3 Shared-Memory Programming

3.1 Introduction

Shared-memory parallel programming is very similar with operating systems programming and general multiprogramming. Indeed, shared-memory programming is done through minor extensions to existing programming languages, operating systems, and code libraries.

In this chapter we examine the general model of applications that best take advantage of the power of shared-memory multiprocessors systems such as the Sequent Symmetry and Encore Multimax. We show that these systems are best suited for the MIMD paradigm where a supervisor task controls a number of subordinate tasks. This is the supervisor/worker model which gives rise to a client-server structure within the application's code.

Next, we study the DYNIX parallel process model used in the Sequent Computer Systems' machines. DYNIX, an extension of UNIX, provides a number of toolbox routines to handle task creation and synchronisation. Examples are given in C, but these routines are available for other programming languages.

How are processes synchronised and storage locked in these systems? We generalise the Sequent model and introduce analysis technique based on timing diagrams. This technique can be used to analyse any highly concurrent application to assure that it is safe, fair, and live.

3.2 A Shared-memory process model

A theoretical model of shared-memory multicomputers called PRAM (Parallel Random Access Memory) overlooks details such as cache coherence and synchronisation. In the PRAM model, algorithms are assumed to run without interference as long as only one memory access is permitted at a time. We say that PRAM guarantees atomic access to data located in shared memory. Access is considered to be atomic if only one processor is allowed to read/write from/to shared memory at a time. Atomic access will be assumed in this chapter, regardless of how it is achieved.

However, task creation and access synchronisation can be costly. In addition, shared-memory computers typically support a limited number of processors. In general, shared-memory computers manage only a few hundred tasks simultaneously, and consist of fewer than 100 processors.

3.2.1 Multitasks versus microtasks

At the large-grained level, PRAM computers can provide traditional time-sharing as in UNIX. Each time a UNIX process is initiated, idle processors are supplied to run the new process. If the system is heavily loaded, the processor with the least amount of work is assigned the new process. This form of dynamic scheduling is typically nonoptimal, but it is fast and simple. When spread over a number of user tasks, load balancing is possible in-the-large, and the operating system does not need to expend much effort to match processes to processors. (The jobs are simply entered in a queue, and the next available processor takes its next job from the head of the queue.)

A major advantage of PRAM is its suitability for virtual memory and traditional input and output. Most of what we know about operating systems applies to this model. Hence, page tables and virtual memory paging hardware, disk I/O, and networking can be implemented as in any other bus-structured system.

The large-grained PRAM model is suitable for multitasking applications where the cost of task creation is rather large. Such tasks are often called heavy-weight tasks because of their high overhead. A heavy-weight task in a multitasking system such as UNIX consists of page tables, memory, and file descriptors in addition to program code and data.

The heavy-weight PRAM model will be called **multitasking** here. It is implemented in UNIX by invocation of **fork**, **exec**, and other related UNIX commands. Multitasking is best suited for heterogeneous tasks, e.g., multiprogramming as in classical time-sharing.

At the fine-grained level, the PRAM model weakens because the high cost of process creation and access synchronisation must be considered. For performance reasons, light-weight processes are needed. MACH, a UNIX look-alike, implements lightweight processes, but most UNIX variants also support less costly microtasking.

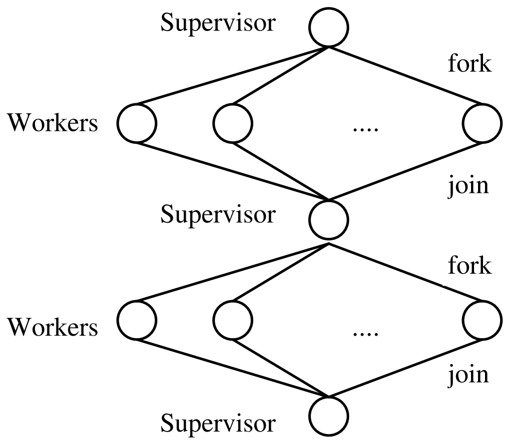


Figure 3-1 Series of fork-join tasks model supervisor/worker paradigm used in most parallel applications on shared-memory multiprocessors

Microtasking permits a finer-grained parallelism. In fact, microtasking makes parallelism within a single application practical, but on PRAM machines, microtasking is best suited to homogeneous tasks (see Figure 3-1). In this model, an application is a series of **fork-join fans**. A fan is a collection of N worker tasks where one task is the supervisor and N-1 tasks are workers. In UNIX, a fan is created by forking processes as shown in Figure 3-1.

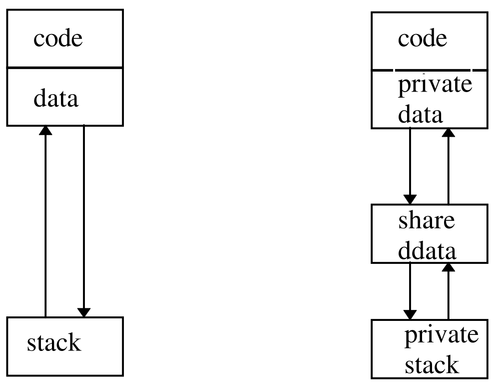
3.2.2 A UNIX shared-memory model

In the standard UNIX model, the address space of an executing process has three segments called the text, data, and stack. The text is where the binary code to be executed is stored; the data segment is where the program's data are stored; and the stack is where activation records and dynamic data are stored. A simplified model of this is shown in Figure 3-2(a).

The data and stack segments expand and contract as the program executes. Therefore, a gap is purposely left in between the data and stack segments as shown in Figure 3-2(a).

Processes (multitasks) are assumed to be mutually independent and do not share addresses. The code of each serial process is allowed to access data in its own data and stack segments, only. Any attempt by another process to access other segments is trapped as a fatal error by UNIX. This is good for multiprogramming, but poses an obstacle to parallel programs.

In most UNIX systems, a new process is created by forking an existing process. Typically, this



(a) serial process (b) parallel process

Figure 3-2 (a) UNIX serial process model;
(b) modified process model used by Sequent DYNIX

means the running program is interrupted, swapped to disk, and two copies swapped back into memory. The two copies are identical in every way, including their entire code. It is up to the programmer to interrogate the process ID returned by the fork routine to determine which process the code is executing.

This swapping is costly, as mentioned before. The first time a program image is copied into memory to be executed, UNIX must set up virtual memory page tables and file buffers, for example. The initialisation takes memory as well as time.

To alleviate the data access restrictions and overhead associated with multitasks in UNIX, Sequent Computer Corporation modified UNIX into DYNIX. The process model of DYNIX is shown in Figure 3-2(b).

A parallel process in DYNIX is identical to the serial model of UNIX plus an additional shared-memory data segment. This segment is allowed to grow as shown in Figure 3-2(b), and so it is placed in the hole between the private data and stack segments.

