Nadar Saraswathi College of Engineering & Technology

NSCET E-LEARNING PRESENTATION

LISTEN … LEARN… LEAD…

COMPUTER SCIENCE AND ENGINEERING

IV YEAR / VIII SEMESTER

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

P.MAHALAKSHMI,M.E,MISTE

ASSISTANT PROFESSOR

Nadar Saraswathi College of Engineering & Technology,

Vadapudupatti, Annanji (po), Theni – 625531.

UNIT II

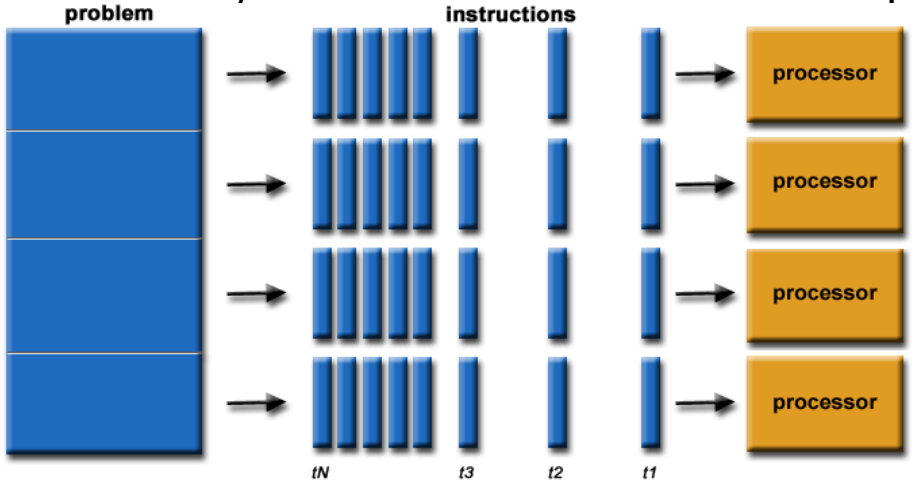
PARALLEL PROGRAM CHALLENGES

Introduction

Parallel Programming

In the simplest sense, ***parallel Programming*** is the simultaneous use of multiple compute resources to solve a computational problem.

- ✓ A problem is broken into discrete parts that can be solved concurrently
- ✓ Each part is further broken down to a series of instructions
- ✓ Instructions from each part execute simultaneously on different processors
- ✓ An overall control/coordination mechanism is employed



Advantages

Advantages of Parallel Computing over Serial Computing are as follows:

- ✓ It saves time and money as many resources working together will reduce the time and cut potential costs.
- ✓ It can be impractical to solve larger problems on Serial Computing.
- ✓ It can take advantage of non-local resources when the local resources are finite.
- ✓ Serial Computing 'wastes' the potential computing power, thus Parallel Computing makes better work of hardware.

Limitations of Parallel Programming

- ✓ It addresses such as communication and synchronization between multiple sub-tasks and processes which is difficult to achieve.
- ✓ The algorithms must be managed in such a way that they can be handled in the parallel mechanism.
- ✓ The algorithms or program must have low coupling and high cohesion. But it's difficult to create such programs.
- ✓ More technically skilled and expert programmers can code a parallelism based program well.

Applications of Parallel Programming

- ✓ Data bases and Data mining.
- ✓ Real time simulation of systems.
- ✓ Science and Engineering.
- ✓ Advanced graphics, augmented reality and virtual reality.

Topic

Performance and Scalability

Performance

- ✓ Parallel programs are used to improve the performance and efficiency of a system.
- ✓ Some measures are used to evaluate the program in performance aspects.

Speedup and efficiency

Speedup is defined as the process for improving the performance by increasing the speed up of execution of a task.

i) Linear speedup of parallel programs

Equally divide the work among the cores - no additional work for the cores.

Run the program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program.

