Unit-II

D.Venkata Vara Prasad

Data Races

- A data race occurs when multiple threads use the same data item and one or more of those threads are updating it
- Data races are the most common programming error found in parallel code.
- Ex: where a pointer to an integer variable is passed in and the function increments the value of this variable.

Data Races

- Ex: Updating the Value at an Address

```
void update(int * a)
    {
    *a = *a + 4;
    }
```

- The SPARC disassembly for this code

```
ld [%o0], %o1          // Load *a
add %o1, 4, %o1        // Add 4
st %o1, [%o0]          // Store *a
```

Data Races

- Suppose this code occurs in a multithreaded application and two threads try to increment the same variable at the same time
- Value of variable a = 10

Data Races

Thread 1	Thread 2
ld [%o0], %o1 // Load %o1 = 10	ld [%o0], %o1 // Load %o1 = 10
add %01, 4, %o1 // Add %o1 = 14	add %01, 4, %o1 // Add %o1 = 14
st %o1, [%o0] // Store %o1	st %o1, [%o0] // Store %o1
Value of variable a = 14	

Data Races

- Each thread adds 4 to the variable, since they do it at exactly the same time, the value 14 is stored into the variable.
- If the two threads had executed the code at different times, then the variable would have ended up with the value of 18.

Data Races

- Another situation where one thread holds the value of a variable in a register.
- Second thread comes in and modifies this variable in memory while the first thread is running through its code.
- The value held in the register is now out of sync with the value held in memory.

Avoiding Data Races

- It is hard to identify data races
- Avoiding them can be very simple
- Make sure that only one thread can update the variable at a time.
- The easiest way to do this is to place a ***synchronization lock***
- *ensure that before* referencing the variable, the thread must acquire the lock.

Avoiding Data Races

- This version uses a *mutex lock*

```
void * func( void * params )
{
    pthread_mutex_lock( &mutex );
    counter++;
    pthread_mutex_unlock( &mutex );
}
```

Synchronization Primitives

- Synchronization is used to coordinate the activity of multiple threads.
- It is necessary to ensure that shared resources are not accessed by multiple threads simultaneously
- Most operating systems provide a rich set of synchronization primitives

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***
- It ensure that the data structure is modified by only one thread at a time.

Example mutex lock

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

Example mutex lock

- 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 need to wait until the thread holding the lock releases it.

Example mutex lock

- Both routines use the same mutex
- 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
- The other threads will have to wait. This situation is known as a contended mutex

Mutexes and Critical Regions

- 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.

Example Critical Regions

- Assume that an OS 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.
-

Example Critical Regions

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

Example Critical Regions

- 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 to malloc() become thread-safe.

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.
- Where as a mutex lock may sleep if it is unable to acquire the lock

Spin Locks

- ***Advantage*** :spin locks 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.

- ***Disadvantage*** : spin lock will spin on a virtual CPU monopolizing that resource. In comparison, a mutex lock will sleep and free the virtual CPU for another thread to use.

Spin Locks

- Spinning for a short period of time makes that the waiting thread will acquire the mutex lock as soon as it is released.

- Spin for a long period of time consumes hardware resources that could be better used in allowing other software threads to run.

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.
- Ex: 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

- 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 to use available resources
- They can be used for signaling between threads.

Semaphores

- Depending on the implementation
  - the method that acquires a semaphore might be called wait, down, or acquire
  - The method to release a semaphore might be called post, up, signal, or release.

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 need protection with lock.

Readers-Writer Locks

- Sometimes data that is typically read-only needs to be updated.
- A readerswriter lock allows many threads to read the shared data, can lock the readers threads out to allow one thread to acquire a writer lock to modify the data.
- Once a thread has a reader lock, they can read the value of the pair of cells, before releasing the reader lock

Readers-Writer Locks

- A writer cannot acquire the write lock until all the readers have released their reader locks.
- To modify the data, a thread needs to acquire a writer lock. This will stop any reader threads from acquiring a reader lock.
- All the reader threads will have released their lock, and the writer thread actually acquire the lock and is allowed to update the data

Example

```
int readData( int cell1, int cell2 )
    {
        acquireReaderLock( &lock );
        int result = data[cell] + data[cell2];
        releaseReaderLock( &lock );
        return result;
    }
void writeData( int cell1, int cell2, int value )
    {
        acquireWriterLock( &lock );
        data[cell1] += value;
        data[cell2] -= value;
        releaseWriterLock( &lock );
    }
```

Barriers

- There are situations where a number of threads have to 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.

- Ex: 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.

