the classic uniprocessor. Does a uniprocessor exploit any parallelism? We know that at the programming level, the uniprocessor deals with only a single instruction stream and executes these instructions sequentially. However, at the implementation level the uniprocessor exploits what is called *instruction-level parallelism* (*ILP* for short). It is ILP that allows pipelined and superscalar implementation of an instruction-set architecture (see Chapter 5).

Single Instruction Multiple Data (SIMD): All the processors execute the same instruction in a lock-step fashion, on independent data streams. Before the killer micros made this style of architecture commercially non-competitive (in the early 90's), several machines were built to cater to this architecture style. Examples include Thinking Machine Corporation's Connection Machine CM-1 and CM-2; and Maspar MP-1 and MP-2. This style of architecture was particularly well-suited to image processing applications (for example, applying the same operation to each pixel of a large image). Figure 12.37 shows a typical organization of an SIMD machine. Each processor of the SIMD machine is called a *processing element (PE)*, and has its own data memory that is pre-loaded with a distinct data stream. There is a single instruction memory contained in the control unit that fetches and broadcasts the instructions to the PE array. The instruction memory is also pre-loaded with the program to be executed on the PE array. Each PE executes the instructions with the distinct data streams from the respective data

memories. SIMD model promotes parallelism at a very *fine granularity*. For example, a `for` loop may be executed in parallel with each iteration running on a different PE. We define *fine-grained parallelism* as one in which each processor executes a small number of instructions (say less than 10) before communicating with other processors.

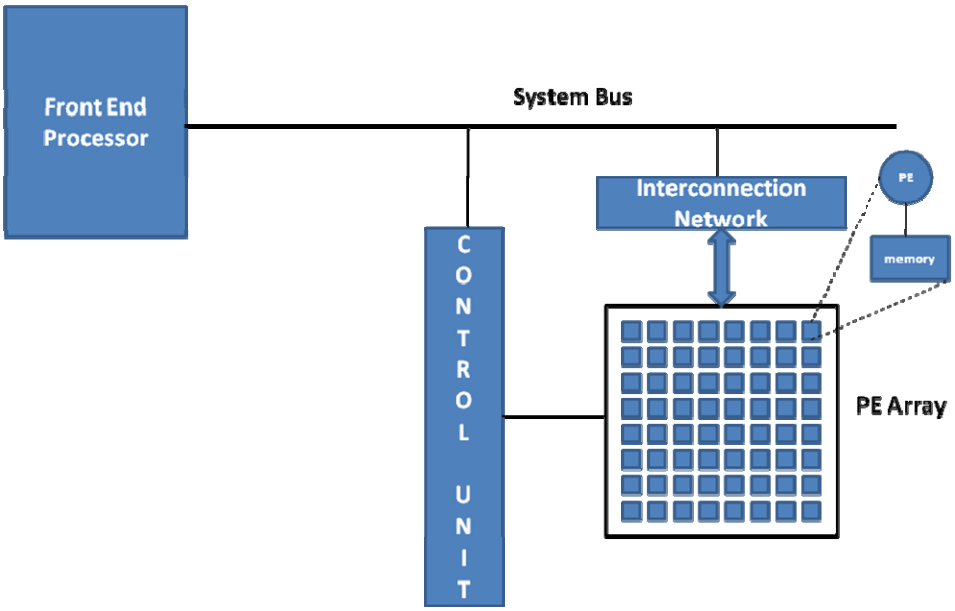


Figure 12.37: Organization of an SIMD machine. Front-end processor is typically a powerful workstation running some flavor of Unix. PE stands for processing element and is the basic building block of an SIMD machine

An SIMD machine is a workhorse to carry out some fine-grained compute-intensive task. Program development for the SIMD machine is usually via a front-end processor (a workstation class machine). The front-end is also responsible for pre-loading the data memories of the PEs and the instruction memory of the array control unit. All I/O is orchestrated through the front-end processor as well. The PEs communicate with one another using an interconnection network. Since SIMD machines may have on the order of 1000's of processors, they use sophisticated interconnection networks as discussed in Section 12.10.2.2 (e.g., mesh or a tree). Though no commercial machines of this flavor are in the market today, the Intel MMX instructions are inspired by the parallelism exploited by this architectural style. There has been a resurgence in this style of computation as exemplified by stream accelerators such as nVidia graphics card (referred to as *Graphics Processing Unit* or *GPU* for short). Further, as stream-oriented processing (audio, video, etc.) are becoming more mainstream, there is a convergence of the traditional processor architecture and the stream accelerators. For example, Intel's Larrabee *General Purpose GPU (GPGPU)* architecture represents one such integrated architecture meant to serve as the platform for future supercomputers.

Multiple Instructions Single Data (MISD): At an algorithmic level, one could see a use for this style of architecture. Let us say we want to run several different face detection algorithms on the same image stream. Each algorithm represents a different instruction

stream working on the same data stream (i.e., the image stream). While MISD serves to round out the classification of parallel architecture styles, there is not a compelling argument for this style. It turns out that most computations map more readily to the MIMD or the SIMD style. Consequently, no known architecture exactly matches this computation style. Systolic arrays[12] may be considered a form of MISD architecture. Each cell of a systolic array works on the data stream and passes the transformed data to the next cell in the array. Though it is a stretch, one could say that a classic instruction processing pipeline represents an MISD style at a very fine grain if you consider each instruction as "data" and what happens to the instruction as it moves from stage to stage.

