Synchronization Primitives

- Synchronization is used to coordinate the activity of multiple threads. There are various situations where it is necessary; this might be to ensure that shared resources are not accessed by multiple threads simultaneously or that all work on those resources is complete before new work starts.

- Most operating systems provide a rich set of synchronization primitives. It is usually most appropriate to use these rather than attempting to write custom methods of synchronization.
- There are two reasons for this. Synchronization primitives provided by the
  operating system will usually be recognized by the tools provided with that operatingsystem. Hence, the tools will be able to do a better job of detecting data races or correctlylabeling synchronization costs.
- The operating system will often provide support for sharing the primitives between threads or processes, which can be hard to do efficiently without operating system support.

1. Mutexes and Critical Regions

- The simplest form of synchronization is a ***mutually exclusive*** (***mutex***) lock. Only one thread at a time can acquire a mutex lock, so they can be placed around a data structure to ensure that the data structure is modified by only one thread at a time.

```
int counter;
mutex_lockmutex;
void Increment()
{
        acquire(&mutex );
        counter++;
        release(&mutex );
}
void Decrement()
{
        acquire(&mutex );
        counter--;
        release(&mutex );
}
```

Fig 7 : how a mutex lock could be used to protect access to a variable

- In the example, the two routines Increment() and Decrement() will either increment or decrement the variable counter.
- To modify the variable, a thread has to first acquire the mutex lock.
- Only one thread at a time can do this; all the other threads that want to acquire the lock need to wait until the thread holding the lock releases it.
- Both routines use the same mutex; consequently, only one thread at a time can either increment or decrement the variable counter

- If multiple threads are attempting to acquire the same mutex at the same time, then only one thread will succeed, and the other threads will have to wait. This situation is known as a contended mutex.
- The region of code between the acquisition and release of a mutex lock is called a ***critical section, or critical region***. Code in this region will be executed by only one thread at a time.
- As an example of a critical section, imagine that an operating system does not have an implementation of ***malloc()*** that is thread-safe, or safe for multiple threads to call at the same time. One way to fix this is to place the call to malloc() in a critical section by surrounding it with a mutex lock.

```
        void * threadSafeMalloc( size _t size )
{
        acquire(&mallocMutex );
        void * memory = malloc( size );
        release(&mallocMutex );
        return memory;
}
```

Fig 8 Placing a Mutex Lock Around a Region of Code

- If all the calls to malloc() are replaced with the threadSafeMalloc() call, then
  only one thread at a time can be in the original malloc() code, and the calls tomalloc() become thread-safe.
- Threads block if they attempt to acquire a mutex lock that is already held by another thread.
- Blocking means that the threads are sent to sleep either immediately or after a few unsuccessful attempts to acquire the mutex.

2. Spin Locks

- Spin locks are essentially mutex locks. The difference between a mutex lock and a spin lock is that a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping.
- In comparison, a mutex lock may sleep if it is unable to acquire the lock.
- The ***advantage*** of using spin locks is that they will acquire the lock as soon as it is released, whereas a mutex lock will need to be woken by the operating system before it can get the lock.
- The ***disadvantage*** is that a spin lock will spin on a virtual CPU monopolizingthat resource. In comparison, a mutex lock will sleep and free the virtual CPUfor another thread to use.
- Often mutex locks are implemented to be a hybrid of spin locks and more traditionalmutex locks. The thread attempting to acquire the mutex spins for a short while beforeblocking. There is a performance advantage to this.
- Since most mutex locks are held foronly a short period of time, it is quite likely that the lock will quickly become free forthe waiting thread to acquire.
- So, spinning for a short period of time makes it morelikely that the waiting thread will acquire the mutex lock as soon as it is released.
- However, continuing to spin for a long period of time consumes hardware resources thatcould be better used in allowing other software threads to run.