Consider,

serial run-time=Tserial

parallel run-time=Tparallel

Then Tparallel can be calculated as

Tparallel =Tserial/p. - this is difficult to achieve

ii) Speedup of parallel programs

Linear speedup in parallel programs can't achieve for the following reasons

Shared memory systems may have critical section

Usage of mutual exclusion mechanism execute the parallel program in a serialized manner

Distributed memory transmit the data across the network – much slower than local memory access

More number of processes leads more overhead

The speedup(S) of a parallel program to be

$$S = T_{serial} / T_{parallel}$$

iii)Efficiency of the parallel program

Efficiency of the parallel (E) is defined as the ratio between the speedup of parallel program(S) and the number of processors(p) involved.

$$E = S/p = (T_{serial}/T_{parallel}) / p$$

$$E = T_{serial} / p \times T_{parallel}$$

Table - Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E=S/p	1.0	0.95	0.90	0.81	0.68

Tparallel, S, and E depend on p, the number of processes or threads.

- ✓ Parallelization includes some additional overhead such as task distribution, critical section execution, co-ordination, coherency, consistency and so on.
- ✓ Time taken for this overhead is considered as overhead time $\mathbf{T_{overhead}}$ – parallel overhead
- ✓ While calculating parallel runtime , parallel overhead also be considered.

$$T_{parallel} = (T_{serial}/p) + T_{overhead}.$$

iv) Amdhal's Law

The potential speedup gained by parallel execution of a program is limited only by the portion that can be parallelized.

$Fraction_{parallelized}$ – amount of task that can be parallelized

$Fraction_{unparallelized}$ – amount of task that can't be parallelized

$$\mathbf{T_{parallel} = Fraction_{parallelized} \times (T_{serial}/p) + Fraction_{unparallelized} \times T_{serial}}$$

Speedup can be calculated as

$$\mathbf{S = T_{serial} / (Fraction_{parallelized} \times (T_{serial}/p) + Fraction_{unparallelized} \times T_{serial})}$$

Scalability

- ✓ Ability of a system or network to the growing amount of the work
- ✓ Efficiency of the parallel program has been defined as E with a fixed number of cores and fixed input size.
- ✓ Able to maintain the efficiency as E even if the rate of problem size is increased, then it can be concluded that the program is scalable.

Suppose that,

Tserial =n

where the units of

Tserial=microseconds

n=problem size.

Also suppose that,

Tparallel =(n/p)+1.

Then

$E= n / p((n/p)+1) = n/ (n+p)$

To see if the program is scalable, we increase the number of

If the program is scalable,

Increase the number of processes/threads by a factor of k

To find the factor x that we need to increase the problem size by so that E is unchanged.

The number of processes/threads = kp

the problem size=xn,

to solve the following equation for x:

$$E = n/(n+p) = xn/(xn+kp)$$

if x=k

$$E = kn/(kn+kp)$$

$$= kn/k(n+p)$$

$$E = n/(n+p)$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

Strongly Scalable

If increase the number of processes/threads, we can keep the efficiency fixed without increasing the problem size, the program is said to be strongly scalable.

Weekly scalable

If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be weakly scalable

- ✓ The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.
- ✓ The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. This is a common situation with many parallel applications.
- ✓ Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- ✓ Parallel support libraries and subsystems software can limit scalability independent of your application

Topic

Synchronization and data sharing – Data Races

Synchronization and Data Sharing

- ✓ For a multithreaded application to do useful work, it is usually necessary for some kind of common state to be shared between the threads.
- ✓ The degree of sharing that is necessary depends on the task. At one extreme, the only sharing necessary may be a single number that indicates the task to be performed.
- ✓ There are various methods for sharing data between threads and the costs of these approaches.
- ✓ It starts with a discussion of *data races*, which are situations where multiple threads are updating the same data in an unsafe way.
- ✓ One way to avoid data races is by utilizing proper synchronization between threads.

Data Races

Definition

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 programming errors in parallel code.

Example 1

In the following program a pointer to an integer variable is passed in and the function increments the value of this variable by 4.

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

The SPARC disassembly for the above code is given below

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

Assume that the same code occurs in a multithreaded application and two threads try to increment the same variable at the same time as below

Value of variable a = 10	
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	

Though each thread adds 4 to the variable, the final value will be 14 as two thread had executed the code at the same time. This is the situation where both threads are running simultaneously. If the two threads had executed the code at different times, then the variable would have ended up with the value of 18.

