Computer Architecture, Operating Systems, and Networks: Handin #7

Nicklas Blom: 20093011/au282254
Anders Alnor: 201804681/au611509
Casper S: 201805311/au602242

Aarhus Universitet — April 4, 2020

Prime numbers are among the most fascinating mathematical objects and there are still a lot of unanswered questions about them. One of the common intuitions about them is that they are distributed "like random numbers" unless there is a simple reason against it. Here, we would like to test this intuition. For instance, the only even prime number is 2 which means that no prime number will have 4, 6, or 8 as their last digit (in decimal) and the only prime number with 2 as its last digit is 2 itself. Similarly, there is only one prime number that ends with digit 5 (the number 5 itself). However, we cannot think of a simple reason why a prime number should not end in digits 1, 3, 7, or 9. So if our intuition is correct, then the prime numbers ending in each of these digits should be more or less evenly distributed. Here, you will write a program to test this intuition. We have seen how using multiple threads we can greatly speed up the running time of computational tasks. However, a very big challenge is that changing a global variable or object using multiple threads can cause race conditions where the changes applied by one thread overwrite the changes done by another thread. In this hand in, you will use non-blocking compare and swap instructions to avoid race conditions. Since this solution crucially relies on instructions supported by the hardware, you will need to implement parts of it in assembly language.

The compare and swap technique we will use to avoid threads overwriting each others' work, is as described in the hand-in instructions, through an atomic swap using the *lock cmpxchg16b* instruction.

Part I
The Code

1 The C Code

All our code is based on the template starting points from the hand-in instructions. Let's begin by examining out main C file, the one called cmp_swap_template.c.

```c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

// C has no boolean type so we define it.
#define true 1
#define false 0
// We define a boolean to be of type 'short int'
typedef short bool;