When a parallel process is created in DYNIX, the shared-memory address space as shown in Figure 3-2(b) is created. Subsequent microtasks spawned from the parent task by the DYNIX **m_fork** routine copy all but the shared data segment into their address spaces. Thus, there is only one copy of the shared data segment among the worker tasks. An example for two tasks is shown in Figure 3-3.

This modification requires extensions to UNIX, languages such as C, and a library of routines to support the model. For example, in C, two keywords must be added to the language to indicate whether a data object is private or shared, e.g., **shared int x** means that x is shared, whereas **private int x** means x is a private variable.

Two new types are needed for locking and setting up barriers. Variables of type **slock_t** are used to set and clear **access locks**, and variables of type **sbarrier_t** are used to set and clear **barriers**. The appropriate operations for objects of these types are listed below.

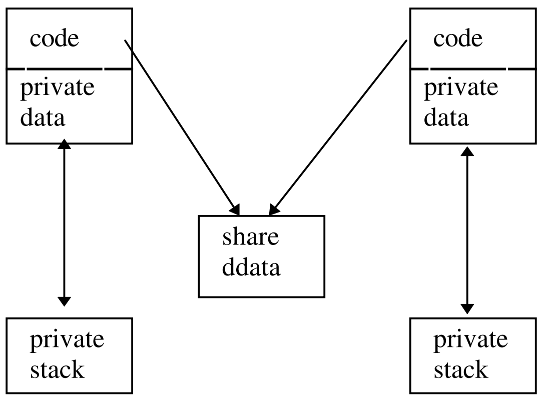


Figure 3-3 Two microtasks created by m_Fork in DYNIX share some data

3.2.2.1 Multitasking

Operations on variables of type slock_t:

s_init_lock():	initialise a lock variable
s_lock():	set the lock
s_unlock():	clear the lock

Operations on variables of type sbarrier_t:

s_init_barrier():	initialise the barrier variable
s_wait_barrier():	wait for all multitasks to reach barrier

The usual **fork()** and **cpus_online()** routines are used to create a multitask and query the system for the number of processors, respectively.

3.2.2.2 Microtasking

Sequent's microtasking library has the following key functions:

m_set_procs(p): By calling m_set_procs with argument p, the parent process initialises to value p a shared variable that controls the number of processes created by a subsequent call to m_fork. The value of p cannot exceed the number of physical processors in the system minus one. The function also initialises barriers and locks.

m_fork(name [,arg,...]): The parent process creates a number of child processes, then the parent process and the child processes begin executing function name with the arguments (if any) also specified by the call to m_fork. After all the processes (the parent and all the children) have completed execution of function name, the parent process resumes execution with the code after m_fork, while the child processes busy wait until the next call to m fork. Therefore, the first call to m fork is more expensive than subsequent calls, because only the first call entails process creation.

m_get_myid: A process calls function m_get_myid to get its unique process number. If the total number of active processes is p, then the process number of the parent is 0, while the process numbers of the child processes range from 1 to p-l.

m_get_numprocs: Function m_get_numprocs returns the number of processes executing in parallel. Given the total number of processes and its own process number, a process can determine which portion of a computation is its responsibility.

m_kill_procs: Function m_kill_procs kills the child processes created by the first call to m_fork.

Operations on variables of type slock_t:

m_lock():	set the lock
m_unlock():	clear the lock

m_lock, m_unlock: Functions m_lock and m_unlock ensure mutually exclusive execution of the code that the two calls surround. Once a process has entered a block of code delimited by m_lock and m_unlock, no other process may enter until the first process has left.

Operations on variables of type sbarrier_t:

m_sync():	wait for all microtasks to arrive
m_single():	wait for supervisor thread to call m_multi()
m_multi():	resume all workers in parallel

These routines are illustrated in the following example. The program creates one supervisor, and nprocs-1 workers. The code for each worker is given as function work(). The example illustrates both methods of synchronising access to a shared variable.

```
/* Global declarations */
shared int x;
shared struct y_struct {
    int state;
    slock_t lp;         /* primitive lock */
    } y;

/* Supervisor microtask */
main() {
    s_init_lock (&y.lp);        /*initialize primitive lock */
    m_set_procs (nprocs);       /*initialize nprocs-1 workers*/
    m_fork (work);              /*create and run nprocs-1 microtasks*/
    m_kill_procs ();            /*done, so kill worker microtasks*/
    }
```

```
/* Worker microtask */
work() {
    /* illustrates microtask locking */
    m_lock();             /* begin mutual exclusive access to x */
    x++;                  /* increment shared variable x */
    printf ("worker");    /* just a test message */
    m_unlock();           /* end of critical section */

    /* illustrates a barrier */
    m_single();              /* all workers wait here until m_multi */
    printf ("Supervisor");   /* only supervisor executes this */
    m_multi();               /* resume all workers ... */

    /* illustrates multitask locking */
    s_lock (&y.lp);          /* lock structure y */
    y.state = 0;             /* change the structure */
    printf ("structure");    /* another test message */
    s_unlock (&y.lp);        /* unlock structure y */
    }
```

A **critical section** is a piece of code that accesses a shared variable. In this example, the shared variables are x and y. Note the locks in each case. These locks provide **mutual exclusion**. That is, only one task is permitted to access each shared variable at a time. Later, we look at more elaborate locks which allow multiple tasks to simultaneously access a shared variable. Most programming problems arising in shared-memory parallel programming can be solved with this handful of routines.

This is a classic example of the supervisor/worker paradigm. Microtasks are set up to execute homogeneous tasks. The function work() is identical in all microtasks, and only the data varies. This example implements one fan per Figure 3-1. More extensive applications would implement perhaps many fans in this fashion.

3.2.2.3 Sample program

We present the sample program of the introduction chapter written in Sequent C. The program computes an approximation to π by using numerical integration to find the area under the curve $4/(1 + x^2)$ between 0 and 1 (Listing I3-1). The program begins with directives to the compiler to include standard header files. The header files stored in *microtask.h* and *parallel.h* contain information on the parallel library functions.

The double-precision floating-point variable *pi* will contain the approximation to π. Since all processes will need access to *pi* in order to store their subtotals, *pi* is declared resident in shared memory.

The main procedure parses the command line, calls *m_set_procs* to set the program's level of parallelism, initialises *pi*, and calls *m_fork* to initiate the parallel computation of *pi*. After all processes have finished executing function *computepi*, the parent process prints the current value of variable *pi*, kills the child processes, and terminates.

In function *computepi*, every process calls function *m_get_myid* to find its unique identifying number in the range *0...numprocs*. By setting the initial value of the loop index *i* to *id* and making the loop increment value *numprocs*, the processes as a whole cover every value of *i* in the range *0...intervals*-l exactly once.