Tools to Detect Data Races

- ✓ Data races can be hard to find. The problem is hard to see from code inspection. At the same time, there are tools to detect data races.
- ✓ Consider the following code creates two POSIX threads. Both threads execute the routine func(). The main thread then waits for both the child threads to complete their work.

Program: race.c

```
#include <pthread.h>
int counter = 0;
void * func(void * params)
{
counter++;
}
void main()
{
pthread_t thread1, thread2;
pthread_create( &thread1, 0, func, 0);
pthread_create( &thread2, 0, func, 0);
pthread_join( thread1, 0 );
pthread_join( thread2, 0 );
}
```

Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc.

Tool : Helgrind

Helgrind, which is part of the Valgrind1 suite, can be used to identify the data races. Valgrind is a tool that enables an application to be instrumented and its runtime behavior examined. The Helgrind tool uses this instrumentation to gather data about data races.

Output of Helgrind

```
$ gcc -g race.c -lpthread
$ valgrind -–tool=helgrind ./a.out
...
==4742==
==4742== Possible data race during write of size 4 at 0x804a020 by thread #3
==4742== at 0x8048482: func (race.c:7) ==4742== by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742== by 0x40414FE: start_thread (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742== by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
==4742== This conflicts with a previous write of size 4 by thread #2
==4742== at 0x8048482: func (race.c:7) ==4742== by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742== by 0x40414FE: start_thread (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742== by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
```

The output from Helgrind shows that there is a potential data race between two threads, both executing line 7 in the file race.c.

Limitation of Helgrind

The tool will find some false positives. The programmer may write code where different threads access the same variable, but the programmer may know that there is an enforced order that stops an actual data race. The tools, however, may not be able to detect the enforced order and will report the potential data race.

Tool : Thread Analyzer

This tool is available with Oracle Solaris Studio. This tool requires an instrumented build of the application, data collection is done by the **collect tool**, and the graphical interface is launched with the **command tha**. Steps to detect Data Races using the Sun Studio Thread Analyzer is given below.

