SEMINAR REPORT
Seminar: September in Summer term 2019

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Memory Synchronization Techniques

Levi Ego
Technische Universität Kaiserslautern, Department of Computer Science

Synchronization is a major issue regarding performance and design in modern computer architectures for concurrent programming and the design of concurrent and distributed systems. In response to the impact of synchronization, various techniques have been devised both at the hardware-supported level and user-level constructs. This seminar paper gently introduced the concerns that led to the need for synchronization and majorly explored different memory synchronization techniques adopted so far; the detailed approach and implementation (with algorithms) pertaining to each of them; evaluation of their effectiveness in terms of correctness, fairness and performance; comparisons and trade-offs among them; and few remarks on their efficient usage cases.

1 Introduction

Among the main modifications / adjustments in the architecture of computer is the design of a single computer having multiple processors. Such modification requires a different methodology regarding the way in which applications for a single computer with multiple processors are written. The synchronization demand is basically not only based on multiprocessor systems but also encompasses all processes that execute concurrently that could exist in single processor systems. Obviously, the two major issues that are of great concern when designing systems for such applications are concurrency and synchronization.

In most systems and platforms, synchronization is essential if multiple computers (processors) will simultaneously be active. Moreover, if synchronization is not properly implemented in situations where two or more computers (processes) are updating a common resource, then, data integrity / consistency may not be achieved. Generally, synchronization in computer systems could be referred to as any kind of relation or connection that exists among events. It can be any number of events and any form of relation such as: before, during or after. It is needed when processes's executions are required to keep to order constraints and the requirements for the order of events are the synchronization constraints.

The progress towards designing of faster hardware and developing of concurrent algorithms that are efficient for complex interactions between processes motivated increase in the development of multiprocessor computers. The main part of multiprocessor systems is the shared memory systems that provide processors the privilege to share a global memory. Also, communication is established among tasks processing on different processors by writing to and reading from the global memory. The components of a shared memory computer system comprise memory, disk, processors and interconnection network. The section in a program where different processes could access the shared resources is known as critical section; i.e like a place where the variables or resources being shared are placed in a program.

The techniques to effectively synchronize parallel execution of programs on shared memory multiprocessors are of increasing interest and great importance is directly related with the increase in parallel machines. In order to ensure that the data structure (global memory) being shared is consistent, processors implement hardware-supported atomic primitives for simple processes (operations) and uses synchronization constructs to manage complex processes against conflicting processes overlapping.

Synchronization constructs could be blocking or busy-wait; blocking constructs tries to avoid processes on waiting while the busy-wait constructs constantly perform a check on shared resources so as to confirm when to proceed. The latter is fundamental to parallel executions on shared memory multiprocessors.

Synchronization techniques showed an increasing trend in hardware support due to performance impact, so, some algorithms implemented earlier can atomically (without being interrupted) read and write individual memory locations.

In single processor systems, interrupts could be disabled to avoid preemption of currently executing code but this is quite inefficient for systems with more than one processor. In multiprocessor system, we need some kind of hardware primitives that can read and modify memory locations atomically in order to implement synchronization. Examples of the hardware primitives are: Test and set, compare and swap, Fetch and increment, Load-link / store-conditional. These hardware primitives are basically the building blocks for other synchronization variants such as barriers, locks that were implemented. Generally, the modern processors embed the hardware primitives which have enabled faster synchronization approaches.

The much benefits such as cost-effectiveness and provided by shared memory multiprocessors made them widely used in many aspects of today's computing and applications. So, the modern multiprocessors usually implement shared memory at the basic level program abstraction since it allows to have a same view of data and it is quite simple for programmers to work with. However, the accesses to shared memory may be reordered thereby resulting in an unexpected behaviour of the program. Consequently, memory consistency models were proposed and in order to achieve increased performance, manufacturers implement relaxed kind of memory consistency so as to allow reordering of accesses on parts of memory. Also, there is possibility of unexpected reordering regarding reordering of memory accesses and to avoid such, systems provide fence instruction also called memory barrier to restrict the unaccepted reordering. This ensures consistency of the executing program with the programmer's expectation and correctness of multiprocesses (threads) executing in relaxed memory models.

In [6], showed that synchronization algorithms that are efficient for shared memory multiprocessors of random size can be implemented.

The remaining parts of the paper are structured as follows: In section 2 presents a quite overview of related works and solutions of the addressed problem; section 3 discusses the detailed solutions and section 4 gives a conclusion on the topic discussed so far.