Once every process has computed its share of the sum, it must add its subtotal to *pi*. Because the statement *pi += localsum* is not an atomic operation, the processes must mutually exclude each other when executing it. They create the necessary critical section through the calls to functions *m_lock* and *m_unlock*.

```
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

shared double pi; /* Approximation to pi */
```

```
main (argc, argv)
    int argc; char *argv[];
{
    void computepi();
    int intervals;      /* Number of intervals*/
    int numprocs;       /* Number of processes to be forked*/
    numprocs = atoi (argv[1]);
    intervals = atoi (argv[2]);
    m_set_procs (numprocs);
    pi = 0.0;
    m_fork (computepi, numprocs, intervals);
    printf ("Estimation of pi is %14.12f.\n", pi);
    m_kill_procs ();
    return();
}

void computepi (numprocs, intervals)
    int numprocs, intervals;
{
    int id;              /* Index of this process */
    int i;
    double localsum, width, x;
    id = m_get_myid ();
    localsum = 0.0;
    width = 1.0/intervals;
    for (i = id; i < intervals; i += numprocs){
      x = (i+0.5)*width;
      localsum += (4.0/(1.0+x*x));
    }
    localsum *= width;
    m_lock ();
    pi += localsum;
    m_unlock ();
}
```

Listing 3-1 Sequent C program to compute π using numerical integration.

3.3 Analysis of shared-memory programs

The toolbox routines for parallel DYNIX make parallel programming a shared-memory computer rather similar to programming a real-time process control program in a traditional serial computer. A combination of low-level locks along with cache memories greatly simplifies application development, but all is not so simple.

First, these toolbox routines can be abused. For example, one processor can set a lock, and cause another processor to saturate the bus by **spin-waiting** for the lock to clear. This will cause the entire system to slow down to the extent that all tasks run slower than they might on one processor!

Second, simple mutual exclusion is not always the best policy. If N tasks want to read a shared variable, there is no harm in allowing free access. Yet, if one task wants to update the shared variable through write access, and (N-1) tasks want read-only access, mutual exclusion is extremely conservative and time consuming. The multiple readers problem is solved by designing a more complex but more efficient lock from the mutual exclusion building blocks.

Finally, locks raise a number of other interesting questions about the correctness of parallel programs just as they do in traditional operating systems theory. What we need are analysis tools to analyse the application code produced using the toolbox routines.

3.3.1 Safe, live, and fair

A **lock** is a mechanism for enforcing a policy for sharing access to data. A **policy** is a set of rules that define allowable values for shared variables. In most cases, we are concerned with two possible classes of states for shared variables: (1) the class of deterministic values, and (2) the class of nondeterministic values. The values in these states do not need to be unique.

For example, suppose we look at the four-way stop pattern found on most any country highway (see Figure 3-4). Two roads cross at an intersection consisting of four stop signs. Vehicles must stop and either travel straight ahead, or turn right or left. As long as only one vehicle arrives from any direction, the outcome is predictable (deterministic). But, if two or more automobiles arrive at the same time, the outcome is unpredictable (nondeterministic). The nondeterministic case is the most interesting for our purpose.

Suppose we select a policy that guarantees a deterministic outcome in all cases. This policy is enforced by a rule that says first priority is always given to the automobile on the right during a trajectory. Thus, if vehicles A and B arrive and stop at the intersection at exactly the same time as shown in Figure 3-4(a), automobile B goes first because B is on the right side of truck A.

(a) Safe turn (a) Deadlock

Figure 3-4 The four-way stop analogy illustrates access policies. The intersection (shared data) is shared by vehicles (tasks).

Without the "right side" policy, the outcome of any access to the intersection is nondeterministic. Each time two vehicles arrive at the intersection at exactly the same instant, a nondeterministic event occurs. This is exactly what happens in a shared-memory computer system when two or more tasks attempt to simultaneously change a shared variable. Only one task is allowed to access the variable first, but which one is left up to chance. We say the nondeterministic access is indeterminate because the resultant value of a shared variable after two write accesses will depend on the order of the writes, and the order is nondeterministic. An indeterminate state of a variable is a consequence of nondeterministic access.

We can summarise:

1. **determinism**: access order is always the same no matter how many times the program is run.
2. **nondeterminism**: access order is left up to chance, and so may be different each time the program is run.
3. **indeterminacy**: the result of nondeterminism, where the value of a shared variable is left to chance. Each time the program is run, it is possible that a different value is computed for the shared variable.

Locks can be thought of as mechanisms for carrying out policies that often require serialisation of access. Thus, locks typically force parallel tasks to run in serial order during access to the shared variable. The section of program that accesses a shared variable is called a **critical section**, for obvious reasons.

A **safe policy** is one that enforces deterministic results. A lock is said to be safe if it enforces determinism. Thus, mutual exclusion, while often overly conservative, is a lock that enforces a safe policy because only one task is permitted access at a time. Clearly, safe access is not sufficient in many real applications, however, as we now show.

Suppose we take the four-way stop analogy even further (see Figure 3-4(a)). After automobile B passes through the intersection, suppose a long line of fast sports cars follows it, and because they are on the right side of truck A, they too are allowed to pass through the intersection. According to our policy of "right vehicle first," there is nothing that prevents a slow truck from being stopped forever by much faster sports cars.

A **policy is unfair** if it permits one task to wait indefinitely while other tasks repeatedly access shared data. A **fair lock** guarantees access to all tasks that have requested access, before second or third accesses are granted to any pending tasks. That is, everyone gets access before anyone gets multiple accesses.

We can add a rule to the traffic policy to guarantee fairness: Vehicles are given access to the intersection in the order they arrive, with the exception that vehicles arriving simultaneously follow the "right side goes first" rule.

This seems to solve the problem, but alas, there is one more case to resolve. Figure 3-4(b) shows a rare case where four vehicles arrive at the intersection at exactly the same time. The fair policy states that the vehicles on the right are allowed to go first, but this leads to a circular gridlock. All vehicles wait for the vehicle on their right, and since all vehicles have someone on their right side, they all must wait!

If the vehicles are tasks, then the tasks are **deadlocked**—a state where none of the tasks can continue on because they are blocked by their neighbouring tasks.

We say a **policy is live** if it prevents deadlock. It must be impossible for a deadlock to ever happen for a lock to enforce liveness. Thus, even though the four-way stop gridlock is unlikely to occur, the fact that it can occur means the traffic policy is **deadlock prone**.

We can fix this problem by a number of techniques, but the simplest technique is to number the stop signs 1, 2, 3, 4, and then use these numbers as priorities whenever a deadlock happens. Thus, the vehicle at stop #1 goes first, followed by #2, and so forth. This is an example of a deadlock prevention algorithm.

Again, we can summarise what we have learned from this analogy:

1. **Safe**: A lock is safe if it is impossible for a shared value to take on indeterminate values when two or more tasks compete for access.
2. **Fair**: A lock is fair if it is impossible for one task to starve waiting tasks by repeated accesses.
3. **Live**: A lock is live if it is impossible for the lock to indefinitely postpone access by waiting tasks.
4. **Critical section**: A section of code that accesses a shared variable.

Typically, parallel programs employ simple mutual exclusion to guarantee safe access. This is done by forcing serial execution of a critical section. But, is it enough? The answer is generally "no," because of a number of factors that enter into a good parallel program.