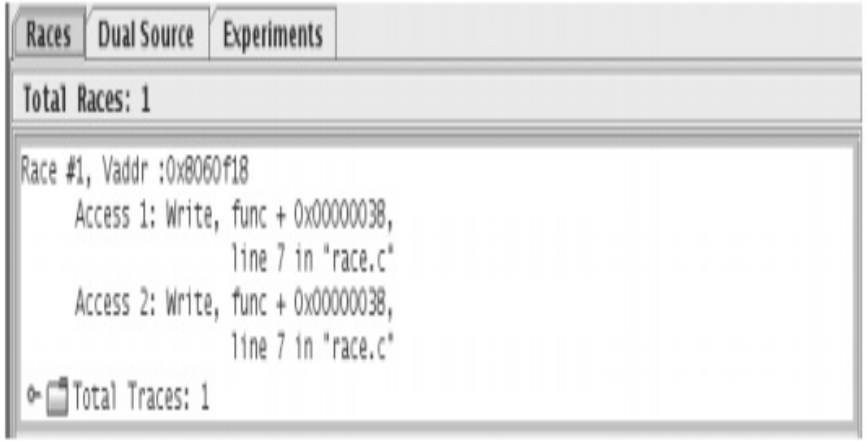
$ cc -g -xinstrument=datarace race.c

$ collect -r on ./a.out

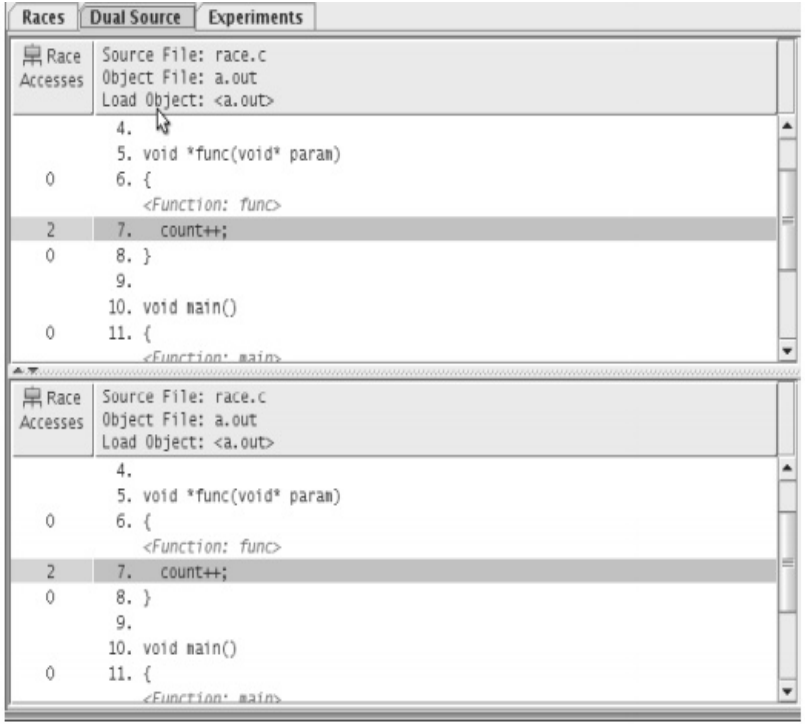
Recording experiment tha.1.er ...

$ tha tha.1.er&

The initial screen of the tool displays a list of data races, as shown in the following figure.



Once the user has identified the data race they are interested in, they can view the source code for the two locations in the code where the problem occurs



Avoiding Data Races

- ✓ Data races can be avoided easily. It is necessary to ensure that only one thread can update the variable at a time. The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.
- ✓ The following program shows a modified version of the code which uses a **mutex lock** to protect accesses to the variable counter.

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

- ✓ Although this ensures the correctness of the code, it does not necessarily give the best performance

Topic

Synchronization Primitives

Introduction

- ✓ Synchronization is used to coordinate the activity of multiple threads.
- ✓ It ensures 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.

Advantages of using synchronization primitives

i) They are recognized by the tools that can detect data races and labels synchronization cost provided with that operating system

ii) Operating system will support sharing of these primitives between threads or processes

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

Example for Mutex

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

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

Contended mutex

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.

Critical section, or Critical region

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 for Critical Section

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

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.

Limitations

It can serialize a program. If multiple threads simultaneously call threadSafeMalloc(), only one thread at a time will make progress. So, it stops the program from taking advantage of multiple cores.

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.

Advantage

- ✓ 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.

Disadvantage

- ✓ A 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.

Spinning for a short period of time makes it more likely 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 that could be better used in allowing other software threads to run.

Semaphores

- ✓ Semaphores are counters that can be either incremented or decremented.

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.

Readers-Writer Locks

- ✓ 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.

- ✓ 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.

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 all complete their work before any of the threads can start on the next task.

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.