2 Related Work

A lot of variants of synchronization techniques have been implemented. In [6] made detailed contrast regarding recent and previous busy wait synchronization algorithms for multiprocessors having shared memory with a major target of minimizing transactions which results to contention in the network. It showed performance outcomes for different implementations spin locks and barriers and their impacts for designers of hardware and software. The main findings of their work for busy wait synchronization was that memory and interconnect contention should not be a challenge in shared multiprocessors. The findings include performance overview for different busy wait algorithms and this showed that using atomic instruction in appropriate algorithms can effectively minimize synchronization contention to zero. This implies that appropriate spin locks and barriers implementation for varieties of shared memory multiprocessor architecture can remove all contentions in busy-wait. What is needed are the hardware primitives such as fetch instructions and memory hierarchy which grants each processor access to read some part of the shared memory without utilizing the interconnection network. Furthermore, the growing effectiveness of busy wait locks and barriers does not depict the only logical reasoning for synchronization techniques implemented in hardware. Subsequently, some recent works proposed synchronization “fences”.

Fence and atomic instructions are necessary for relaxed memory model machines in order to enforce program correctness during execution. In [5] introduced a concept called scoped fence which places constraints on how programs should be affected by the fences. It is simply a ‘fence’ instruction with scope and customizable in a way that it can only order memory accesses in its scope not knowing the access on the memory beyond its scope. S-Fence gives programmers the privilege of customizing fence instructions by defining the scope for fences. The defined information for the scope is encoded into binaries for use in the hardware. Hardware uses the defined information during run time to ascertain whether to stall a fence because of memory access in the scope that are incompleted. S-Fence gives programmers the privilege to accurately pass across an ordering need of the hardware in order to improve performance.

In [5] performed experimental analysis of different fence instructions and presented new improved fence instruction / algorithm using a quite better fence instruction set such as sfence, lfence in Pentium 4 other than sync in the PowerPc which is a simple fence instruction. Also, it observed that various kind of delays could be enforced by combining different fence instructions and some of these combinations are much better than others.

In [3] discussed concurrent systems with detailed explanations on the mostly major issues and usual results essential for its design. It explored different synchronization mechanism such as semaphores, mutex and barriers implemented by recent operating systems. Also considered other requirements like starvation-freedom, First-In-First-Out in addition to basic requirement of synchronization stated by Dijkstra that could be imposed. It presented efficient algorithms for mutual exclusion problem like Bakery Algorithm – which is the common and suitable mutual exclusion algorithms utilizing only the registers that are safe and satisfies the requirement for FIFO though unbounded; Here, a fixed number of shared memory accesses are required in the absence of contention so as to enter and exit critical path; Adaptive Algorithms – facilitates entrance to the critical path and is guided by the current rate of contention but the time complexity does not depend on the overall number of processors; and Local-spinning algorithms in which process spins registers that are locally accessible.

In [8] explored software techniques for implementing synchronization constraints such as Mutex, Semaphore, Barrier, Reusable Barrier, Queue, FIFO Queue and explored some classical synchronization problems like Producer-consumer problem, Readers-writers problem, No-starve mutex, Dining philosophers; and less classical problems like the roller coaster, River crossing and their simple formulations.

The next section 3 introduces the various existing solutions of the addressed problem.

3 The Solution

Synchronization in a simple term could imply two or more things occurring or happening at the same rate or same time while in computer systems, it could be defined as any form of relationships that exists among any number of operations or events which could be before, during or after. In synchronization, there are basically two concepts: data synchronization and process synchronization which are different but related.

Data synchronization tries to maintain the data by ensuring coherency or consistency of multiple copies of data with each other. It is data exchange which simply sends messages or reads and writes a memory being shared. Process synchronization is processes or multiple threads sharing system resources executing at the same time in such a way to handle concurrent accesses to shared data in order to minimize attempts that can result to data inconsistency. Some examples of process synchronization are Mutex, Semaphore and Lock; and are usually exploited for data synchronization implementation. Synchronization is needed when we have several processes that are executing which need to obey some certain restrictions or order; such is either categorized as cooperation or contention. These two concepts: data exchange and synchronization complements each other because the validity of the data being exchanged is based on the correctness of synchronization of all operations during the exchange.

In shared memory systems, we can have various designs such as cache coherent, distributed and central shared memory systems. Central shared memory systems have the shared resource placed in a central place and the process or processor has no private cache. So, shared memory is remotely being accessed in such systems through the interconnection network linking the memory and process. Cache coherent allow processes to have their own private cache and whenever the shared memory is being accessed, a copy of it is made locally accessible by migrating to the local cache line and the local copy becomes invalid if there is any occurrence of update on the shared memory by some other processes. For distributed shared memory systems, every process keeps a part of the shared memory in its own local memory by taking ownership of that part of the shared memory. So, the part of the shared memory that can be accessed locally by a process is dependent on that part of the shared memory residing physically in the local memory of the process.

It is quite essential to minimize how many times a process usually access a memory location that is being shared which is locally not accessible by that process.

Obviously, accesses to shared resources in a concurrent manner can lead to unintended or wrong operation; therefore, there is need to protect parts of the program that access the shared resource. The protected part is referred to as "critical section" or critical region and more than one process cannot be executing it at a time.