First, a mutual exclusion lock cannot guarantee fairness to waiting tasks, because faster tasks are allowed to get ahead of slower tasks just as we illustrated in the fourway stop analogy.

Second, a mutual exclusion lock may on its own be live, but when used in combination with other mutual exclusion locks, it may cause a **system deadlock**. System deadlocks can occur anytime a system of tasks contain two or more locks. Such occurrences are precisely analogous to the four-way stop gridlock.

Finally, simple mutual exclusion may lead to performance problems as mentioned above.

We need to understand the way locks behave in a shared-memory computer before we can properly understand the behaviour of parallel programs. This means we must look at some specific examples of algorithms for enforcing various synchronisation policies.

3.3.2 Locks that do not work

An **atomic action** is any action within a computer system that is guaranteed to be performed without interruption and without parallelism. Such actions are called atomic because they are **indivisible**, e.g., they cannot be decomposed into smaller actions which can be done at the same time. In serial computers, atomic actions might be single instructions, or memory read/write operations. In a shared-memory multiprocessor, instructions can be executed in parallel. Furthermore, with cache memory being updated behind the programmer's back, it is not always certain that memory read/write operations are done atomically. One of the most difficult aspects of concurrent processing is finding an atomic action to build on.

Cache coherency algorithms guarantee consistency among all copies of a shared-memory variable, so we can model memory access as an atomic action. Similarly, whenever instructions from two or more processors attempt to update a shared variable, we can assume that the cache coherency algorithm will select one processor to go first, while the others wait. That is, the shared variable is treated much like the intersection in the four-way stop analogy by the cache coherency algorithm. So, instructions that perform updates can be thought of as atomic instructions, because parallelism is restricted.

3.3.2.1 Example

A **spin lock** simply tests the lock variable over and over until some other task releases the lock. That is, the waiting task spins on the lock until the lock is cleared. Then, the waiting task sets the lock while inside of the CR. Finally, the lock is cleared, so another task can enter its critical section.

We show an unsafe lock below:

```
while C do;        (* Spin until C is False *)
C := TRUE;         (* Lock(C) so only one access *)
CR;                (* Critical section: access the
                      shared variable *)
C := FALSE;        (* unLock(C) so other tasks can go*)
```

Assume C = FALSE, initially, and task P2 executes first:

```
P2: while C do;
```

Then, suppose task P1 executes a request for access:

```
P1: while C do;
```

Now, suppose we switch back to task P2:

```
P2: C := TRUE;
```

and likewise, task P1 executes the lock setting action:

```
P1: C := TRUE;
```

The cache coherency algorithm will prevent both of these sets from happening at exactly the same time, but one of the tasks will set C:=TRUE followed by the other task setting it to TRUE also. Regardless, the two tasks can now enter their critical sections at the same time. Clearly, the lock is not safe.

The problem with this simple-minded lock is that it lets more than one task into CR at the same time. This is because the lock is tested in one atomic action and set in another atomic action. If the two atomic actions were combined into a single test-and-set operation, we could guarantee safe access. Even so, an atomic test-and-set lock does not guarantee fairness.

This example illustrates a *race condition* within the lock itself. A race condition is a condition that can lead to indeterminacy. In this case, the indeterminacy occurs because we do not know which task sets C to TRUE. We say there is a *race* between the two tasks.

3.3.2.2 Example

The previous lock is unsafe, because it takes two atomic actions to set and test the lock. What happens if we provide an array of locks, and dedicate each array element to one task? The idea is to avoid a race to set and test a single lock. Unfortunately, the analysis shows that the following lock is also faulty, but for a different reason.

```
Flag[me] := TRUE;          (*  Set my flag... *)
while Flag[other] do:      (*  Spin on the other
                               task's flag *)
CR;                        (*  Enter one at a time *)
Flag[me] := FALSE;         (*  Clear the lock *)
```

This lock is "more fair" because it forces ping-ponging between the two tasks; first me=l accesses the data, then other=2 accesses the data, and the pattern repeats assuming the tasks make repeated requests by setting their Flags.

The problem with this improvement is that it introduces a possibility of deadlock! The deadlock state can be reached when both tasks set their flags and then simultaneously spin. When this happens, the two tasks must be terminated because they will never leave the deadlock state.

3.3.3 Locks that work

An obvious solution to the problems encountered above is to use a **simple test-and-set atomic operation** to implement the lock. In shared bus multiprocessors the test&set synchronisation operation is widely accepted to support the implementation of high level language constructs. In its simplest form the test&set operation is based on a simple variable called 'lock' which can have two states: CLOSED and OPEN. test&set is an indivisible, atomic operation on the lock variable defined by the following C program:

```
char *lock;
while (exchange(lock, CLOSED) == CLOSED);
```

Exchange(value1,value2) is usually a CPU instruction meaning that the value of the second parameter (value2) should be exchanged with the value of the memory location given by the first argument (value1) and the latter value should be returned as a result of the instruction. The instruction exchange requires two memory cycles: the first for reading value1 and the second for writing value2. The atomic operation means that during the execution of these two memory cycles the shared bus must not be granted to any other processor in the system.

Indeed, such instructions exist within most processors, so, why is this a poor solution? One of the main problems of implementing synchronisation schemes in cache coherent architectures appears in deciding what happens if the test&set operation was failed, i.e. the lock was in the state CLOSED. Obviously, as the definition of the test&set operation shows, the processor should repeat the operation as long as the lock is CLOSED. This is a form of **busy waiting** which ties up the processor in an idle loop and increases the shared bus traffic and contention. This type of lock that relies on busy waiting is called **spin-lock** and considered as a significant cause of performance degradation when a large number of processes simultaneously use it.

Application of spin-locks in cache based multiprocessors gives rise to a new problem, called **thrashing.** Assume that processor Pi successfully executes a test&set operation on lock. As a result, a dirty copy of lock remains in the cache Ci (Figure 3-5.a). Subsequent test&set operation on lock by Pj causes cache Ci to send a copy of lock to Cj and both copies become shared. Notice that fetching a block from a remote cache requires much longer time than fetching a single byte from the main memory. Moreover, the second phase of test&set should write lock, which entails sending an invalidate command on the shared bus and makes the copy of lock in Cj dirty (Figure 3-5.b). When Pj finishes the test&set operation, a fair bus arbitration policy allocates the shared bus to another processor Pk, which wants to perform test&set on lock, too. Obviously, the same bus operations are required and performed again and finally, Ck contains a dirty copy of lock (Figure 3-5.c). Since the test&set operation of Pj was failed, Pj attempts again to execute test&set. As a result, the block of lock is copied back to Cj, and so on. Cj and Ck play ping-pong with the block of lock as long as Pi remains in state CLOSED.