The following code 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();
```

The variable total can be computed only when all threads have reached the barrier.

Atomic Operations and Lock Free Code

- ✓ Using synchronization primitives can add a high overhead cost.
- ✓ This is particularly true if they are implemented as calls into the operating system rather than calls into a supporting library.
- ✓ 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 functionally equivalent code without introducing the same amount of overhead.

Atomic operation

An atomic operation is one that will either successfully complete or fail; it is not possible for the operation to either result in a "bad" value or allow other threads on the system to observe a transient value.

Example

This would be an atomic increment, which would mean that the calling thread would replace a variable that currently holds the value N with the value N+1.

The operation of incrementing a variable can involve multiple steps, as shown below**.**

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

Lock-free code

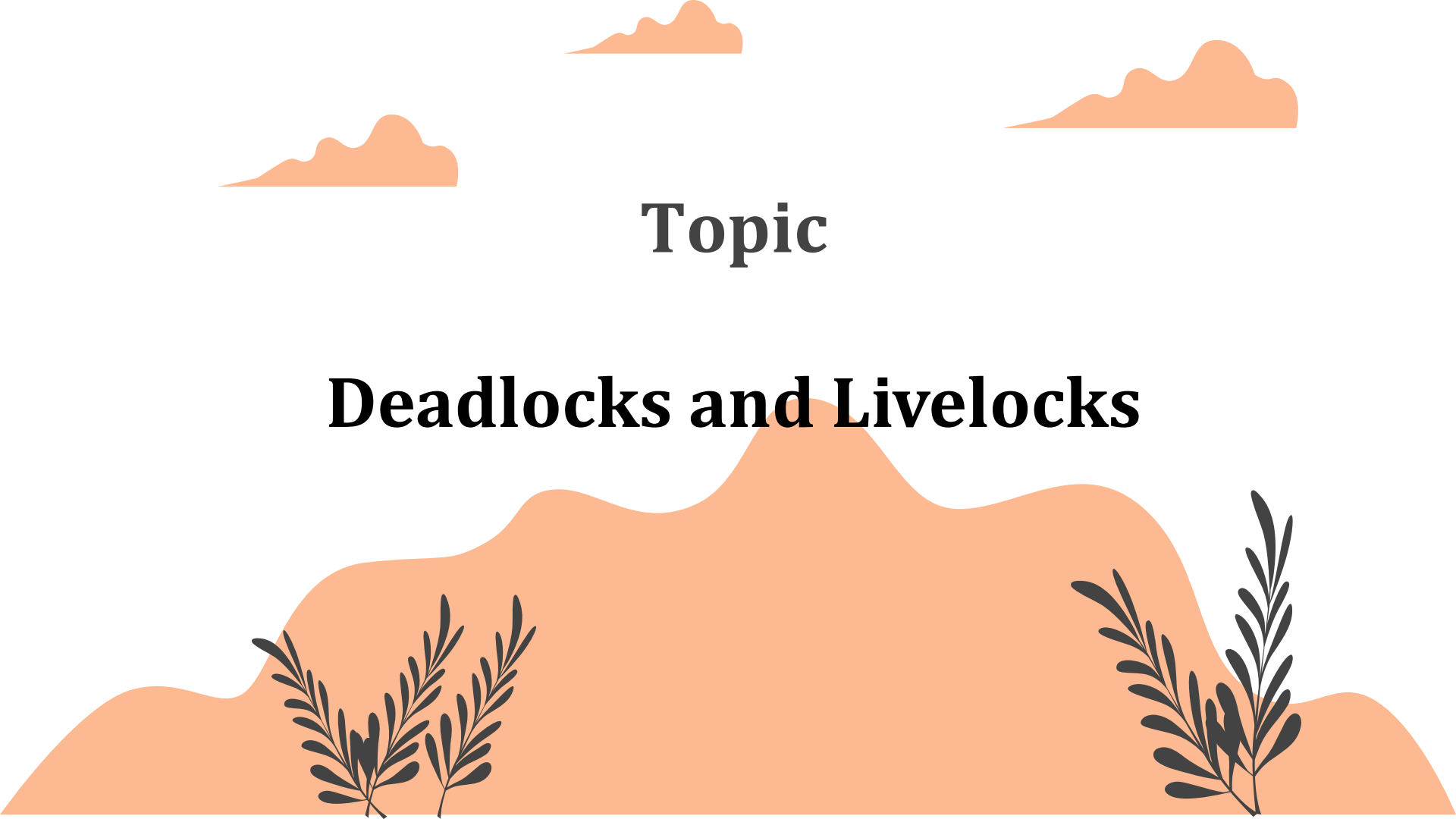
- ✓ Atomic operations are often used to enable the writing of lock-free code
- ✓ A lock-free implementation would not rely on a mutex lock to protect access; instead, it would use a sequence of operations that would perform the operation without having to acquire an explicit lock.
- ✓ This can be higher performance than controlling access with a lock.
- ✓ Most of the more complex atomic operations are actually lock-free implementations.
- ✓ They use a low-level, hardware-provided atomic operation and wrap code around that to ensure that the required higher-level operation is actually atomic.

Example

A low-level atomic operation would be compare and swap (CAS), which atomically swaps the value held in a register with the value held in memory if and only if the value held in memory matches the expected value

Topic

Deadlocks and Livelocks



Introduction

The fundamental ways to share access to resources between threads :

- ✓ Deadlock
- ✓ Livelocks

Deadlock

- ✓ The 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. The following code shows the situation.

Thread 1

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

Thread 2

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

Avoiding 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.

Livelock

- ✓ A livelock traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks.
- ✓ The programmer has tried to implement a mechanism that avoids deadlocks as below. 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, the developer has written some code that causes the held lock to be released if it is not possible to acquire the second lock.
- ✓ The routine canAquire( ), in this example, returns immediately either having acquired the lock or having failed to acquire the lock.

Two Threads in a Livelock

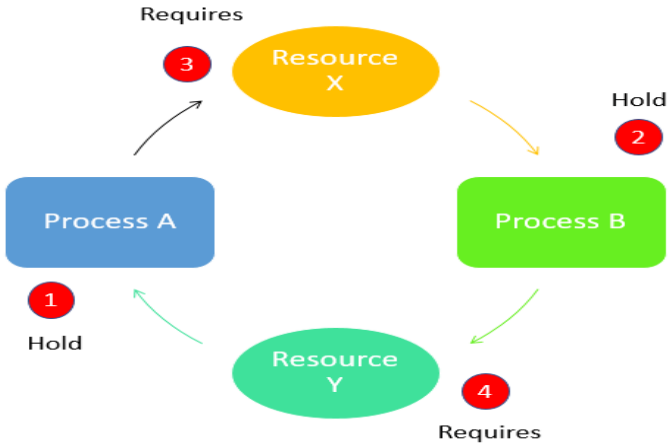
Thread 1