Multiple Instructions Multiple Data (MIMD): This is the most general architectural style, and most modern parallel architectures fall into this architectural style. Each processor has its own instruction and data stream, and work asynchronously with respect to one another. The processors themselves may be off-the-shelf processors. The programs running on each processor may be completely independent, or may be part of a complex application. If it is of the latter kind, then the threads of the same application executing on the different processors need to synchronize explicitly with one another due to the inherent asynchrony of the architectural model. This architectural style is best suited for supporting applications that exhibit medium and coarse-grained parallelism. We define *medium-grained parallelism* as one in which each processor executes approximately 10-100 instructions before communicating with other processors. We define *coarse-grained parallelism* as one in which each processor executes on the order of 1000's of instructions before communicating with other processors.

Early multiprocessors were of the SIMD style. Necessarily, each processor of an SIMD machine has to be specially designed for that machine. This was fine so long as general-purpose processors were built out of discrete circuits. However, as we said already, the arrival of the killer micros made custom-built processors increasingly unviable in the market place. On the other hand, the basic building block in an MIMD machine is a general-purpose processor. Therefore, such architectures benefit from the technology advances of the off-the-shelf commercial processors. Further, the architectural style allows using such a parallel machine for running several independent sequential applications, threads of a parallel application, or some combination thereof. It should be mentioned that with the advent of stream accelerators (such as nVidia GeForce family of GPUs), hybrids of the parallel machine models are emerging.

12.10.2.4 Message-passing Vs. Shared Address Space Multiprocessors

MIMD machines may be subdivided into two broad categories: message-passing and shared address space. Figure 12.38 shows a picture of a message-passing multiprocessor. Each processor has its private memory, and processors communicate with each other using messages over an interconnection network. There is no shared memory among the processors. For this reason, such architectures are also referred to as *distributed memory* multiprocessors.

[12] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," inProc. SIAM Sparse Matrix Symp., 1978, pp. 256-282.

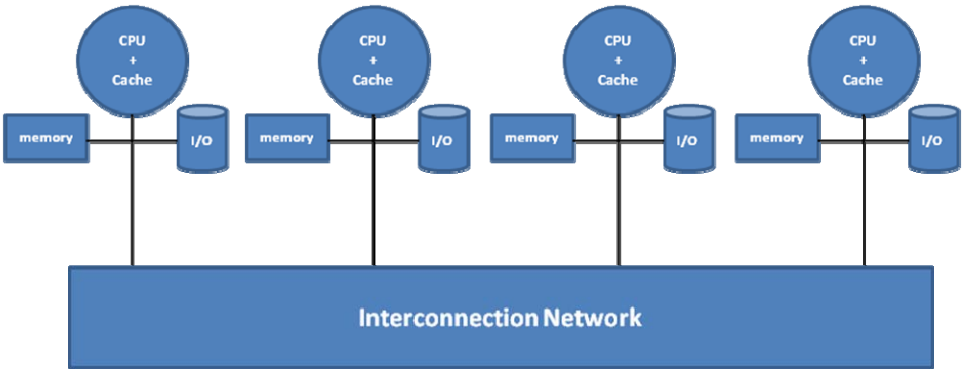


Figure 12.38: Message passing multiprocessor

IBM's Bluegene series is an example of a modern day message-passing machine that fits this model. Several examples of message passing machines of yester years include TMC's CM-5, Intel Paragon, and IBM SP-2. The previous generation of parallel machines relied on special interconnection network technologies to provide low-latency communication among the processors. However, with advances in computer networking (see Chapter 13), local area networking technology is offering viable low-latency high bandwidth communication among the processors in a message-passing multicomputer. Consequently, *cluster parallel machine*, a collection of computers interconnected by local area networking gear such as Gigabit Ethernet (see Section 13.6.1) has become a popular platform for parallel computing. Just as **`pthreads`** library offers a vehicle for parallel programming in a shared memory environment, *message passing interface* (*MPI* for short) is a programming library for the message-passing computing environment.

Shared address space multiprocessor is the second category of MIMD machines, which as the name suggests provides address equivalence for a memory location irrespective of which processor is accessing that location. In other words, a given memory address (say 0x2000) refers to the same physical memory location in whichever processor that address is generated. As we have seen, a processor has several levels of caches. Naturally, upon access to a memory location it will be brought into the processor cache. As we have already seen in Section 12.9, this gives rise to the cache coherence problem in a multiprocessor. Shared address space machines may be further subdivided into two broad categories based on whether this problem is addressed in hardware or software. Non-Cache Coherent (NCC) multiprocessors provide shared address space but no cache coherence in hardware. Examples of such machines from yester years include BBN Butterfly, Cray T3D, and Cray T3E. A current day example is Cray XDI.