3. Semaphores

- Semaphores are counters that can be either incremented or decremented. They can be used in situations where there is a finite limit to a resource and a mechanism is needed to impose that limit.
- An example might be a buffer that has a fixed size. Every time an element is added to a buffer, the number of available positions is decreased.
- Every time an element is removed, the number available is increased.
- Semaphores can also be used to mimic mutexes; if there is only one element in the semaphore, then it can be either acquired or available, exactly as a mutex can be either locked or unlocked.
- Semaphores will also signal or wake up threads that are waiting on them to use available resources; hence, they can be used for signaling between threads.
- For example, a thread might set a semaphore once it has completed some initialization. Other threads could wait on the semaphore and be signaled to start work once the initialization is complete.

- Depending on the implementation, the method that acquires a semaphore might be called wait, down, or acquire, and the method to release a semaphore might be called post, up, signal, or release.
- When the semaphore no longer has resources available, the threads requesting resources will block until resources are available.

4. **Readers-Writer Locks**

- Data races are a concern only when shared data is modified. Multiple threads reading the shared data do not present a problem. Read-only data does not, therefore, need protection with some kind of lock.
- However, sometimes data that is typically read-only needs to be updated. A readerswriter lock (or multiple-reader lock) allows many threads to read the shared data but can then lock the readers threads out to allow one thread to acquire a writer lock to modify the data.
- A writer cannot acquire the write lock until all the readers have released their reader locks. For this reason, the locks tend to be biased toward writers; as soon as one is queued, the lock stops allowing further readers to enter.
- This action causes the number of readers holding the lock to diminish and will eventually allow the writer to get exclusive access to the lock.

- The code snippet in Fig 9 shows how a readers-writer lock might be used.

- Most threads will be calling the routine readData() to return the value from a particular pair of cells. Once a thread has a reader lock, they can read the value of the pair of cells, before releasing the reader lock.
- To modify the data, a thread needs to acquire a writer lock. This will stop any reader threads from acquiring a reader lock. Eventually all the reader threads will have released their lock, and only at that point does the writer thread actually acquire the lock and is allowed to update the data.

Using a Readers-Writer Lock

```
intreadData( int cell1, int cell2 )
{
        acquireReaderLock(&lock );
        int result = data[cell] + data[cell2];
        releaseReaderLock(&lock );
        return result;
}
voidwriteData( int cell1, int cell2, int value )
{
        acquireWriterLock(&lock );
        data[cell1] += value;
```

```
            data[cell2] -= value;
            releaseWriterLock(&lock );
        }
```

Fig 9 Using a Readers-Writer Lock

5. Barriers

- There are situations where a number of threads have to all complete their work before any of the threads can start on the next task. In these situations, it is useful to have a barrier where the threads will wait until all are present.
- One common example of using a barrier arises when there is a dependence between different sections of code. For example, suppose a number of threads compute the values stored in a matrix. The variable total needs to be calculated using the values stored in the matrix.
- A barrier can be used to ensure that all the threads complete their computation of the matrix before the variable total is calculated.
- Fig 10 shows a situation using a barrier to separate the calculation of a variable from its use

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
```

Fig.10 Using a Barrier to Order Computation

- The variable total can be computed only when all threads have reached the barrier. This avoids the situation where one of the threads is still completing its computations while the other threads start using the results of the calculations. Notice that another barrier could well be needed after the computation of the value for total if that value is then used in further calculations.

```
Compute_values_held_in_matrix();
    Barrier();
    total = Calculate_value_from_matrix();
    Barrier();
    Perform_next_calculation( total );
```

Fig.11 Use of Multiple Barriers

6. Atomic Operations and Lock-Free Code

- Using synchronization primitives can add a high overhead cost. This is particularly true ifthey are implemented as calls into the operating system rather than calls into a supportinglibrary.
- These overheads lower the performance of the parallel application and can limit scalability. In some cases, either ***Atomic*** operations or ***lock-free*** code can produce functionallyequivalent code without introducing the same amount of overhead.
- An ***Atomic*** operation is one that will either successfully complete or fail; it is not possiblefor the operation to either result in a "bad" value or allow other threads on the systemto observe a transient value.
- An example of this would be an atomic increment,which would mean that the calling thread would replace a variable that currently holdsthe value N with the value N+1.