To avoid this situation spin-locks should be replaced by other types of locks in cache based multiprocessors. One possible candidate is the so-called **snooping lock** which relies on the same hardware support but defined by a different software algorithm:

```
while (exchange(lock, CLOSED) == CLOSED)
    while (*lock == CLOSED);
```

The advantage of this approach comes from the introduction of the simplified second while loop which can be executed on the cached copy of lock without accessing the shared bus since it only reads the lock. The situation after executing the exchange operations by Pj and Pk is identical to that of the spin-lock implementation shown in Figure 3-5.c. However, the continuation is different. When Pj executes the *lock==CLOSED test instruction, the dirty block of lock from Ck is copied to Cj and both blocks become shared. Afterwards the two copies can be read independently and in parallel by Pj and Pk without any access to the common bus. This simple modification of the test&set algorithm eliminates the ping-pong effect but does not solve every problem. When the locking processor releases the lock, the waiting processors compete for executing their exchange operations which results in a temporary bus saturation. The problem is not too serious if large critical sections are applied in the program. However, in the case of short critical sections the bus saturation can become continuous due to the frequent exchange operations.

The solution can be further improved by exchanging the order of the test loop and exchange instruction:

```
for (;;) {
    while (*lock == CLOSED);
    if (exchange(lock, CLOSED) != CLOSED)
        break;
}
```

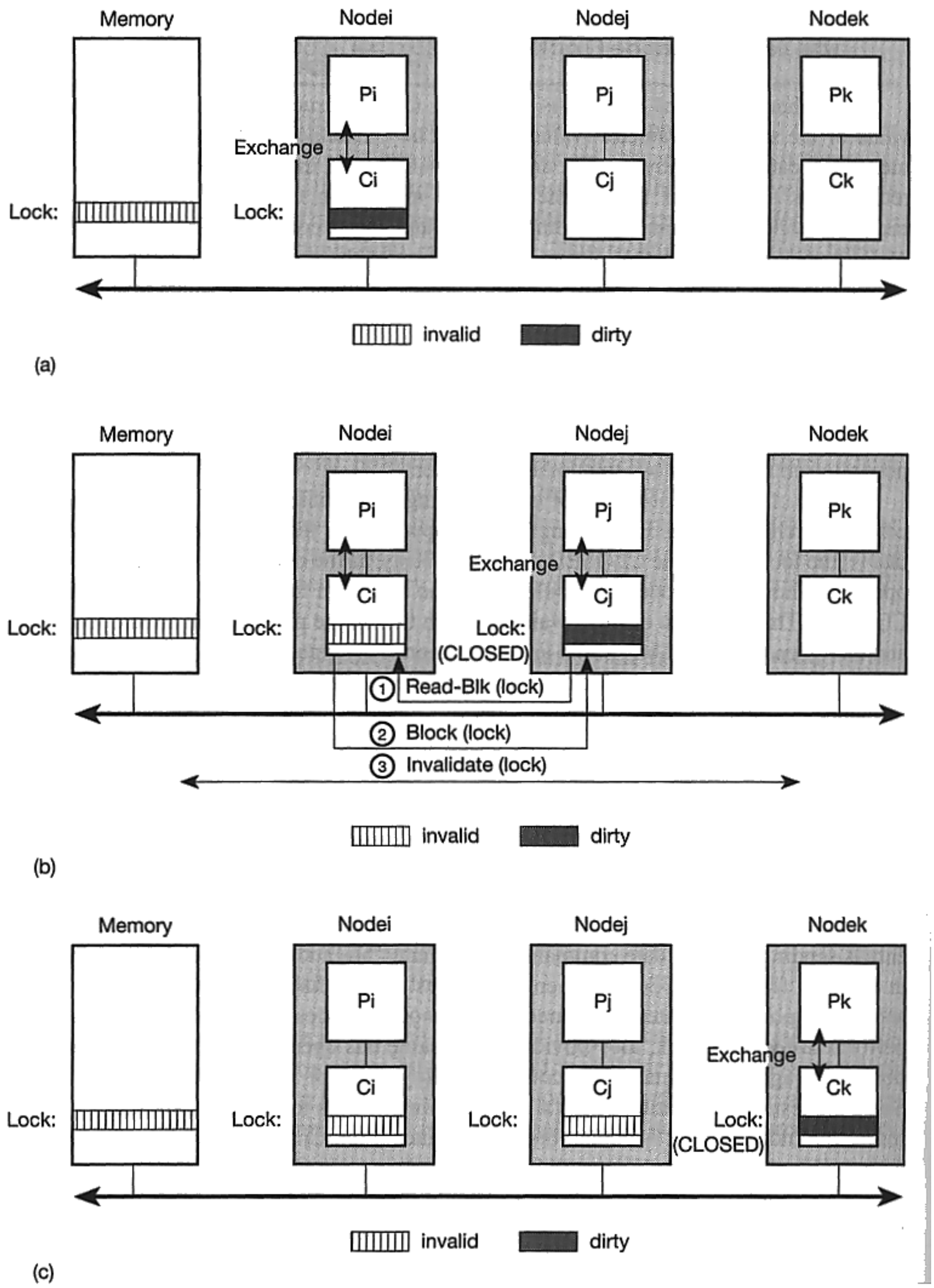


Figure 3-5 Cache implementation of spin-locks. (a) Cache states after Pi successfully executed test&set on lock; (b) bus commands when Pj executes test&set on lock and cache states after; (c) cache states after Pk executed test&set on lock

This variation, called the **test-and-test&set** scheme, has the advantage that a newcomer processor does not disturb the waiting processors. Assume that Pj starts its synchronisation action when Pi is already engaged in its critical section and Pk continuously executes its synchronisation action in order to enter its critical section. Pj (like Pk) receives a shared copy of the lock on which it can test the state of the lock independently from the other processors.

However, the simple solution shown above introduces extra latency for unused locks and cannot prevent bus saturation when the lock is released. There are several approaches to remedy the bus saturation situation at lock release. Anderson proposed **collision avoidance locks** [Anderson89] which enable the waiting processes to restart the exchange operation after a lock release with different delay time. Another approach applies a tree of locks in which the root lock protects the critical section. The tree forms a tournament for competitive processes. The winner of a contest at a lower level becomes a contestant at the next level. When finally a process reaches the root lock and wins the contest it has the permission to enter its critical session. This kind of aggregate of locks is called **tournament locks**. The individual locks of the tournament ones are constructed as snooping locks.

3.3.3.1 Queue lock

The most effective lock type, the **queue lock** was proposed in [GraunThak90]. The structure of a queue lock is more complicated than a single byte representing CLOSED or OPEN state. It actually contains in an array as many such bytes as many processes use the lock for synchronisation. More than that each process can define in the array what it specifies as locking state. The main idea of the queue lock is that processes are waiting for the lock in a queue (represented by the lock array) from which they are served in a first-in-first-out way. Accordingly, there is no competition after an unlock operation, only the "next" process can relock the lock. Moreover, relocking does not need any write operation and hence, the relocking process enters into the critical section without requiring any shared bus operation for accessing the queue lock.

