Data Races

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

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

Fig1 :Updating the Value at an Address

- The SPARC disassembly for this code is shown in fig 2

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

Fig 2.SPARC Disassembly for Incrementing a Variable Held in Memory

- Suppose this code occurs in a multithreaded application and two threads try to increment the same variable at the same time. Table .1 shows the resulting instruction stream
- Fig 3 :Two Threads Updating the Same Variable (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	

- In the example, each thread adds 4 to the variable, but because they do it at exactly the same time, the value 14 ends up being 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. This is the situation where both threads are running simultaneously. This illustrates a common kind of data race .

- Another situation where one thread holds the value of a variable in a register and a 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.

- The point is that a data race situation is created whenever a variable is loaded and another thread stores a new value to the same variable: One of the threads is now working with “old” data.

- Data races can be hard to find. Take the previous code example to increment a variable. It might reside in the context of a larger, more complex routine. It can be hard to identify the sequence of problem instructions just by inspecting the code.

- Not only is the problem hard to see from inspection, but the problem would occur only when both threads happen to be executing the same small region of code.

Tools to Detect Data Races

- The code shown in fig 3 contains a data race. The code uses POSIX threads, Using POSIX Threads. The code creates two threads, both of which execute the routine func(). The main thread then waits for both the child threads to complete their work.

- ```
  #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 );
  ```
  ```

}

Fig 3: Code Containing Data Race

- Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc and then use **Helgrind**, which is part of the **Valgrind1** suite, to identify the data race.
- 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.

- Using Helgrind to Detect Data Races

```
$ 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)
```

Fig 4 below shows the output from Helgrind

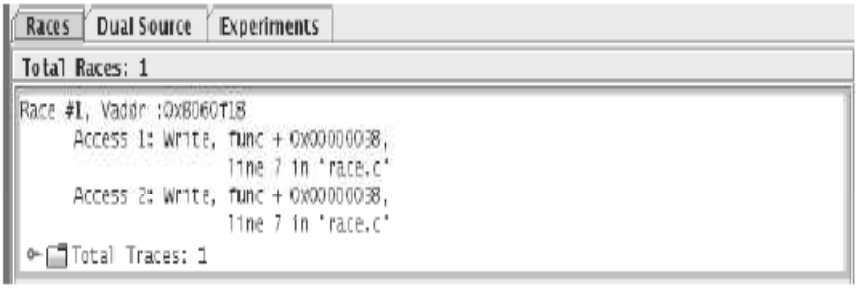
- The output from Helgrind shows that there is a potential data race between two threads, both executing line 7 in the file ***race.c.*** This is the anticipated result, but it should be pointed out that the tools will find some false positives. The programmer may write code where different threads access 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.
- Another tool that is able to detect potential data races is the Thread nalyzer in 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***.

- Fig 5 below shows the steps to do this.

```
$ cc -g -xinstrument=dataracerace.c
$ collect -r on ./a.out
Recording experiment tha.1.er ...
$ tha tha.1.er&
```

Fig 5 :Detecting Data Races Using the Sun Studio Thread Analyzer



Avoiding Data Races

- Although it can be 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 around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock
- Code below shows a modified version of the code. This version uses a ***mutex lock***, described in more detail in the next section, to protect accesses to the variable counter.

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

Fig 6: Code Modified to Avoid Data Races