Cache coherent (CC) multiprocessors provide shared address space as well as hardware cache coherence. Examples of this class of machines from yester years include KSR-1, Sequent Symmetry, and SGI Origin 2000. Modern machines of this class include SGI Altix. Further, the individual nodes of any high-performance cluster system (such as IBM Bluegene) is usually a cache coherent multiprocessor.

In Section 12.9, we introduced the reader to a bus-based shared memory multiprocessor. A large-scale parallel machine (regardless of whether it is a message-passing or a shared address space machine) would need a more sophisticated interconnection network than a bus. A large-scale parallel machine that provides a shared address space is often referred to as *distributed shared memory (DSM)* machine, since the physical memory is distributed and associated with each individual processor.

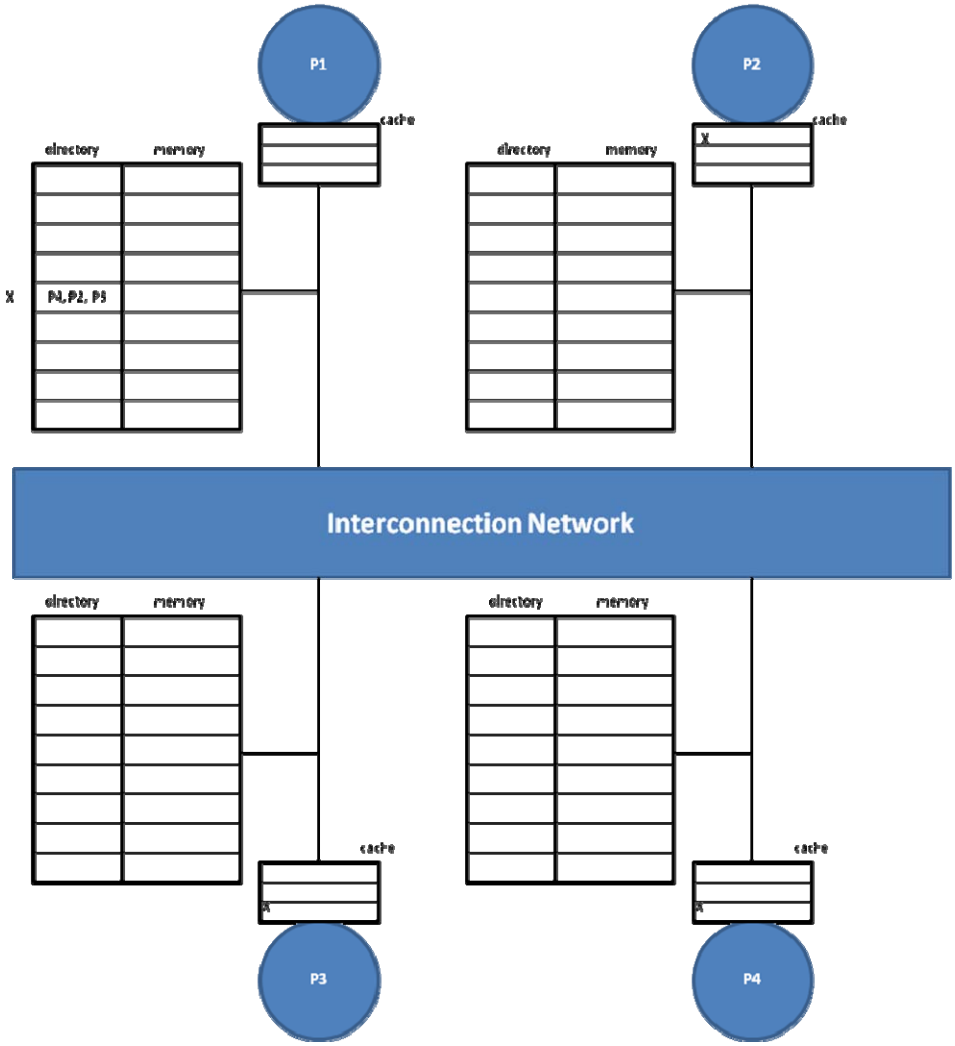


Figure 12.39: DSM Multiprocessor with directory based coherence

The cache coherence mechanisms we discussed in Section 12.9 assume a broadcast medium, namely, a shared bus so that a change to a memory location is seen simultaneously by all the caches of the respective processors. With an arbitrary interconnection network such as a tree, a ring, or a mesh (see Figure 12.35), there is no longer a broadcast medium. The communication is point-to-point. Therefore, some other mechanism is needed to implement cache coherence. Such large-scale shared memory multiprocessors use a *directory-based scheme* to implement cache coherence.