```
void update1()
{

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

Thread 2

```
void update2()
{

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

Example of Livelock



- ✓ In the above image, each of the two given processes needs two resources, and they use the primitive polling enter registry to try to acquire the locks necessary for them. If the attempt fails, the method works again.
  - Process A hold Y resource
  - Process B holds resource X
  - Process A require X resource
  - Process B require Y resource
- ✓ Assuming, process A runs first and acquires data resource X and then process B runs and acquires resource Y, no matter which process runs first, none of them further progress.
- ✓ However, neither of the two processes are blocked. They use up CPU resources repeatedly without any progress being made but also stop any processing block.
- ✓ Therefore, this situation is not that of a deadlock because there is not a single process that is blocked, but we face the situation something equivalent to deadlock, which is LIVELOCK.

Topic

Communications between Threads and Processes

Introduction

All parallel applications require some element of communication between either the threads or the processes. There is usually an implicit or explicit action of one thread sending data to another thread.

Example

one thread might be signaling to another that work is ready for them. These mechanisms usually require operating system support to mediate the sending of messages between threads or processes.

Memory

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. Of course, memory accesses still need to be controlled to ensure that only one thread writes to the same memory location at a time. A multithreaded application will share memory between the threads by default, so this can be a very low-cost approach.

Shared Memory

- ✓ Sharing memory between multiple processes is more complicated.
- ✓ By default, all processes have independent address spaces, so it is necessary to preconfigure regions of memory that can be shared between different processes.
- ✓ To set up shared memory between two processes, one process will make a library call to create a shared memory region. The call will use a unique descriptor for that shared memory. This descriptor is usually the name of a file in the file system.
- ✓ The create call returns a handle identifier that can then be used to map the shared memory region into the address space of the application.
- ✓ This mapping returns a pointer to the newly mapped memory.
- ✓ This pointer is exactly like the pointer that would be returned by malloc() and can be used to access memory within the shared region. When each process exits, it detaches from the shared memory region, and then the last process to exit can delete it.

The following code shows the rough process of creating and deleting a region of shared memory.

Creating and Deleting a Shared Memory Segment
ID = Open Shared Memory(**Descriptor**);
Memory = Map Shared Memory(ID);
...
Memory[100]++; ... Close Shared Memory(ID);
Delete Shared Memory(Descriptor);

✓ The following code shows the process of attaching to an existing shared memory segment. In this instance, the shared region of memory is already created, so the same descriptor used to create it can be used to attach to the existing shared memory region.

Attaching to an Existing Shared Memory

Segment ID = Open Shared Memory(**Descriptor**);
Memory = Map Shared Memory(ID);
...
Close Shared Memory(ID);

A shared memory segment may remain on the system until it is removed

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 would have to use some form of 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.

Example

The producer consumer 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. If there is no data on the queue, then the consumer needs to sleep until it is signaled that an item of data has been placed on the queue

- ✓ The following pseudocode shows how a 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( );
```

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.