The general premise of a queue lock is to place requests for the CR (Critical Region) into a first-come-first-served queue. This ordering prevents starvation of one process by another. In addition, the queue lock reduces bus activity by limiting the updates to the lock variable. Reads do not cause bus accesses, so they can be plentiful, but writes may cause a flurry of activity that saturates the bus.

To implement this idea we need a hardware_lock() routine that turns off interrupts, thus making a group of statements behave like an atomic action. We designate these sections of code as atomic functions as before.

The lock variable is actually a structure consisting of a head index into a list of waiting tasks. The head index points to the next task in FIFO order. For example, we might use the following structure:

```
struct q_lock {
    int head:                /*  index into array of next
                                 task in FIFO order */
    int tasks [NPROCS]       /*  up to NPROCS-1 tasks
                                 waiting, one active */
} lock;
```

There are three atomic functions for setting, testing, and clearing the lock. Assuming each is atomic, we can create a simple spin lock that only updates the shared lock once at the beginning and once at the end of access. The spin lock only reads the lock, and does not update it.

```
void add_to_q(lock, myid);              /* add task myid to
                                           end of queue lock */
while (head_of_q(lock) != myid);        /* read and wait til
                                           my turn */
CR:                                     /* enter CR and access
                                           shared data */
void remove_from_q(lock, taskid);       /* delete myid.
                                           error if
                                           myid != taskid '/
```

The lock is fair, because access is granted to tasks in the order they request access by executing the add_to_q() routine. If two tasks are running on different processors, and the code is duplicated for each processor, then it is possible for the add_to_q() routine to be entered and executed at exactly the same instant by the two tasks. Conceptually, it is possible for the queue lock to be updated simultaneously by the two competitors, but in practice, the cache coherency algorithm must resolve which update to the lock is allowed to happen first. If we assume the cache acts deterministically, the lock is safe.

3.3.3.2 Multiple readers lock

The simple mutual exclusion locks presented thus far are best suited for the case when most of the accesses are WRITE or UPDATE accesses. These do not perform as well as more elaborate locks when there is a high number of benign accesses to READ the shared memory, because only one task is allowed in the critical section at a time.

In many applications, most accesses are READ and only a few are WRITE. This suggests a lock that discriminates between READERS and WRITERS in order.

We want a lock that gives any number of READERS simultaneous access to the shared data, but as soon as a WRITER appears on the scene, we want to stop giving out accesses to READERS, and after all READERS are flushed out, give mutually exclusive access to the sole WRITER. This seems like an easy enough algorithm, but a fair READERS/WRITER lock is extremely convoluted to design and implement. We note the following:

1. **Safe**: Only one WRITER is allowed to write the shared data at any instance in time.
2. **Fair**: When a WRITER is waiting in line in the waiting queue, all additional READERS must allow the waiting WRITER to go in next, except when a WRITER is currently writing the shared data. When a READER is waiting in the waiting queue, all additional WRITERS must allow the waiting READER to go in next, except when a WRITER is currently writing the shared data.
3. **Live**: No deadlock states. That is, the lock itself must not prevent the READERS and WRITERS from using the shared data once they have entered the waiting queues.

How can we enforce this policy? The following is a DYNIX Pascal code for READERS/WRITERS locking on the Sequent multiprocessor. It uses a counter to enumerate the number of readers, and to monitor when all readers have been flushed out of the critical section.

In addition, you will observe its use of low-level locks to achieve locking of the counters themselves. These low-level locks use ALM (atomic lock memory) on the Sequent to achieve cache efficiency. We can ignore this detail, however, by assuming the low-level routines are safe.

```
TYPE {Types used by the low-level lock routines}
     lockType = RECORD
          lkAlm: ^char; {lock on atomic lock memory}
          lkShadow: -128..127; {shadow lock}
     END;
     lockPtr = ^lockType; {linked list forms a queue}

     RWLockType = RECORD
          mutexcLock, countLock: lockPtr;
          readCount, writeCount: integer;
     END;

VAR
     uLock: RWLockType: (the READERS/WRTERS lock)
          (Low level locks are written in C)
     function create_lock: lockPtr: cexternal;
     procedure lock(p: lockPtr): cexternal;
     procedure unlock(p: lockPtr): cexternal;
```

```
PROCEDURE createRWlock(VAR x: RWlockType);
     BEGIN
          x.mutexcLock := create_lock;
          x.countLock := create_lock;
          x.readCount := 0; {no readers active}
          x.writeCount := 0; {no writers active}
     END;

FUNCTION isReadLocked(x: RWLockType): BOOLEAN;
     BEGIN
          lock(x.countLock); {mutual exclusive update to flag}
          isReadLocked := x.readCount > 0; {are some readers active?}
          unlock(x.countLock); {release mutual exclusion}
     END;


FUNCTION isWriteLocked(x: RWLockType): BOOLEAN;
     BEGIN
          lock(x.countLock); {exclusive update of flag}
          isWriteLocked := x.writeCount > 0; {is one writer active?}
          unlock(x.countLock);
     END;

PROCEDURE Readlock(VAR x: RWlockType);
     BEGIN
     lock(x.mutexcLock);
          WHILE isWriteLocked(x) DO ;
          lock(x.countLock);
          x.readCount := x.readCount + 1;
          unlock(x.countLock);
          unlock(x.mutexcLock);
     END;

PROCEDURE ReadUnlock(VAR x: RWlockType);
     BEGIN
          lock(x.countLock);
          x.readCount := x.readCount - 1;
          unlock(x.countLock);
     END;

PROCEDURE Writelock(VAR x: RWlockType);
     BEGIN
          lock(x.mutexcLock);
          WHILE isReadLocked(x) or isWriteLocked(x) DO ;
          x.writeCount := x.writeCount + 1;
          unlock(x.mutexcLock);
     END;

PROCEDURE WriteUnlock(VAR x: RWlockType):
     BEGIN
          lock(x.countLock);
          x.writeCount := x.writeCount - 1; {sole writer clears out}
          unlock(x.countLock);
     END;
```

The application program interface to these routines is quite simple. Initially, the lock is created:

```
    createRWlock(uLock);
```

All READER tasks execute the following sequence:

```
ReadLock(uLock):
CR;
ReadUnLock(uLock):
```

All WRITER tasks execute the sequence:

```
WriteLock(uLock)
CR;
WriteUnLock(uLock):
```

Figure 3-6 uses a timing diagram to analyse the behaviour of this sophisticated lock. Clearly, the lock is safe because of the mutual exclusion locks used. It is not so obvious that the lock is fair or live. The timing diagram of Figure 3-6 analyses only one possible configuration that could lead to unfair treatment of WRITERS.

In Figure 3-6, the timing diagram follows time from left to right and activation of routines from top to bottom. The first READER, task Readerl, enters routine ReadLock() as indicated by a jump in the timing line. A slight dip in the line indicates that Readerl is in its critical region, CR. Similarly, the ReadUnLock() routine is entered corresponding with a second elevated line.