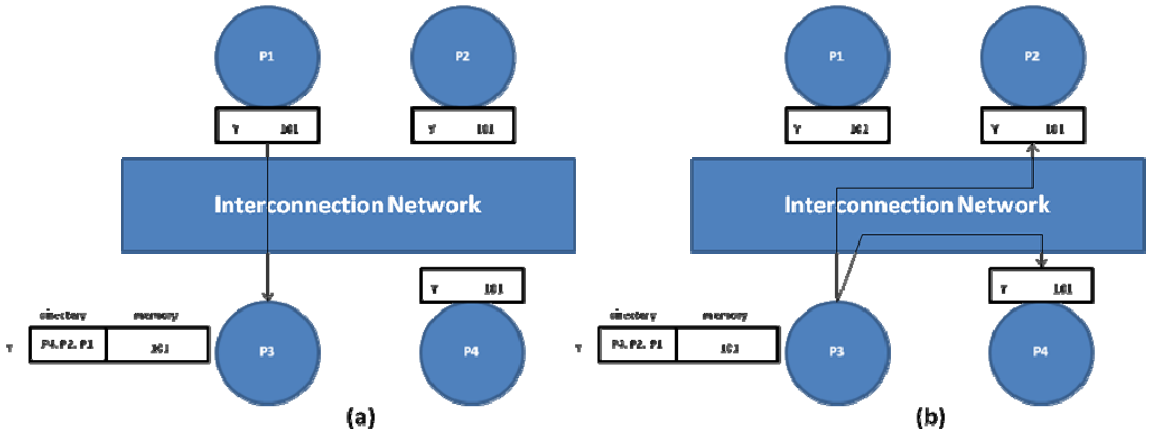
The idea is quite simple. Shared memory is already physically distributed; simply associate a directory along with each piece of the shared memory. There is a directory entry for each memory location. This entry contains the processors that currently have this memory location encached. The cache coherence algorithm may be invalidation or update based. The directory has the book-keeping information for sending either the invalidations or updates. Figure 12.39 shows a directory-based DSM organization. Location X is currently encaches in P2, P3, and P4. If P4 wishes to write to location X, then the directory associated with X sends an invalidation to P2 and P3, before P4 is allowed to write to X. Essentially, the directory orders the access to memory locations that it is responsible ensuring that the values seen by the processors are coherent.

Example 17:

In a 4-processor DSM similar to that shown in Figure 12.39, memory location Y is in physical memory of P3. Currently P1, P2, P4 have a copy of Y in their respective caches. The current values in Y is 101. P1 wishes to write the value 108 to Y. Explain the sequence of steps before the value of Y changes to 108. Assume that the cache write policy is write-back.

Answer:

a) Since memory location Y is in P3, P1 sends a request through the interconnection network to P3 and asks write permission for y (Figure 12.40-(a)).
b) P3 notices that P2 and P4 have copies of Y in their caches by looking up directory entry for Y. It sends invalidation request for Y to P2 and P4 (Figure 12.40-(b)).
c) P2 and P4 invalidate their respective cache entry for Y and send back acknowledgements to P3 via the interconnection network. P3 removes P2 and P4 from the directory entry for Y (Figure 12.40-(c)).
d) P3 sends write permission to P1 (Figure 12.40-(d)).
e) P1 writes 108 in the cache entry for Y (Figure 12.40-(e)).