Definition
The problem can be conceptualized formally by assuming each process to be executing an infinite loop of sequence of instructions that is separated into four parts which are the entry, remainder, critical and exit parts. First, a process executes the remainder part at the beginning and will require to execute its critical part at some point. It goes through the entry part so as to access its critical part and the entry part ensures that no other process is permitted to execute its critical part during the time the process (current process) is executing its critical part. Also, whenever a process is done with its execution of critical part, it has to execute its exit part in which it prompts other processes that it has left its critical part. Once it is done with the exit code execution, it goes back to the remainder part.

3.1 Atomicity

Usually, there is always need for two or more tasks to write and / read same data. The majority of the challenges posed by resource allocation or resource sharing in multiprocessor (multitasking) systems is basically because operations are usually not atomically executed; i.e, operations are not executed as a single uninterruptible instruction. This leads to the possibility of a task interfacing with resource being preempted at any time and when rescheduled again, may not take into account if the data (current state) it is currently working on is consistent with the previous data (state) before preemption. Some hardware designs are implemented in some computers which perform uninterrupted instructions known as atomic instructions in that their execution cannot be interfered by other concurrent activities. In order to know the operations that are atomic. Atomic operations could be known from the specific information provided on each hardware platform for each operation in order to write concurrent programs.

In processors, most of the instructions at the machine level execute atomically but not usually the case for high level programming languages which may not guarantee atomicity because they are mostly translated into a sequence of several machine instructions.

There are variants of atomic operations in the operating system, application's support library, language run's time and hardware.

The hardware support that are commonly available are:

3.1.1 Test and set

Most processors offer test and set instruction which basically returns the old value of a memory location (shared memory / register) and sets it to new value.

If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does its steps before the other one starts. The hardware ensures that no other process is permitted to start another test-and-set when a process is currently executing a test-and-set until the current process ends its test-and-set.

```
int TestandSet(int *old_value_ptr, int_new_value)
    {
    int old_value = *old_value_ptr;            // fetch old value
    old_value_ptr = new_value;         // store 'new_value' into "old_value_ptr"
    return old_value;                  // return the old value
    }
```

3.1.2 Fetch and increment

It increments the value or content at an address (memory location) atomically and returns the old value of the address atomically.

```
int FetchAndIncrement(int *old_ptr)
    {
    int old_value = *old_ptr;
    *old_ptr = old_value + 1;
    return old_value;
    }
```

3.1.3 Load-link/store-conditional

The load-link operation reads the current content of a memory location atomically while the store-conditional subsequently writes a value into the same memory location atomically. A process can successful perform a store-conditional operation on a location (address) if the location has not been modified or updated by any other process after the last load-link operation performed on the same location by a process. If the store-conditional is not successful, a failure status is returned.

```
int Load_linked(int *adr_ptr)
    {
            return *adr_ptr;
    }
int Store_conditional(int *adr_ptr, int str_value)
{
    if ( *adr_ptr has not been modified after Load-linked to this address)
        {
        *adr_ptr = str_value;
        return 1;       // successful store-conditional operation
    }else {
        return 0;       // failed store-conditional operation
          }
}
```

3.1.4 Compare and Swap

It takes old value, new value and content of a memory location (register) and compares the current value of the location with the old value, it sets the value of the register to new value if the current value is equal to the old value and returns “true” but the register value remains unchanged if its current value does not equals the old value.

The pseudo-implementation

```
CompareandSwap():
int CompareandSwap(address_value, old_value, new_value)
    {
        lock_get();
        if (*address_value == old_value)
        {
            *address_value = new_value;
            lock_release();
            return true;
        }
        else
        {
        lock_release();
        return false;
        };
    }
```

3.1.5 Swap

It atomically interchanges the value of a local and shared register.

3.1.6 Read-modify-write

A process performs a read of the value at an address, uses the value and computes a new value and get it assigned back to the address atomically.

3.2 Mutual Exclusion

Mutual exclusion in a simple term implies that the occurrence of two events must not take place concurrently. Here, a mutex serves as a token that could be passed among threads in order to permit only one particular thread to proceed at any particular time. Whenever a thread wants access to a shared resource, it needs to acquire the mutex and gives away (releases) the mutex once it is done. At any particular time, only one thread can acquire the mutex thereby ensuring that accesses to a shared resource are carried out by one thread at a time.

Mutex can have variants in implementation that could support some additional features not obtainable in binary or counting semaphores such as Priority inversion, task deletion, ownership and recursive locking. Mutex ownership is acquired by a task through locking of the mutex by the task and the ownership is lost when it is unlocked by the task through releasing it. Recursive lock permits the task that acquires ownership of mutex (i.e, the task that locks it) to keep acquiring it for several times while the mutex is still in the locked state. Such feature can automatically be incorporated into the mutex or explicitly be enable when creating the mutex. Task deletion ensures that within the time a task lock and unlocks the mutex, that the task is not deleted before completion.