The values of WriteCount and ReadCount for uLock are displayed at the bottom of the diagram. Each time a value changes, a vertical dashed line is drawn to highlight the change. These counters enumerate the number of tasks of each kind that are currently in the CR or waiting to be admitted.

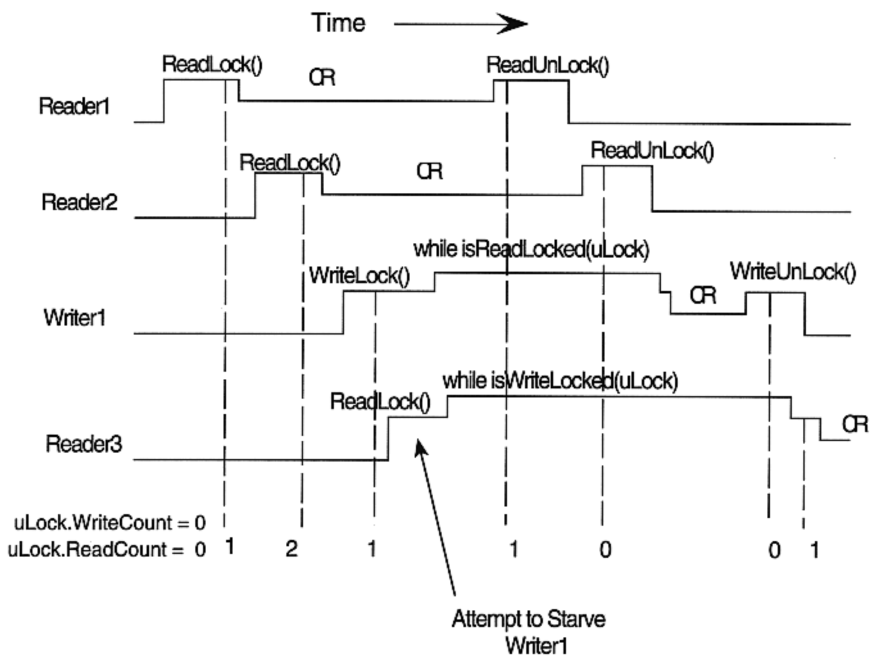


Figure 3-6 Timing diagram for READERS/WRITER sequence Reader1, Reader2, Writer1, and Reader3. Illustrates fairness between readers and sole writer tasks.

Of special interest is the case where Reader3 attempts to enter the CR while the other two readers are in the CR, and Writer1 has requested access. If Reader3 is allowed to enter the CR along with its like kind, Writer1 would be starved. This unfair act is avoided because Writer1 asserts itself by setting the mutex lock WriteCount. When Reader3 attempts to set its lock, it must wait for this counter to revert to zero, thus ensuring fairness.

3.3.4 Systemwide synchronisation

The telephone company has the longest history of dealing with complex parallel systems. People are allowed to pick up their telephone and attempt to call anyone else at any time. If the destination telephone is already in use, the caller hears a busy tone and must hang up and dial again. Callers are not allowed to call themselves, for example. Even with call-waiting service, only one connection between caller and callee is permitted at a time. This greatly simplifies systemwide synchronisation problems. For example, gridlock is avoided.

Unfortunately, parallel systems that share resources such as disk files, printers, and modems are subject to the gridlock problems that we described earlier. It is possible for any parallel program that employs locks on two or more resources to fail due to gridlock. This problem is the classical deadlock problem in operating systems, and in this section we show how it can happen in a parallel program, and we briefly describe a solution.

Suppose we use the Readers/Writers lock of the previous section to control access to a data structure in RAM and a file on disk. Task T1 wants to copy the data structure from RAM to the disk file; task T2 wants to copy the file into the RAM data structure. In both cases, the accesses are to be synchronised by locks, e.g., d_lock for the data structure and f_lock for the file on disk.

Tasks T1 and T2 might be written as follows:

```
T1: ReadLock (d_lock);         {lock RAM data for READER access}
    WriteLock (f_lock);        {lock disk file for WRITER access}
    CR;                        {copy from RAM to disk file ....}
    ReadUnLock (d_lock);       {release the RAM data}
    WriteUnLock (f_lock);      {release the File}
T2: ReadLock (f_lock);         {lock disk file for READER access}
    WriteLock (d_lock);        {lock RAM data for WRITER access}
    CR;                        {copy from disk file to RAM....}
    ReadUnLock (f_lock);       {release the file}
    WriteUnLock (d_lock);      {release the RAM data}
```

This solution looks perfectly innocent on the surface, but if we analyse the interactions between T1 and T2 using the interleave matrix, we learn that a deadlock is possible. However, the interleave analysis requires that we know the internals of ReadLock() and WriteLock(). In general, application programmers may not know how such locks are written.

This problem does not stem from implementation of the locks, because we have just shown that these locks are safe, fair, and live. The problem is a systemwide problem. Regardless of how we implement the lock routines, the possibility of system deadlock persists.

Figure 3-7 shows a tabular analysis of this phenomenon, for one possible sequence of events. The tabular display can be used along with a *horizontal-vertical deadlock detection algorithm* find the problem. In Figures 7(a)-(c) we observe the interleaved execution of T1 and T2. The sequence is:

```
T1: ReadLock(d_lock);
T2: ReadLock(f_lock);
T1: WriteLock(f_lock);
T2: WriteLock(d_lock);
```

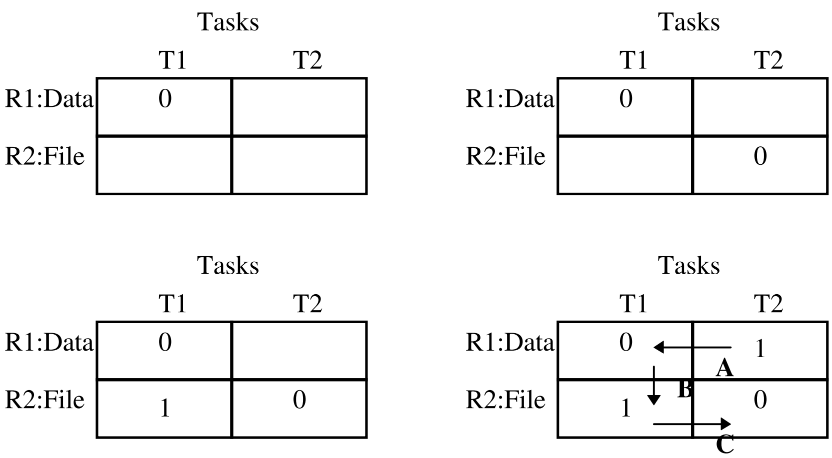


Figure 3-7 Tabular analysis of deadlocked tasks T1 and T2

As the sequence is followed, we place a request number in the cell of the table corresponding to the task and the resource being requested. Thus, when T1 requests d_lock, we place a number in the cell corresponding to T1:Data. We start numbering each row at zero, and increment each time we place a number in a row.