The following code segment shows the pseudocode for the consumer thread.

Code for 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();
```

- ✓ The interaction with the mutex is interesting. The producer thread needs to acquire the mutex before adding an item to the queue. It needs to release the mutex after adding the item to the queue, but it still holds the mutex when signaling.
- ✓ The consumer thread cannot be woken until the mutex is released. The producer thread releases the mutex after the signaling has completed; releasing the mutex is necessary for the consumer thread to make progress.

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. Stopping a running application by pressing ^C causes a SIGKILL signal to be sent to the process.
- ✓ Windows has a similar mechanism for events. The handling of keyboard presses and mouse moves are performed through the event mechanism. Pressing one of the buttons on the mouse will cause a click event to be sent to the target window.
- ✓ Signals and events are really optimized for sending limited or no data along with the signal, and as such they are probably not the best mechanism for communication when compared to other options.

The following code shows how a signal handler is typically installed and how a signal can be sent to that handler. Once the signal handler is installed, sending a signal to that thread will cause the signal handler to be executed.

Installing and Using a Signal Handler

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

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 rather like constructing a shared memory segment.
- ✓ 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.

- ✓ The following code shows code for creating and placing messages into a queue.
- ✓ 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);

- ✓ The following code shows the process for receiving messages for a queue. Using the descriptor for an existing message queue enables two processes to communicate by sending and receiving messages through the queue.
- ✓ Opening a Queue and Receiving Messages

ID=Open Message Queue ID(Descriptor);

Message=Remove Message from Queue(ID);

...

Close Message Queue(ID);

Named Pipes

UNIX uses pipes to pass data from one process to another.

Example

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 provide a similar mechanism that 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.
- ✓ 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.
- ✓ The following code shows the steps necessary to set up and write data into a pipe, before closing and deleting the pipe.
- ✓ 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, and one of the processes using it should also be responsible for deleting it.

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 );
```

The following code shows the steps necessary to open an existing pipe and read messages from it. Processes using the same descriptor can open and use the same pipe for communication.

Opening an Existing Pipe to Receive Messages

```
ID=Open Pipe( Descriptor );
Read Pipe( ID, buffer, sizeof(buffer) );
...
Close Pipe( ID );
```

Communication Through the Network Stack

- ✓ Communication across the network usually involves a client-server model. To set up a server, it is first necessary to open a socket and then bind that socket to the address on the local host before starting to listen for incoming connections.
- ✓ When a connection arrives, data can be read from it or written to it, until the connection is closed. Once the connection is closed, it is possible to close the socket.
- ✓ The following code illustrates how the server thread of a client-server network connection can be set up.

Setting Up Socket to Listen for Connections

```
ID = Open Socket( Descriptor );
Bind Socket( ID, Address );
Listen( ID ) Conx = Wait for connection( ID );
Read( Conx, buffer, sizeof(buffer) );
...
Close( Conx ); Close Socket( ID );
```

- ✓ The following code shows the steps necessary to set up a client socket to connect to the server. Connecting to a remote server also requires initially setting up a socket.
- ✓ Once the socket is open, it can be used to connect to the server. After the communication is complete, the socket can be closed.
- ✓ Setting Up a Socket to Connect to a Remote Server

```
ID=Open Socket( Descriptor );
Connect( ID, Address );
Write( ID, buffer, sizeof(buffer) );
...
Close( ID );
```

Other Approaches to Sharing Data Between Threads

- ✓ Files: Data can be written to a file to be read by another process at a later point.
- ✓ Doors: Solaris doors allow one process to pass an item of data to another process and have the processed result returned. Doors are optimized for the round-trip and hence can be cheaper than using two different messages.

Thank You