Mutex can be regarded as a binary semaphore but differs in the aspect of "signaling" and "waiting" in semaphore by any task while in mutex, the only permitted task to release it is the task that acquires the mutex. So, once a task takes ownership of the mutex by acquiring / locking it, no any other is permitted to either lock or unlock the mutex. Consequently, any attempt to acquire the mutex by another task while the mutex is already locked will result in blocking of that another task while the case is different in a binary semaphore that could be unlocked by any other task (including a task that does not acquire it initially). The waiting / blocked task is unblocked by the operating system immediately the first task (task that currently acquires the mutex) has released the mutex.

Mutual Exclusion solution
Initially, mutex value is set to 1 which implies that a thread can proceed and have access to the shared resource while value of zero implies that any thread that tries to acquire the mutex will be kept on "waiting" until the mutex has been released. So, any thread that first enters a

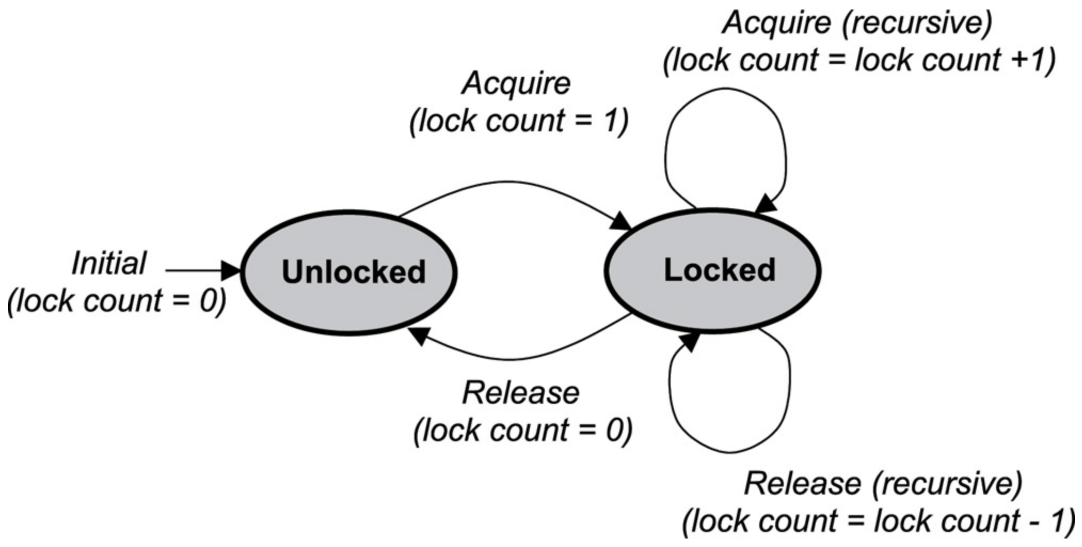

Figure 1: Mutex [1]

“waiting” state will be the first to proceed in acquiring the mutex once it is released and it goes on in like manner.

Thread A

```
mutex.W()       //waiting part of the mutex
# critical region
counter = counter + 1
mutex.S()       //signal part of the mutex
```

Thread B

```
mutex.W()
# critical region
counter = counter + 1
mutex.S()
```

3.3 Semaphore

Semaphores employ signaling techniques in order to permit one or more processes / threads to access a region (critical part). It uses a flag that is initialized to a particular value of which the flag is decremented each time a process attempts accessing the region (critical section) and conversely, it is incremented whenever the process exits the section. This makes it possible for a single semaphore to be taken by different processes several times depending on the initialized value of the flag. Once the value of the flag becomes zero, no other thread can access the section and any attempt by a thread to access or acquire the semaphore results in blocking the thread if it decides to wait for the availability of the section.

Basically, a semaphore could be referred to as an integer having three distinct behaviours:

1. Once the semaphore is created, it is initialized to any integer value and subsequently can permit either increment or decrement operations on it but its current value cannot be read.

2. A thread blocks itself if after decrementing the value of the semaphore and the result is negative; so, it cannot proceed until the value has been incremented by another thread.

3. Whenever the value of the semaphore is incremented by a thread, then, one of the threads in the "waiting" state gets unblocked if there is any waiting thread. A positive value of the semaphore refers to the allowable number of threads without blocking, a negative value denotes blocked and "waiting" number of threads and the value of zero denote that there is no blocked (waiting) thread and any attempt to further decrement by a thread will result in that thread being blocked.

Semaphore can be adopted as mutual exclusion by setting its values between 0 and 1. It is usually manipulated using "wait" and "signal" operations. Three variants of semaphore exit: Binary semaphore, counting semaphore and mutual exclusion (mutex). A binary semaphore is initially set to values of "0" or "1" when created, therefor it takes only "0" or "1" as its values and it is regarded as being available or unavailable when the value is 0 or 1 respectively. It is placed as a global resource making it to be shared among all processes that will be using it. Being global resource enables it to be released by any process (task) whether it is initially acquired by the process or not. Counting semaphore has a "count" variable that grants processes the privilege to acquire and release it several times and also a global resource thereby allowing it to be shared among tasks and released by any task.