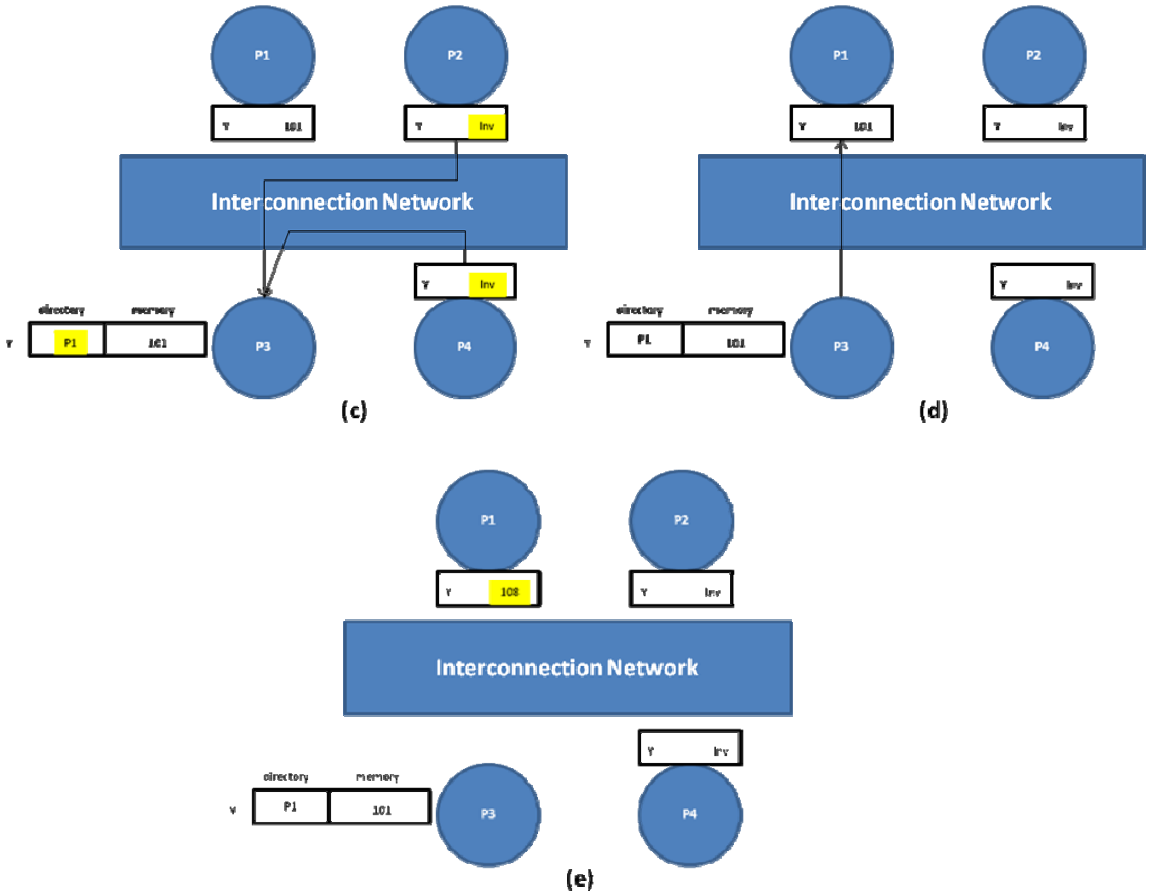


Figure 12.40: Sequence of steps for Example 17

12.10.2.5 Memory Consistency Model and Cache Coherence

A cache-coherent multiprocessor ensures that once a memory location (that may be currently cached) is modified the value will propagate to all the cached copies and the memory as well. One would expect that there will be some delay before all the cached copies have been updated with the new value that has just been written. This begs the question what is the view from the programmer's perspective? *Memory consistency model* is the contract between the programmer and the memory system for defining this view. We have seen repeatedly in this textbook that system design is all about establishing a contract between the hardware and the software. For example, ISA is a contract between the compiler writer and the processor architect. Once the ISA is set, the implementer of the ISA can exercise his/her choice in the hardware realization of the ISA. Memory consistency model is a similar contract between the programmer and the memory system architect.

Let us motivate this with an example. The following is the execution history on processors P1 and P2 of an SMP. The memory is shared by definition but the processor registers are private to each processor.

Initially,

```
                Mem[X] = 0

                Processor P1                Processor P2

Time 0:         R1 <- 1
Time 1:         Mem[X] <- R1                R2 <- 0
Time 2:         ....            ....
Time 3:         ....                        R2 <- Mem[X]
Time 4:         ....                        ....
```

What would you expect the contents of R2 on P2 to be at time 4? The intuitive answer is 1. However, this really depends on the implementation of the memory system. Let us say that each of the instructions shown in the execution history is atomic with respect to that processor. Thus at time 1, processor P1 has written a value of 1 into Mem[X]. What this means as far as P1 is concerned is that if it tries to read Mem[X] after time 1, the memory system guarantees that it will see the value 1 in it. However, what guarantee does the memory system give for values to the same memory location when accessed from other processors? Is it legal for the memory system to return a value 0 for the access at time 3 on P2? The cache coherence mechanism guarantees that both the caches on P1 and P2 as well as the memory will *eventually* contain the value 1 for Mem[x]. But it does not specify *when* the values will become consistent. The memory consistency model answers the *when* question.

An intuitive memory consistency model proposed by Leslie Lamport is called *sequential consistency (SC)*. In this model, memory reads and writes are atomic with respect to the *entire system*. Thus, if we say that P1 has written to Mem[X] at time 1, then the new value for this memory location is *visible* to *all* the processors, henceforth. Thus, when P2 reads Mem[X] at time 3 it is also guaranteed by the memory system that it will see the value 1. As far as the programmer is concerned, this is all the detail about the memory system that he/she needs to write correct programs. The implementation of the memory system with caches and the details of the cache coherence mechanisms are irrelevant to the programmer. This is akin to the separation between architecture and implementation that we have seen in the context of processor ISA design.

Let us give a more precise definition of the SC memory model. It basically says that the effects of the memory accesses from an individual processor will be exactly the same as if there are no other processors accessing the memory. That is, each memory access (read or write) will be atomic. One would expect this behavior from a uniprocessor and therefore it is natural to expect this from a multiprocessor as well. As far as memory accesses from different processors, the model is saying that the observed effect will be an arbitrary interleaving of these individually atomic memory accesses emanating from the different processors.

One way to visualize the SC memory model, is to think of memory accesses from different processors as the cards dealt to a number of players in a card game. You may have seen a card sharp take two splits of a card deck and do a shuffle merge as shown in

the adjoining picture, while preserving the original order of the cards in the two splits. The SC memory model does an *n-way* shuffle merge of all the memory accesses coming from the *n* processors that comprise the multiprocessor.

Returning to the example above, we showed a particular post-mortem execution history in which the access to Mem[X] from P2 *happened after* the access to Mem[X] from P1. The processors are asynchronous with respect to each other. Therefore, another run of the same program could have an execution in which the accesses are reversed. In this case, the access by P2 to Mem[X] would return a value of 0. In other words, for the above program, the SC model would allow the memory system to return a 0 or a 1 for the access by P2 to Mem[X].

One can see that the SC memory model could result in parallel programs that have data races (see Section 12.2.3 for a discussion of data races). Fortunately, this does not affect correct parallel program development as long as the programmer uses synchronization primitives as discussed in earlier sections to coordinate the activities of the threads that comprise an application. In fact, as we have seen in earlier sections, the application programmer is completely oblivious to the memory consistency model since he/she is programming using libraries such as *pthreads*. Just as the ISA is a contract between the compiler writer and the architect, the memory consistency model is a contract between the library writer and the system architect.