```

```
/* We will count prime numbers up to this value
 Start small for testing, then increase to make sure that
 when computing with one thread the program takes about
 10 seconds, otherwise, the noise in the time calculation
 might cancel out any meaningful pattern.
 */
#define MAX 10000000

int tnum; // The number of threads

/* We will need to maintain the number of primes of a particular
 type.
 */
typedef struct ptypes_st {
    int n1;
    int n3;
    int n7;
    int n9;
} ptypes_t;

/* Or you can do:
typedef struct ptypes_st {
    int types[4];
} ptypes_t;
*/


/* Here we need to define another struct to make the
 compare and swap work.
 In particular, we need a struct that is a combination of a counter and ptypes_t* type.
 We need another global variable of that type.

FILL HERE!!
*/
typedef struct pair_st {
    long counter;
    ptypes_t* dataPointer;
} pair_t;

pair_t* global_pair;


//datastructure to be parsed into the thread_function
typedef struct threadData_st {
    int lNumber;
    int uNumber;
} threadData_t;


/* We will dynamically allocate an array of pthread_t objects
 Thus, "threads" will be an array of pthread_t objects.
 */
pthread_t* threads;

/* A rather slow way to test if a number if prime
 Don't change this function
 */
bool isPrime(long n){
    if (n==1) return false;
    long test=2;
    while (test*test <= n) {
        if (n%test == 0) return false;
        test++;
    }
    return true;
}

/* An external function written in assembly that does
 compare and swap for us.

 It accepts three parameters of some yet unknown type that need to be
 defined somewhere.
```

```c
 For the documentation of the function, see the assembly file.
 Remember to pay attention to the order by which the parameters are passed
 in System V ABI
 */
extern int my_cmpr_swap(pair_t* old, pair_t* cur, pair_t* mod);


void* thread_function(threadData_t* arg){
    /* Here we loop over all the integers in the range 1-MAX that are assigned
     to us (the current thread).
     if we find a problem number i then we have to atomically
     update the total number of prime numbers of particular type.
     */
    threadData_t data = *arg;
    int lower = data.lNumber;
    int upper = data.uNumber;

    ptypes_t* temp = malloc(sizeof(ptypes_t));
    temp->n1 = 0;
    temp->n3 = 0;
    temp->n7 = 0;
    temp->n9 = 0;

    int i;
    for(i=lower ; i<upper ; i++){
        if (isPrime(i)) {
            int value = i%10;
            if(value == 1) {
                temp->n1++;
                continue;
            }
            if(value == 3) {
                temp->n3++;
                continue;
            }
            if(value == 7) {
                temp->n7++;
                continue;
            }
            if(value == 9) {
                temp->n9++;
                continue;
            }
        }
    }


    pair_t* localCopy = malloc(sizeof(pair_t));

    pair_t* modifiedObject = malloc(sizeof(pair_t));

    int j = 0;
    while(j == 0){

        localCopy->counter = global_pair->counter;
        localCopy->dataPointer = global_pair->dataPointer;

        ptypes_t* swap = malloc(sizeof(ptypes_t));
        swap->n1 = temp->n1 + localCopy->dataPointer->n1;
        swap->n3 = temp->n3 + localCopy->dataPointer->n3;
        swap->n7 = temp->n7 + localCopy->dataPointer->n7;
        swap->n9 = temp->n9 + localCopy->dataPointer->n9;
        modifiedObject->counter = 1 + localCopy->counter;
        modifiedObject->dataPointer = swap;

        j = my_cmpr_swap(localCopy,  global_pair, modifiedObject);

        free(localCopy->dataPointer);
    }

    free(temp);
    free(localCopy);
```

```
    free(modifiedObject);
}

int main(int argc, char **args){
    // Initialize the global variable
    ptypes_t* startingPoint = malloc(sizeof(ptypes_t));
    startingPoint->n1 = 0;
    startingPoint->n3 = 0;
    startingPoint->n7 = 0;
    startingPoint->n9 = 0;


    if (argc != 2) {
      printf("You need to specify the number of threads.\n");
      exit(-1);
    }
    tnum=atoi(args[1]);
    if (tnum <1) tnum=1;
    /* tnum is now the number of threads...
    fill here!
    */

    float divisor = MAX/(float)tnum;
    int upper = 0;
    threadData_t* dataSet = malloc(tnum*sizeof(threadData_t));


    /* Probably we need to do a bit more work here ...
    Since we will need to define one more global variable.
    */
    global_pair = malloc(sizeof(pair_t));
    global_pair->counter = 0;
    global_pair->dataPointer = startingPoint;

    // An array of threads.
    pthread_t* threads = malloc(tnum*sizeof(pthread_t));

    int i;
    for (i=0; i<tnum ; i++){
        // Spawn the i-th thread.
        int lower = upper;
        upper = divisor*(i+1);
        dataSet[i].lNumber = lower;
        dataSet[i].uNumber = upper;
        int code = pthread_create(&threads[i], NULL, thread_function, &dataSet[i]);
        if (code) {
            printf("Something went wrong, aborting. \n");
            exit(-1);
        }
    }

    // We need to wait for the threads to finish their work.
    for (i=0; i<tnum; i++){
        pthread_join(threads[i], NULL);
    }
    free(dataSet);
    free(threads);

    //To avoid having to follow global_pair's pointer to the dataPointer every time we just mak
    ptypes_t* resultsFinal = global_pair->dataPointer;
    long int result = (resultsFinal->n1 + resultsFinal->n3 + resultsFinal->n7 + resultsFinal->n9
    printf("The result was %ld primes numbers ending with 1,3,7 or 9 in the range of [0,9999999        result
    printf("Primes numbers ending with 1: %d \n.", resultsFinal->n1);
    printf("Primes numbers ending with 3: %d \n.", resultsFinal->n3);
    printf("Primes numbers ending with 7: %d \n.", resultsFinal->n7);
    printf("Primes numbers ending with 9: %d \n.", resultsFinal->n9);
    return 0;
}
```

To break this down into more understandable chunks, let's start by looking at main and following along with normal program execution.

First, we create a starting ptypes pointer to give to our global pair later. This will only be a starting point,

since once the first thread runs its swap, this data will no longer be needed, but we require an initial state for now.
Skipping the parts that are no different to our last assignment, moving on to line 185, we now initiate the global pair that all our threads will be dealing with. Initialising its counter to 0 and setting the dataPointer to our newly created starting point of all 0s.
Between line 194 and 205 we set up the threadData that each thread will be given, which is the upper and lower bound for their computation and we then spawn all these threads, with the thread_function in which all the magic happens.

Finally, in the end, we wait for all the threads to finish, free the last bits of information we couldn't free while we still had running threads and print the output.