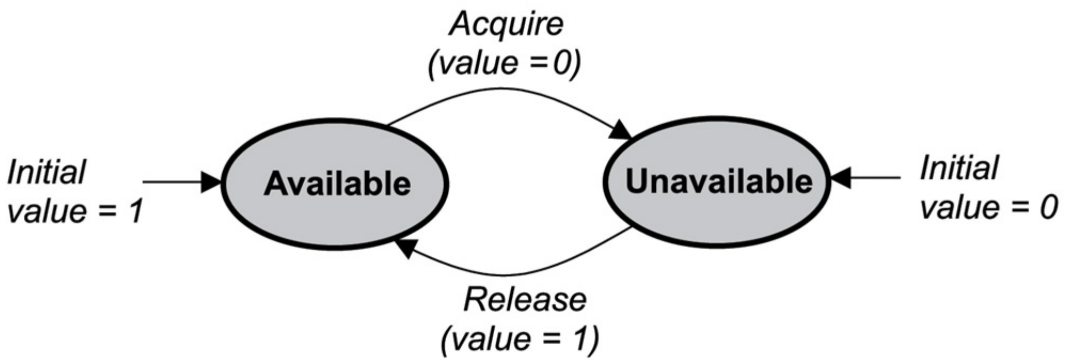

Figure 2: The state diagram of a binary semaphore [1]

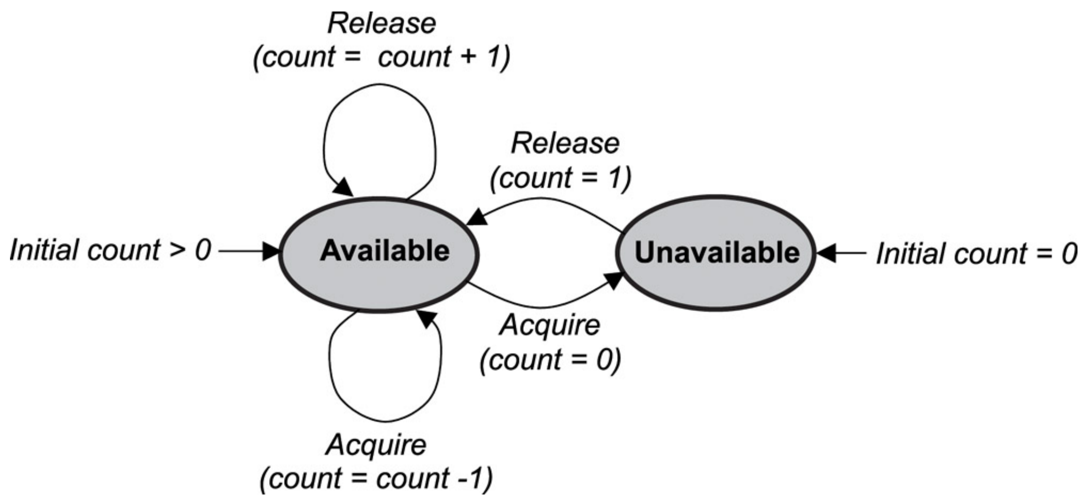

Figure 3: The state diagram of a counting semaphore [1]

The semaphore has to be initialized and deleted: (sem_initial (Semaphore,Init_count)); (sem_delete(Semaphore)). The basic structure of the semaphore could have at least two fields for implementation of its data structure.

```
struct S {
int counter; // semaphore count.
queue Queued_tasks; // tasks blocked that are waiting.
}
```

```
S Sem;
semaphore_wait(Sem)
{
if (Sem.counter > 0) then Sem.counter = Sem.count - 1;
else the task in Sem.Queued_tasks is blocked;
}

semahpore_signal(Sem)
{
if (Sem.Queued_tasks is non-empty) then allow a task in Sem.Queued_tasks;
else Sem.counter = Sem.counter + 1;
}
```

If the value of the “counter” is zero or negative, then, a task executing “semaphore_wait()” is blocked and whenever a “semaphore_signal()” is executed, “counter” is decremented. When the counter value becomes non-negative or zero, one of the tasks in the queue which has been blocked is unblocked. So, a negative value of sem.counter implies that a minimum of one task has been blocked while its absolute value tells how many tasks that were blocked on the semaphore.

Also, a semaphore can be applied to other synchronization targets such as two tasks synchronizing with each other whether one of them wants to exclude the other or not from accessing a shared resource; this can also be implemented using "semaphore_wait()" and "semaphore_signal()".