The numbers increase along each row, representing the order of each request for that resource. This is the bakery system where each arriving customer takes a number and waits for his or her turn for service. As each task is served, it is removed from the table, and all numbers along the row are decremented to reflect the new priority. When a task's number reaches zero, it is permitted to lock the resource and make its access.

In this simple example, we want to find out if it is ever possible for the pattern of requests to deadlock the system. We do this with the horizontal-vertical algorithm, which is run each time a task places a number in a row. Figure 3-7(d) illustrates how this algorithm detects a deadlock caused by T1 interacting with T2.

After T2 places a 1 in row Rl:Data, a horizontal search for the 0 is made as shown by letter A in the table. After 0 is found, a vertical search is made to locate the row containing a nonzero entry. This is shown as path B. The process is repeated, searching horizontal for a zero, and vertical for a nonzero, until the search either returns to the original column, or no further row/column can be found. If the search ends up in the original column we detected a deadlock situation.

Because the horizontal-vertical algorithm is performed each time a number is placed in a row, we can always locate a circuit in the table that returns us to the original column, if one exists. This algorithm works regardless of the number of processors and the number of resources. A larger example is shown in Figure 3-8 to illustrate the generality of this method.

The horizontal-vertical algorithm works as follows:

1. Each time a request is made, the algorithm checks for a circular path in the table which leads back to the requesting task.
2. A task is given a zero number when it is allowed access, and a positive integer one greater than the current row maximum representing its order in the waiting line when it is denied immediate access.
3. Every row has at most one zero element (this can be relaxed for READERS with more effort).
4. Every column has at most one positive integer, because a task can only wait on one resource at a time.

From these facts we can deduce that the algorithm will find at most one circuit in the table each time it is executed. When such a circuit is found, the requesting task must be terminated to remove the deadlock. We have shown only the detection algorithm here.

3.4 Monitors

Most parallel algorithms implemented on multiprocessors require a process to perform a series of operations on a shared data structure, as if it were an **atomic operation**. For example, a process may need to fetch the value at the beginning of a linked list and advance the list pointer to the next list element. When the hardware cannot perform the entire series of operations as an atomic operation, the process must have some way to enforce mutual exclusion, keeping all other processes from referencing the resource while it is being modified. The piece of code in which mutual exclusion must be enforced is called a critical section.

Unfortunately, it is easy to miss critical sections, unless you think of the shared resource in a systematic way. One way to structure accesses to shared resources is by using a **monitor**. A monitor consists of variables representing the state of some resource, procedures that implement operations on that resource, and initialisation code. The values of the variables are initialised before any procedure in the monitor is called; these values are retained between procedure invocations and may be accessed only by procedures in the monitor. Monitor procedures resemble ordinary procedures in the programming language with one significant exception. *The execution of the procedures in the same monitor is guaranteed to be mutually exclusive*. Hence monitors are a structured way of implementing mutual exclusion.

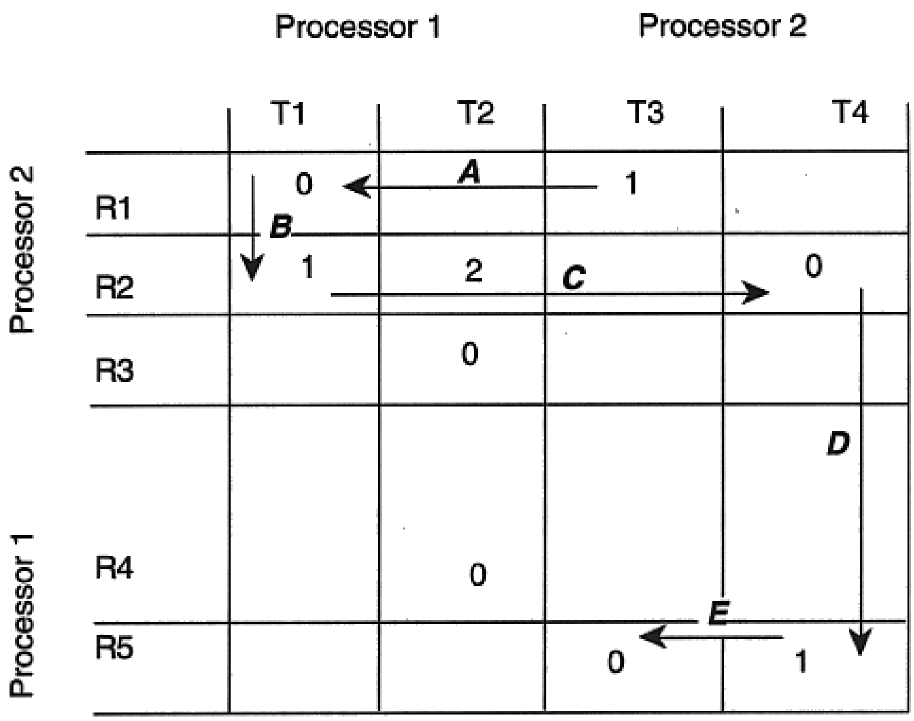


Figure 3-8 Horizontal-vertical algorithm detects deadlock in a system of 4 tasks sharing access to 5 resources

Programming languages that support monitors include Concurrent Pascal [BHansen75], [BHansen77] and Modula [Wirth77a], [Wirth77b], [Wirth77c]. Even if your parallel programming language does not support monitors, you can implement one yourself. For example, in the Sequent C language, you can implement a monitor by declaring a shared lock variable for each resource, putting an s_lock statement that accesses the variable at the start of each procedure, and putting an s_unlock statement at the end of each procedure. You also must have enough self-discipline to use only these procedures to access the shared resource. The following program shows a primitive stack monitor implemented in Sequent C:

```
/* stack.h file included by modules calling stack functions */
void stack_init ();       /* Initialize stack */
int stack_empty ();       /* Returns TRUE if stack is empty */
void push();              /* Pushes integer onto stack */
int pop();                /* Pops integer from stack */

/* stack.c file implementing the monitor */
shared slock_t stack_lock;
shared int top_of_stack, stack[MAX_STACK_SIZE];

void stack_init () {
    s_init_lock (&stack_lock);
    top_of_stack = -1;
}

int stack_empty () {
    int result;
    s_lock(&stack_lock);
    result = (top_of_stack < 0);
    s_unlock(&stack_lock);
    return result;
}
```

```
void push ( i )
    int i;
{
    s_lock ( &stack_lock );
    stack [ ++top_of_stack] = i;
    s_unlock(&stack_lock);
}

int pop () {
    int result;
    s_lock(&stack_lock);
    result = stack[top_of_stack- -];
    s_unlock (&stack_lock);
    return result;
}
```

3.5 Summary

Programming shared memory multiprocessors is similar to multiprogramming or conventional concurrent programming. It requires special attention for the correct solution of synchronisation problems. The main difference between multiprogramming and shared-memory programming comes from the special lock variables that are used in shared memory multiprocessors.