Inc Dec Code Review

Tanner Johnson

Contents

- This powerpoint as a substitute for the video
- Inc_dec.c contains cod
- Makefile to build application
- Will go over the code
- The two key problems this code exemplifies
- Show how to reproduce this issues
- Code output

Code: Includes

```c
#include <pthread.h>  // pthread_create, pthread_join
#include <stdio.h>    // printf
#include <stdlib.h>   // exit
```

- Pretty straight forward. See hello submission for details on these includes and the system calls they give us access too

Code: Globals

```
// Set the count s.t. gsum will not overflow in the worst case i.e. where inc
// or dec is ran to completion followed by the other without any interleaving.
// So we need | gsum | < 2^63 for a signed 64 bit gsum counter.
// The worst case value is (COUNT+1)(COUNT)/2 < (COUNT)^2
// If we set COUNT = 2^20 = (1024*1024),
// then we can bound the worst value to 2^40.
// This gives us a lot of iterations, 2^20, but ensures no overflow behavior.
#define COUNT  (1024)*(1024)

// Unsafe global counter.
long long gsum=0;

// Whether or not to print in the inc/dec thread the partial sums on each iter.
// Turn on to see the undeterministic behavior of SCHED_OTHER w/ low COUNT
// Turn off to see speed up execution when COUNT is large to see race condition
#define PRINT_IN_THREAD 0
```

Code: Globals (cont)

- COUNT is the # of iterations the inc and dec thread go through incrementing and decrementing respectfully
  - Note that on each iteration, it incs or decs the global sum by the iteration # i.e. 1,2,3,.. Not by one each time
  - The comment block above explains how to set this appropriately so we get a large number of iterations without overflowing the 64 bit signed value
- gsum. The unsigned 64b value that is added and subtracted too. As commented this variable is unprotected and modifying is not thread safe
- PRINT_IN_THREAD. Will make sense when one sees the thread code. 0 not to print in thread. 1 to print partial sums ever iteration

Code: Threads

- Pretty straight forward
- Parse ID
- Then loop COUNT times
- Adding or subtracting

  The iteration # to gsum

- Print partial sum if set

```c
// Simply increment the global sum in a loop
void *incThread(void *threadp)
{
    int i;
    int idx = *((int*)threadp);

    for(i=0; i<COUNT; i++)
    {
        gsum += (long long) i;

        // Will on compile in the print statement if macro is set
        #if PRINT_IN_THREAD
            printf("Increment thread idx=%d, iter=%d, gsum=%lld\n", idx, i, gsum);
        #endif
    }
}

// Simply decrement the global sum in a loop
void *decThread(void *threadp)
{
    int i;
    int idx = *((int*)threadp);

    for(i=0; i<COUNT; i++)
    {
        gsum -= (long) i;

        // Will on compile in the print statement if macro is set
        #if PRINT_IN_THREAD
            printf("Decrement thread idx=%d, iter=%d, gsum=%lld\n", idx, i, gsum);
        #endif
    }
}
```

Code: Main

- Check if 64b CPU
- Spawn threads
- Wait till they exit
- Print gsum final val
- See hello code for

  Details on pthread

  calls

```c
int main (int argc, char *argv[])
{
    // Just check if machine is 64bit. If its not gsum will overflow.
    if(sizeof(gsum) != 8)
    {
        printf("Warning, not running on 64bit machine!\n");
        exit(1);
    }

   pthread_t threads[2];
   int id0 = 0;
   int id1 = 1;

   // Spawn the inc / dec threads using the SCHED_OTHER scheduler
   pthread_create(&threads[0], NULL, incThread, (void *)&id0);
   pthread_create(&threads[1], NULL, decThread, (void *)&id1);

   // Wait on them to exit
   pthread_join(threads[0], NULL);
   pthread_join(threads[1], NULL);

   // Print the final count. Will = 0 if nothing bad happens. If race cond
   // triggered then this will be non zero
   printf("TEST COMPLETE: gsum=%lld\n", gsum);
}
```

Problem 1) Race Condition on gsum

- A race condition is where the ordering of events changes the final state.
- In this case, the time in which the threads get interrupted and interleaved can cause undefined behavior.
- Specifically, one expects gsum to be 0 at end of execution as inc and dec run the same number iterations.
- However if one gets interrupted during this increment/ decrement operation this will cause the this operation to not persist i.e. it will be overwritten.
- Please read the comment block on the top of inc_dec.c to see this in detail.

Problem 1) Race Condition on gsum, reproducing

- Set COUNT = 1024*1024
  - The comment block above COUNT explains why this is a good value i.e. won't overflow gsum and gives us a lot of time to produce the race condition
- Set PRINT_IN_THREAD to 0 so we are not printing 2^20 printfs
- Run:

```
 inc_dec> make
gcc -O0 -g   -c inc_dec.c
gcc  -O0 -g   -o inc_dec inc_dec.o -lpthread
 inc_dec> ./inc_dec
TEST COMPLETE: gsum=-533134503868
 inc_dec> ./inc_dec
TEST COMPLETE: gsum=91855642295
 inc_dec> ./inc_dec
TEST COMPLETE: gsum=-109724289167
 inc_dec> ./inc_dec
TEST COMPLETE: gsum=240006928749
```

Problem 2) Undertiminstic Scheduling

- Threads are created using the SCHED_OTHER, the linux CFS scheduler
- This causes the threads to be scheduled at seemingly random times and intervals
- To see this behavior set COUNT to a low value such as 100
- Set PRINT_IN_THREADS to 1
- And run: make & ./inc_dec
- Run it a few times and you see that the two threads do no get scheduled in a deterministic manner.