Task A:

```
Semaphore Sem;
{...
main()
 do_first();                //executes first section of the job
 semaphore_signal(Sem);
 do_another_thing();        //does something different
}
```

Task B:

```
main()
{...
    do_another_thing();         //does something different
    semaphore_wait(Sem);
    do_second();                //executes second section of the job
    ...
}
```

Finally, the locking task usually go to sleep resulting in the much cost for context switching. So, busy-wait and critical regions that takes quite a little time are not supported by semaphore; preferably for theses cases are spin locks.

3.4 Spin Lock

It is the easiest to design and another efficient approach for synchronization constructs / implementation. It simply performs the action of "spinning" in order to determine when the lock is available. Before each processor accesses a shared resource, it has to check the flag and if the flag is at reset state, then, the process can set the flag and proceeds in the execution of the thread while in the case where the flag is already set, the thread continue to repeat checking on the flag by spinning in order to determine when the flag is not set. This approach adopted by spin lock is referred to as "busy-wait" or "spinning". So, basically what spin lock does is to keep repeating a check on the lock in order to acquire it without entering into "waiting" state and this offers better performance on the assumption that the locks are utilized for a low period of time. The efficient performance of spin lock is dependent on the cycle reset time; it is more effective if reset cycles are quite low else it results to reduced performance as much processor cycles are wasted through the spinning process.

Spin locks are basically the building blocks used in synchronization implementation like monitors, semaphores and they proffer an approach for realizing mutual exclusion (restricts the

Once a "test-and-set" is executed at first, a zero is read and the lock is set to 1. Then, subsequent execution of "test-and-set" will notice that the value of lock is 1 and will continuously loop until the value of lock is zero which implies that the lock is available.

```
typedef struct spinlock
{
    int lock_flag;
}lock_s;

void init_lock(lock_s *lock)
{
    //if lock is available, flag becomes 0 but if held, the value is 1
    lock->lock_flag = 0;
}

void s_lock (lock_s *lock)
{
   While(TestandSet(&lock->lock_flag, 1) ==1)
   ;   // keeps spinning, i.e it does nothing

void s_Lock (lock_s *lock)
{
   Lock->lock_flag = 0;
}
```

accessing of shared resource by only one process). They are generally used for parallel systems implementation and mostly used in protecting small critical regions and can be executed several number of times during execution. Spin locks are quite suitable for locks used in multiprocessor systems and other contexts such as interrupt service routine and kernel call. However, some kind of operation while inside the critical region cannot be performed if spin lock locks the region such as an operation that requires much time or going on to "sleep" mode. It is much preferable and efficient to use mutexes and semaphores regarding such. In summary, a task attempts to acquire a lock shared by all processes using spin lock and if it fails, it repeats attempting until it succeeds in acquiring the lock.

Adaptive Spinning: the thread keeps spinning for some specified amount of time in order to acquire the lock and once the specified time is elapsed and the lock is still not unavailable, the thread goes into wait state.

A preemptive scheduler (that is scheduler that allows another thread to execute by interrupting the currently executing thread) is needed for spin locks to correctly execute on a single processor system. Spin locks do not perform effectively on a single processor if there is no preemption mechanism incorporated because a spinning task / thread on processor will hardly give it up. So, the overheads from the performance on a single processor can be much ineffective but performs quite well in multiple processors (estimated number of threads and processors should be equal). Spin locks ensure correctness by permitting a single thread to access the critical region at a time thereby providing a correct lock. Spin locks serves as a trade-off in disabling preemption of processes (not efficient for multiprocessors) and the time wasted in busy-wait. However, spin locks do not provide fairness since it does not always ensure that a thread on "waiting" will eventually access the critical section. Therefore, a thread can keep spinning forever due to contention thereby resulting to starvation of the thread. Spin locks have a latency issue which is not deterministic and this affects its use in real-time scenarios but provides nice solution in the aspect of the times utilized in context switching and scheduling using locks which are larger that the needed time for executing the critical region being guarded by the spin lock.

We can also construct a lock using "compare and swap" instruction in the same way "test and set" was used with the lock routine being substituted:

```
void spinlock (lock_s *lock)
{
   While(CompareandSwap(&lock->lock_flag, 0, 1) ==1)
   ;   // keep spinning, i.e, it does nothing
}
```

Using Load-linked and Store conditional

```
void spinlock (lock_s *lock)
{   while(1)
    {
       while(Load_linked(&lock->lock_flag) == 1)
           ; // spin until it is zero
       if (Store_conditional(&lock->lock_flag, 1) == 1)
           return;

   }
}
void s_unlock(lock_s *lock)
{
lock->lock_flag = 0;
}
```

3.5 Barriers

It is usually quite suitable to design algorithms that are sectioned in a way that for any process to move from each section to another or subsequent sections, all other processes must have executed and ended the current section and are set to proceed to the next section. Such kind of restraint can be actualized / ensured through the use of a "barrier" being placed at each section's end of the algorithm.