The SC memory model says nothing about how the processors may coordinate their activities using synchronization primitives to eliminate such data races. As we have discussed in earlier sections, the system may provide synchronization primitives in hardware or software for the purposes of coordination among the threads. Therefore, it is possible to include such synchronization primitives along with the normal memory accesses to specify the contract between hardware and software. This would give more implementation choices for the system architect to optimize the performance of the memory system. The area of memory consistency models for shared memory systems has been a fertile area of research in the late 80's and the early 90's resulting in several doctoral dissertations. Detailed treatment of this topic is beyond the scope of this textbook. The interested reader is referred to advanced computer architecture textbooks for a more complete exposure to this exciting topic.

12.10.3 The Road Ahead: Multi- and Many-core Architectures

The road ahead is exciting for parallel computing. The new millennium has ushered in the age of multi-core processors. The name multi-core comes from the simple fact that the chip now consists of several independently clocked processing cores that constitute the processor. Moore's law fundamentally predicts increase in chip density with time. Thus far, this increase in chip density has been exploited to increase processor

performance. Several modern processors including AMD Phenom II, IBM Power5, Intel Pentium D and Xeon-MP, and Sun T1 have embraced the multi-core technology.

However, with the increase in processor performance the power consumption of single-chip processors has also been increasing steadily (see Section 5.15.2). The power consumption is directly proportional to the clock frequency of the processor. This trend is forcing processor architects to focus attention on ways to reduce power consumption while increasing chip density. The answer is to have multiple processing cores on a single-chip, each of which may be clocked at a lower clock frequency. The basic strategy for conserving power would then involve selectively turning off the power to parts of the chip. In other words, power consumption consideration is forcing us to choose parallelism over faster uniprocessors. With single-chip processors now containing multiple independent processing cores, parallel computing is no longer an option but a must as we move forward. The next step in the evolution of single-chip processors is *many-core* processors, wherein a single-chip may contain 100's or even 1000's of cores. Each of these processing cores will likely be a very simple processor, not that different from the basic pipelined processor we discussed in Chapter 5. Thus, even though we did not get into the micro-architectural intricacies of modern processors in the earlier chapters, the simple implementations discussed in these chapters may become more relevant with the many-core processors of tomorrow.

One might be tempted to think that a multi-core architecture is no different from shrink-wrapping an SMP into a single chip; and that a many-core architecture is similarly a shrink-wrapped version of a large-scale multiprocessor with 100's or 1000's of PEs. However, it is not business as usual, considering the range of issues that requires a radical rethinking including electrical engineering, programming paradigm, and resource management at the system level.

At the electrical engineering level, we already mentioned the need for reducing the power dissipation. The reason for conserving power is a pragmatic matter. It is just impossible to cool a processor that is smaller than the size of a penny but consumes several hundred watts of power. Another concern is distribution of electrical signals, especially the clock, on the chip. Normally, one would think that placing a 1 or 0 on a wire results in this signal being propagated as is to the recipients. This is safe assumption as long as there is sufficient time for the signal to travel to the recipient on the wire, referred to as the wire delay. We have seen in an earlier chapter (see Chapter 3) that the wire delay is a function of the resistance and capacitance of the electrical wire. As clock speed increases, the wire delay may approach the clock cycle time causing a wire inside a chip to behave like a transmission line, i.e., the signal strength deteriorates with distance. The upshot is that 1 may appear like a 0 at the destination, and vice versa leading to erroneous operation. Consequently, chip designers have to rethink and re-evaluate some basic assumptions about digital signals as the chip density increases in the multi- and many-core era. This is giving rise to new trends in integrated circuit (IC) design, namely, three dimensional design popularly referred to as *3D ICs*. Basically, to reduce the wire length, the chips are being designed with active elements (transistors) in several parallel planes in the z-dimension. Wires, which hitherto were confined to two dimensions, will now be run in

three dimensions, thus giving an opportunity to reduce the wire delay between active elements. A 3D chip is akin to building skyscrapers! Mainstream general-purpose processors are yet to embrace this new technology but it is just a matter of time.

At the architecture level, there are several differences between an SMP and a multi-core processor. In an SMP, at the hardware level, the only shared hardware resource is the bus. On the other hand, in a multi-core processor, several hardware resources may be common to all the cores. For example, we showed the memory hierarchy of AMD Barcelona chip in Chapter 9 (see Figure 9.45). The on-chip L3 cache is shared across all the four cores on the chip. There could be other shared resources such as I/O and memory buses coming out of the chip. It is architectural issue to schedule the efficient use of such shared resources.

At the programming level, there are important differences in the way we think about a parallel program running on a multi-core versus on an SMP. A good rule of thumb in designing parallel programs is to ensure that there is a good balance between computation and communication. In an SMP, threads should be assigned a significant amount of work before they are required to communicate with other threads that are running on other processors. That is, the computations have to be *coarse-grained* to amortize for the cost of inter-processor communication. Due to the proximity of the PEs in a multi-core processor, it is conceivable to achieve *finer-grain* parallelism than would be feasible in an SMP.