Barriers

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

- 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
  ```

Deadlocks and Livelocks

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

Deadlocks and Livelocks

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

---------------------- Ex: Two Threads in a Deadlock

Deadlocks and Livelocks

- The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order.
- 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.
- It enable thread 1 to acquire B and then eventually release both locks, allowing thread 2 to make progress.

Deadlocks and Livelocks

- A ***livelock*** traps threads in an unending loop releasing and acquiring locks.
- Livelocks shows a mechanism that avoids deadlocks.
- If the thread cannot obtain the second lock it requires, it releases the lock that it already holds

Deadlocks and Livelocks

- The two routines ***update1()*** and ***update2()*** each have an outer loop.
- Routine ***update1()*** acquires lock A and then attempts to acquire lock B
- Routine ***update2()*** does this in the opposite order.
- This is a classic deadlock.
- To avoid it, the routine ***canAquire(),*** returns immediately either having acquired the lock or having failed to acquire the lock.
-

Deadlocks and Livelocks

Thread 1	Thread 2
void update1()	void update2()
{	{
int done=0;	int done=0;
while (!done)	while (!done)
{	{
acquire(A);	acquire(B);
if (canAcquire(B))	if (canAcquire(A))
{	{
variable1++;	variable2++;
release(B);	release(A);
release(A);	release(B);
done=1;	done=1;
}	}
else	else
{	{
release(A);	release(B);
}	}
}	}
}	}

Two Threads in a livelock

Deadlocks and Livelocks

- 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, before attempting to acquire both locks again
- The thread exits the loop when it manages to successfully acquire both locks.

Communication
Between
Threads and Processes

Memory, Shared Memory, and Memory-Mapped Files

- The easiest way for multiple threads to communicate is through memory.
- If two threads can access the same memory location, the cost of that access is little more than the memory latency of the system
- A multithreaded application will share memory between the threads by default, so this can be a very low-cost approach.

Memory, Shared Memory, and Memory-Mapped Files

- The only things that are not shared between threads are variables on the stack of each thread (local variables) .
- Memory accesses need to be controlled to ensure that only one thread writes to the same memory location at a time
- Sharing memory between multiple processes is more complicated

2. Condition Variables

- Condition variables communicate readiness between threads by enabling a thread to be woken up when a condition becomes true.
- Without condition variables, the waiting thread have to use polling to check whether the condition had become true.
- Condition variables work in conjunction with a mutex
- The mutex is there to ensure that only one thread at a time can access the variable.

2. Condition Variables

- The ***producerconsumer*** model can be implemented using condition variables
- Suppose an application has one producer thread and one consumer thread
- The producer adds data onto a queue, and the consumer removes data from the queue.

2. Condition Variables

- ***producer*** thread adding an item onto the queue.

```
Acquire Mutex();
Add Item to Queue();
If ( Only One Item on Queue )
{
Signal Conditions Met();
}
Release Mutex();
```

2. Condition Variables

- The producer thread needs to signal a waiting consumer thread only if the queue was empty and it has just added a new item into that queue.

- If there were multiple items already on the queue, then the consumer thread must be busy processing those items and cannot be sleeping.

- If there were no items in the queue, then it is possible that the consumer thread is sleeping and needs to be woken up.

2. Condition Variables

- Consumer Thread Removing Items from Queue

```
Acquire Mutex();
Repeat
Item = 0;
If ( No Items on Queue() )
{
Wait on Condition Variable();
}
If (Item on Queue())
{
Item = remove from Queue();
}
Until ( Item != 0 );
Release Mutex();
```

2. Condition Variables

- The consumer thread will wait on the condition variable if the queue is empty.

- When the producer thread signals it to wake up, it will first check to see whether there is anything on the queue.

- If there is an item on the queue, then the consumer thread can handle that item; otherwise, it returns to sleep

3. Signals and Events

- Signals are a UNIX mechanism where one process can send a signal to another process and have a handler in the receiving process perform some task upon the receipt of the message.
- Many features of UNIX are implemented using signals
- Windows has a similar mechanism for events. The handling of keyboard presses and mouse moves are performed through the event mechanism.
- Signals and events are optimized for sending limited or no data along with the signal.

3. Signals and Events

- Once the signal handler is installed, sending a signal to that thread will cause the signal handler to be executed.

```
void signalHandler(void *signal)
{
        ...
}
int main()
{
        installHandler( SIGNAL, signalHandler );
        sendSignal( SIGNAL );
}
```

4. Message Queues

- A message queue is a structure that can be shared between multiple processes.
- Messages can be placed into the queue and will be removed in the same order in which they were added.
- Constructing a message queue looks like constructing a shared memory segment.

4.Message Queues

- The first thing needed is a descriptor, typically the location of a file in the file system.
- This descriptor can either be used to create the message queue or be used to attach to an existing message queue.
- Once the queue is configured, processes can place messages into it or remove messages from it.
- Once the queue is finished, it needs to be deleted.

4.Message Queues

- Creating and Placing Messages into a Queue

```
ID = Open Message Queue Queue( Descriptor );
Put Message in Queue( ID, Message );
        ...
Close Message Queue( ID );
Delete Message Queue( Description );
```

5.Named Pipes

- UNIX uses pipes to pass data from one process to another.
- Ex: The output from the command ***ls***, which lists all the files in a directory, could be piped into the ***wc*** command, which counts the number of lines, words, and characters in the input.
- The combination of the two commands would be a count of the number of files in the directory

Named Pipes

- Named pipes can be controlled programmatically.
- Named pipes are file-like objects that are given a specific name that can be shared between processes.
- Any process can write into the pipe or read from the pipe.
- There is no concept of a “message”; the data is treated as a stream of bytes.

Named Pipes

- The method for using a named pipe is much like the method for using a file
- The pipe is opened, data is written into it or read from it, and then the pipe is closed.
- One process needs to actually make the pipe, and once it has been created, it can be opened and used for either reading or writing.
- Once the process has completed, the pipe can be closed

Named Pipes

- Setting Up and Writing into a Pipe

```
Make Pipe( Descriptor );
ID = Open Pipe( Descriptor );
Write Pipe( ID, Message, sizeof(Message) );
...
Close Pipe( ID );
Delete Pipe( Descriptor );
```