So, barriers offer a mechanism in order to ensure before any process will proceed beyond a current phase of the algorithm (computation), all other processes must have arrived at that phase.

It subjects processes involving in concurrent computations to wait until all of the processes have gotten to a specific point in the program. They are employed in marking out sections in application programs. Barriers are commonly used in parallel algorithms with short data phases; processes are allowed to proceed beyond the barrier once all the processes have gotten the barrier.

Barrier implementation can initialize a counter with a number equal to that of the threads and the counter is decremented whenever the barrier is reached by one of the threads and the thread is blocked. The cost of the barrier is in direct relation with the number if the threads since synchronization is required for each decrements made.

Barrier Solution

```
mutex.W()
counter = counter + 1
mutex.S()

if counter == number_threads;
    Barrier.S() //processes are allowed to proceed

Barrier.W();
Barrier.S();

//critical region
```

The synchronization demand is that before the critical section will be executed, all other threads must have executed the rendezvous. Example, if we have p threads of which this value is saved in a variable p which could be accessed by all threads, the first thread to arrive is blocked and subsequent threads will be blocked until all the threads have arrived, then, the threads can be permitted to proceed again.

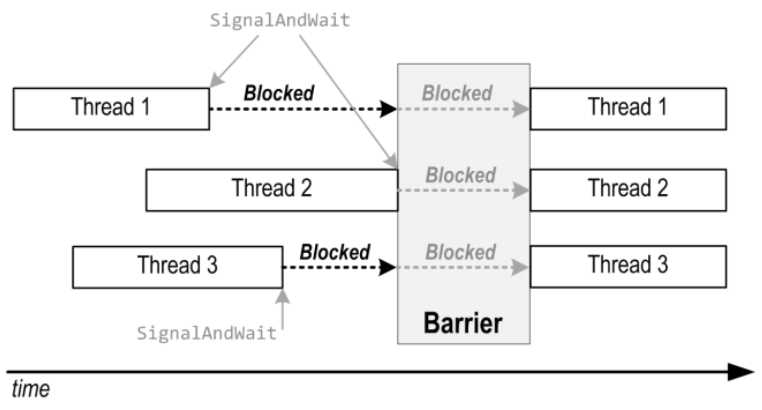

Figure 4: **Barrier** [2]

Counter tells the number of threads that have reached the barrier and is guarded by the mutex. Mutex ensures that counter can safely be incremented by the threads.

It is usual in barrier synchronization to have some threads waiting for other threads to reach the barrier before threads can proceed thereby resulting in much negative impact in the performance of the process.

3.6 Read / Write Locks

Most times data is usually guarded against multiple writings at the same time but not for multiple reading happening at the same time. Therefore, several tasks can be allowed at the same time to acquire a "read" lock I order to access a particular critical section but only allows one single task to acquire the "write" lock depending on if all the locks acquired for read have been released. Tasks attempting to perform "read" are blocked whenever a task is performing a "write" on the data until the writing task is through with writing.

The "reads" are necessary for accessing data structures such as linked lists which are complex, usually few writings and majority of the tasks mostly perform read in order to search for the interested element in the list. High concurrency occurs in multiple reader / single writer ownership during reading. The multiple read / single write lock is preferred for the longer parts of programs that are mostly accessed (read) with few times of making changes.

Read / Write code in Linux

```
read_write_lock  rw_lock = RW_LOCK_UNLOCKED;  // initialize

r_lock(&rw_lock);
// critical section allows "read only"
readunlock(&rw_lock);

w_lock(&rw_lock);
//critical section allows "read and write"
w_unlock(&rw_lock);
```

3.7 Monitors

Monitors are used for communication, synchronization and to guarantee that resources are accessed exclusively. It encompasses definition for resources and the exclusive manipulation of operations on it. The operations serve as shared resource's gateway which the processes call in order for the resource to be accessed. An operation of the monitor can be activated at a time by only a single call in order to ensure that data is protected inside the monitor from many users at a time thereby imposing mutual exclusion. Processes that try to access the monitor are blocked when the monitor is occupied and the blocked processes are stored in an entry queue for the monitor.

A monitor uses a condition variable with “signal” and “wait” operations defined in it for synchronizing tasks. When a particular condition is true, calling process execution is suspended and monitor is unlocked thereby allowing another task to make use of it. Signal operation is performed when that particular condition is false and then suspended processes are resumed for execution in the ready queue for processing. The queue currently waiting to be processed based on a condition is linked with that condition variable.