```
LOAD [%o0], %o1 // Load initial value
ADD %o1, 1, %o1 // Increment value
STORE %o1, [%o0] // Store new value back to memory
```

Fig.12 Steps Involved in Incrementing a Variable

- During the execution of the three steps shown in Fig 4.12, another thread could have interfered and replaced the value of the variable held in memory with a new value, creating a data race.
- An ***Atomic*** increment operation would not allow another thread to modify the samevariable and cause an erroneous value to be written to memory.
- Performing the incrementoperation atomically is logically equivalent to implicitly acquiring a mutexbeforethe increment and releasing it afterward.
- The difference is that using a mutex relies onother threads using the same mutex to protect that variable.
- A thread that did not use thesame mutex could cause an incorrect value to be written back to memory.
- With ***Atomic***operations, another thread cannot cause an incorrect value to be written to memory.Most operating systems or compilers provide support for a range of atomic operations.
- A ***lock-free*** implementation would not rely on a mutex lock to protect
- access; instead, it would use a sequence of operations that would perform the operation without having to acquire an explicit lock.
- Lock-free does not imply that the other threads would not have to wait; it only indicates that they do not have to wait on a lock.

7. Deadlocks and Livelocks

- ***Deadlock***, where two or more threads cannot make progress because the resources that they need are held by the other threads.
- **Example**: Suppose two threads need to acquire mutex locks A and B to complete some task. If thread 1 has already acquired lock A and thread 2 has already acquired lock B, then A cannot make forward progress because it is waiting for lock B, and thread 2 cannot make progress because it is waiting for lock A. The two threads are deadlocked.
- Fig13 shows this situation.

Thread 1	Thread 2

void update1()
{
acquire(A);
acquire(B); <<< Thread 1 waits here
variable1++;
release(B);
release(A);
}
	void update2()
	{
	acquire(B);
	acquire(A); <<< Thread 2 waits here
	variable1++;
	release(B);
	release(A);
	}

Fig13Two Threads in a Deadlock

- The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order. So if thread 2 acquired the locks in the order A and then B, it would stall while waiting for lock A without having first acquired lock B. This would enable thread 1 to acquire B and then eventually release both locks, allowing thread 2 to make progress.
- A ***livelock*** traps threads in an unending loop releasing and acquiring locks. Livelockscan be caused by code to back out of deadlocks.
- In Fig.14 shows a mechanism that avoids deadlocks. If the thread cannot obtain the second lock it requires, it releases the lock that it already holds.
- The two routines ***update1()*** and ***update2()*** each have an outer loop. Routine
  update1() acquires lock A and then attempts to acquire lock B, whereas ***update2()***does this in the opposite order. This is a classic deadlock opportunity, and to avoid it, thedeveloper has written some code that causes the held lock to be released if it is not possibleto acquire the second lock. The routine ***canAquire(),*** returnsimmediately either having acquired the lock or having failed to acquire the lock.

Thread 1	Thread 2

void update1()
{
int done=0;
while (!done)
{
acquire(A);
if (canAcquire(B))
{
variable1++;
release(B);
release(A);
done=1;
}
else
{
release(A);
}
}
}
	void update2()
	{
	int done=0;
	while (!done)
	{
	acquire(B);
	if (canAcquire(A))
	{
	variable2++;
	release(A);
	release(B);
	done=1;
	}
	else
	{
	release(B);
	}
	}
	}

Fig14 Two Threads ina Livelock

- Each thread acquires a lock and then attempts to acquire the second lock that it needs. If it fails to acquire the second lock, it releases the lock it is holding, beforeattempting to acquire both locks again.
- The thread exits the loop when it manages to successfully acquire both locks, which will eventually happen, but until then, the application will make no forward progress.