At the operating system level, there is a fundamental difference between an SMP and a multi-core processor. Scheduling threads of a multithreaded application on a multi-core versus an SMP requires rethinking to take full advantage of the shared hardware resources between the cores. In an SMP, each processor has an independent image of the operating system. In a multi-core processor, it is advantageous for the cores to share the same image of the operating system. The OS designers have to rethink what operating systems data structures should be shared across the cores, and what should be kept independent.

Many-core processors may bring forth a whole new set of problems at all of the above levels and add new ones such as coping with partial software and hardware failures.

12.11 Summary

In this chapter, we covered key concepts in parallel programming with threads, the operating system support for threads, and the architectural assists for threads. We also reviewed advanced topics in operating systems and parallel architectures.

The three things that an application programmer has to worry about in writing threaded parallel programs are thread creation/termination, data sharing among threads, and synchronization among the threads. Section 12.3 gives a summary of the thread function calls and Table 12.2 gives the vocabulary the reader should be familiar with in the context of developing threaded programs. Section 12.6 gives a summary of the important thread programming API calls supported by the *pthreads* library.

In discussing implementation of threads, we covered the possibility of threads being implemented above the operating system as a user level library with minimal support from the operating system itself in Section 12.7.1. Most modern operating systems such as Linux and Microsoft XP and Vista support threads as a basic unit of CPU scheduling. In this case, the operating system implements the functionality expected at the programming level. We covered kernel level threads in Section 12.7.2 as well as an example of how threads are managed in the Sun Solaris operating system in Section 12.7.3.

The fundamental architectural assist needed for supporting threads is an atomic read-modify-write memory operation. We introduced Test-And-Set instruction in Section 12.8, and showed how using this instruction, it is possible to implement higher-level synchronization support in the operating system. In order to support data sharing among the processors, the cache coherence problem needs to be solved which we discussed in Section 12.9.

We considered advanced topics in operating systems and architecture as they pertain to multiprocessors in Section 12.10. In particular, we introduced the reader to a formal treatment of deadlocks, sophisticated synchronization constructs such as monitor, advanced scheduling techniques, and classic problems in concurrency and synchronization (see Section 12.10.1). In Section 12.10.2, we presented a taxonomy of parallel architectures (SISD, SIMD, MISD, and MIMD) and discussed in depth the difference between message-passing style and shared memory style MIMD architectures.

The hardware and software issues associated with multiprocessors and multithreaded programs are intriguing and deep. We have introduced the reader to a number of exciting topics in this area. We have barely scratched the surface of some these issues in this Chapter. We hope we have raised the reader's curiosity level enough through this Chapter to take him/her through further exploration of these issues in future courses. A more in-depth treatment of some of the topics covered in this chapter may be found in the textbook by Culler, et al.[13]

12.12 Historical Perspective

Parallel computing and multiprocessors have been of interest to computer scientists and electrical engineers from the very early days of computing.

In Chapter 5, we saw that pipelined processor design exploits instruction-level parallelism or ILP. In contrast, multiprocessors exploit a different kind of parallelism, namely, thread level parallelism or TLP. Exploitation of ILP does not require the end user having to do anything different – he/she continues to write sequential programs. The compiler in concert with the architect does all the magic to exploit the ILP in the sequential program. However, exploitation of TLP requires more work. Either the program has to be written as an explicitly parallel program that uses multiple threads as

[13] Culler, Singh, and Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann.

developed in this chapter, or the sequential program has to be converted automatically into a multithreaded parallel program. For the latter approach, a sequential programming language such as FORTRAN is extended with directives that the programmer inserts to indicate opportunities for automatic parallelization. The compiler exploits such directives to parallelize the original sequential program. This approach was quite popular in the early days of parallel computing but met with diminishing returns since its applicability is restricted usually to exploiting loop-level parallelism and not function parallelism. Just as a point of clarification, loop-level parallelism notices that successive iterations of a "for" loop in a program are independent of one another and turns every iteration or groups of iterations into a parallel thread. Function parallelism or task parallelism deals with conceptual units of work that the programmer has deemed to be parallel (similar to the vision example used in developing the thread programming constructs in this chapter).

One can easily understand the lure of harnessing parallel resources to get more work done. It would appear that an application that achieves a given single processor performance could in theory achieve an N-fold improvement in performance if parallelized. However, we know from Chapter 5 that Amdahl's law limits this linear increase in performance due to the inherent serial component in the application. Further, there is usually a lag between the performance of a single processor and that of the individual processors that comprise a parallel machine. This is because constructing a parallel machine is not simply a matter of replacing the existing processors with faster ones as soon as they become available. Moore's law (Please see Section 3.1) has been giving us increased single processor performance almost continuously for the past 30 years. Thus, a parallel machine becomes quickly outdated as the next generation of higher-performance microprocessors hit the market. For example, a modern day notebook computer costing a few thousand dollars has more processing power than a multi-million dollar Cray machine of the 70's and 80's. The obsolescence factor has been the main reason software for parallel machines has not seen the same rapid growth as that for uniprocessors.