General Structure of a Monitor

```
< name > : monitor
 begin
    //local data of the monitor should be declared
    .
    .
    function (parameters);
        begin
            //body of the
        end;

    //declare rest of the functions
    .
    .
    begin
        //Initialization of local data of the monitor
    end;
end.
```

The major challenge in the use of monitor is that it does not provide concurrency rather it permits only one process being active at a particular time in the monitor. A “wait” and “signal” can suspend a process from executing and unblocks another process respectively which are similar to the operation of P and V in semaphores. The difference is seen in the execution of operation P which actually does not need to block a process since the semaphore can have a value more than zero contrary to the “wait” operation in monitor of which a process is always blocked. Likewise same scenario when operation “V” is performed in semaphore which can either increment the value of the counter of the semaphore in case of no task to be unlocked or unblocks task that is waiting in the queue while if “signal” operation in a monitor is performed and there is no process to be unblocked, then, no effect takes place on the condition variable.

3.8 Fence Instructions

Contemporary multiprocessors systems are designed to realize a better performance through the implementation of a relaxed memory model (consistency). Such systems with relaxed memory are designed with a mechanism called “Fence instructions” which enables it to override the default access order of their relaxed memory in a selective manner.

Fence instructions place a kind of order on the operations in the memory which ensures that all previously executing operations in the memory by the processor must have finished executing whenever the fence instruction is started for execution. Also, it ensures that none of the operations in the memory following it in the program is permitted to execute until the execution of the fence instruction is completed.

There are different names and descriptions regarding fence instructions used in various architectures like Alpha architecture uses write memory barrier and memory barrier (MB), MIPS and PowerPC uses sync; Intel IA-64 uses memory fence, Intel Pentium 4 uses load fence (lfence), store fence (sfence) and memory fence; SPARC V8 uses store barrier; SPARC V9 uses write-write (MEMBAR), write-read, read-write and read-read fences.

The fence instructions that are part of the concurrent programs are meant to ensure that accesses to shared data done concurrently are correct without taken note of how the data being accessed is processed but there are traditional fences which equally place an order with respect to access to memory which belongs to the section that executes the data. So, in the case where there is long latency in accessing the memory during data processing, then, the fence instructions that are part of the algorithm are suspended in order for the traditional fences to finish thereby resulting to unrequired stalls at the fences. In order to avoid such, a technique should be devised that should separate accesses to memory that need to be ordered from other accesses to memory. In [5] introduced a concept called "scoped fence" that restricts the impacts fences have on programs. It gives some levels of privileges to programmers by using fence instruction that can be customized so as to enable them define the scope for the fence instruction. The information defined in the scope is transformed in binary form for the hardware use. The hardware uses the information defined in the scope during execution in order to ascertain if a fence should stall because of a memory access that is yet to be completed in the scope. This helps in minimizing the overhead arising from the fence since the order on the memory access has been restricted to some certain scopes. Also, scope fence is quite scalable since changes are made locally to processor and does not need communication among processors.

Fence instructions are quite beneficial considering program's correctness if they are executed on architectures with relaxed memory but are quite slower when compared to other existing instructions.

The next section 4 gives conclusion based on the solutions discussed.

4 Conclusions

Memory synchronization is a major consideration in the design of a single computer with multiple processors and single processor executing concurrent processes. Correct synchronization will ensure data integrity / consistency in multiprocessor systems which usually adopt shared memory. Techniques to efficiently implement synchronization draws much attention and are quite important with the increase in parallel machines. The various synchronization techniques such as spin locks, barriers are based on the hardware primitives (e.g: fetch and increment, test and set, compare and swap,Load-link / store conditional) which are the basic building blocks for their implementation.

Modern multiprocessors provide mechanism called fence instruction for selectively overriding their default relaxed memory access order and also to prevent reordering of memory accesses that are not permitted. Fence instructions provides correctness of multithreaded programs running under relaxed memory models and guarantee consistency with programmer's expectation.

References

[1] *The state diagram of a binary semaphore.* `http://web.archive.org/web/20190222011118/http://www.embeddedlinux.org.cn/rtconforembsys/5107final/LiB0037.html`. Accessed: 2019-09-13.

[2] *Thread Execution Barrier.* `http://web.archive.org/web/20190905010737/http://www.albahari.com/threading/part4.aspx`. Accessed: 2019-09-13.

[3] Allen Downey (2008): *The little book of semaphores.* Green Tea Press.

[4] Xing Fang, Jaejin Lee & Samuel P Midkiff (2003): *Automatic fence insertion for shared memory multiprocessing.* In: *Proceedings of the 17th annual international conference on Supercomputing*, ACM, pp. 285–294.

[5] Changhui Lin, Vijay Nagarajan & Rajiv Gupta (2014): *Fence Scoping.* In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE Press, Piscataway, NJ, USA, pp. 105–116, doi:10.1109/SC.2014.14. Available at `https://doi.org/10.1109/SC.2014.14`.

[6] John M Mellor-Crummey & Michael L Scott (1991): *Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS)* 9(1), pp. 21–65.

[7] Simple But Not Scalable: *Lock-based Concurrent Data Structures.*

[8] Gadi Taubenfeld (2008): *Shared Memory Synchronization. Bulletin of the EATCS* 96, pp. 81–103.