Parallel architectures were a high-end niche market reserved for applications that demanded such high performance, mainly from the scientific and engineering domains. Examples of companies and the parallel machines marketed by them include TMC (connection machine line of parallel machines starting with CM-1 to CM-5); Maspar (MP-1 and MP-2); Sequent (Symmetry); BBN (Butterfly); Kendall Square Research (KSR-1 and KSR-2); SGI (Origin line of machines and currently Altix); and IBM (SP line of machines). Typical characteristics of such machines include either an off-the-shelf processor (e.g., TMC's CM-5 used Sun SPARC, and SGI's Origin series used MIPS), or a custom-built processor (e.g., KSR-1, CM-1, CM-2, MP-1, and MP-2); a proprietary interconnection network, and glue logic relevant to the taxonomical style of the architecture. The interconnection network was a key component of such architectures since efficient data sharing and synchronization among the processors were dependent on capabilities in the interconnection network.

The advent of the powerful single-chip microprocessor of the 90's (dubbed "Killer micros") turned out to be a disruptive technology that shook up the high-performance computing market. Once the single-chip microprocessor performance surpassed that of a custom crafted processor for a parallel machine, the economic viability of constructing such parallel machines came into question. While a few have survived by reinventing themselves, many parallel computing vendors that thrived in the niche market disappeared (for example, TMC and Maspar). In parallel with the advance of single-chip microprocessor performance, local area network technology was advancing at a rapid pace as well. We will visit the evolution of local area networks (LAN) in Chapter 13, but suffice it to say here that with advent of switched Gigabit Ethernet (see Section 13.11), the need for proprietary interconnection networks to connect processors of a parallel machine became less relevant. This breakthrough in LAN technology spurred a new class of parallel machines, namely, clusters. A cluster is essentially a set of compute nodes interconnected by off-the-shelf LAN technology. This has been the workhorse for high-performance computing since the mid to late 90's to the present day. Clusters promote a message-passing style of programming and as we said earlier, the MPI communication library has become a *de facto* standard for programming on clusters. What is inside a compute node changes of course with advances in technology. For example, at present it is not uncommon to have each computing node as an *n*-way SMP, where *n* may be 2, 4, 8, or 16 depending on the vendor. Further, each processor inside an SMP may be a hardware multithreaded multi-core processor. This gives rise to a mixed parallel programming model: shared memory style programming within a node and message-passing style programming across the nodes.

12.13 Review Questions

1. Compare and contrast processes and threads

2. Where does a thread start executing?

3. When does a thread terminate?

4. How many threads can acquire a nutex lock at the same time.

5. Does a condition variable allow a thread to wait conditionally or unconditionally?

6. Define deadlock and explain how it can happen and how it can be prevented.

7. What problems could be encountered with the following construct

```
if(state == BUSY)
 pthread_cond_wait(c, m);
state = BUSY;
```

8. Compare and contrast the contents of a PCB and a TCB.

9. Is there any point to using user threads on a system which is only scheduling processes and not threads. Will the performance be improved?

10. User level thread synchronization in a uniprocessor (select one from the following to complete the sentence)

1. needs no special hardware support since turning off interrupts will suffice
2. needs some flavor of a *read modify write* instruction
3. can be implemented simply by using load/store instruction

11. Ensuring that all the threads of a given process share an address space in an SMP is (select one from the following to complete the sentence)

1. impossible
2. trivially achieved since the page table is in shared memory
3. achieved by careful replication of the page table by the operating system for each thread
4. achieved by the hardware providing cache coherence

12. Keeping the TLBs consistent in an SMP (select one from the following to complete the sentence)

1. is the responsibility of the user program
2. is the responsibility of the hardware
3. is the responsibility of the operating system
4. is impossible

13. Select all choices that apply regarding threads created in the same address space.

1. they share code
2. they share global data
3. they share the stack
4. they share the heap

14. From the following sentences regarding threads, select the ones that are True.

1. An operating system that provides no support for threads will block the entire process if one of the (user level) threads in that process makes a blocking system call.

2. In pthreads, a thread that does a pthreadcondwait will always block.

3. In pthreads, a thread that does a pthreadmutexlock will always block.

4. In Solaris, all user level threads in the same process compete equally for CPU resources.

5. All the threads within a single process in Solaris share the same page table.

15. What is the big advantage of kernel threads in Sun Solaris?

    1. Allow for non-blocking upcalls to upper software layers
    2. Allow O/S to schedule different threads on different processors
    3. Abstraction of user level threads into LWP's

16. Implement the monitor shown in Example 16 in C using pthreads library.

17. Implement the solution shown in Example 16 using Java.

18. Write a solution to the readers-writers problem using mutual exclusion locks that gives priority to writers.

19. Write a solution to the readers-writers problem using mutual exclusion locks that is fair to both the readers and writers (Hint: use an FCFS discipline in the solution).

20. Repeat Exercises 18 and 19 using monitors.

21. Repeat Exercises 18 and 19 using Java.

22. Implement Example 16 using counting semaphores, allowing at most n simultaneous readers or 1 writer into the database at any time.

23. Figure 12.34 gives a monitor solution for the Dining Philosophers problem. Re-implement the solution using mutual exclusion locks.