Now of course we still need to discuss the primary part of the program; The thread function. 91.
Firstly, I will note that while the function call to create the threads expect the thread_function to take void* as its argument, we instead specify that we want threadData*. This will make the compiler show a warning, but not an error. We can do this, since void* is just any memory location anyway, so we're just narrowing it down. Conceptually a little bit like subclassing, even though that's of course not what it is since C is not object oriented and has no concept of a class like that.
With that out of the way let's see what the function does. Initially, we set up some data. Extract the passed along upper and lower bounds, and create a temp ptypes, where we'll locally count the number of primes we find ending with each of the checked digits. We then perform the actual checks, updating our local temp struct. Finally, now that the local temp struct contains the number of primes for this threads' range, we can attempt to add it to the global count. We do this by first creating a local copy of the global pair, and then adding our temp values to what we see in this local copy. All this goes to a swap structure. Finally, we add 1 to the counter signifying that we have have modified the pair, and add the new counter and swap struct to a pair we call modifiedObject. We run our assembly code, which checks if the local copy is still the same as the global pair, and if it is, it means that no other thread has made a swap while these computations were happening, so we can swap our modified object for the global pair. If our assembly instruction returns false however, indicating that the local and global copy were different, no swap occurs, and all the local copy is reset to the global, and we construct a new swap and modified object trying over again until it works.
When we finally make the swap successfully, we free all the resources we no longer need; The local copy, the modifiedObject and the temp. - However, if the swap does not work, we in fact also perform a free, clearing away just the dataPointer of the localCopy. Since the swap wasn't successful, we know that whatever this pointer is referencing has already been invalidated by another thread, so we shouldn't cling on to the local copy either.

2 The Assembly Code

Now let's have a look at the assembly instructions we call within the thread_function

```
.text
.global my_cmpr_swap


# Parameters:
# rdi: old status
# rsi: cur status
# rdx: mod status
#
# We want to use "lock cmpxchg16b (reg)" instruction
# where reg is some register.
# cmpxchg16b: Compares rdx:rax (as a 128 bit integer) with the 128
# bit integer starting at address reg.
# We denote this 128 bit integer by m128.
# If rdx:rax equals m128, then the instruction sets the Z flag
```

```
# and copies rcx:rbx (as a 128 bit integer) into m128.
# otherwise, it clears the Z flag and copies the m128 into
# the registers rdx:rax.
#
# VERY IMPORTANT!!! Remember that intel is Little Endian!
#
# This means, the first 64 bits at address reg correspond
# to the lower half (or the lowest 64 bits) of m128!
# This means, when comparing rdx:rax as a 128 bit integer to
# m128, rax is compared to the 64 bit integer at (reg)
# and rdx is compared to the 64 bit integer at (reg+8).
# Similarly, when rcx:rbx is copied into m128,
# rbx is copied into the 64 bit integer starting at (reg)
# and rcx is copied into the 64 bit integer at (reg+8)

my_cmpr_swap:
    # First we save rbx,
    # since rbx needs to be used and by
    # calling conventions, it is our job to save and restore it
    pushq %rbx

    # you need to figure out which reg to use and how to
    # set up registers rdx:rax, and rcx:rbx
    movq (%rdx), %rbx
    movq 8(%rdx), %rcx
    movq (%rdi), %rax
    movq 8(%rdi), %rdx

    lock cmpxchg16b (%rsi)

    # If Z flag is set are successful so we return 1.
    # else we return 0
    # Could avoid jmp entirely with conditional mov logic
    jz success
    movq $0,%rax
    jmp end

success:
    movq $1,%rax

end:
    popq %rbx
    ret
```

The comments in the code are already fairly explicit, but to elaborate a bit more what we do, is simply to set up the registers with the correct information from memory. Since RSI holds the starting address to the current global object in memory, we set up everything else to compare and swap against that memory location. We do this by copying the local copy (old status) into the compared registers, i.e. from the memory pointed to by RDI to RDX:RAX. We also need to set up the swap registers, which will be the data that will be put into the memory location if RDX:RAX=(RSI). We do this by copying the memory pointed to by RDX into RCX:RBX.
In the actual code, we do this in reverse order, starting with setting up the swap and then setting up the compare - this is to not overwrite the rdx register, which originally holds the memory address of the modified object, but after set up will be made to hold the contents of the local copy (old status).

Once everything is set up properly we just call the instruction lock cmpxchg16b on the RSI memory location and the swap is performed if the compared values are the same.

After that we just set the boolean result in RAX depending on the Z flag and clean up after ourselves, returning to the C code.

Part II

Tests

1 Testing The Assembly

To test that our assembly code works as we believe it does, we've written an extensive test program that uses the function in a variety of different swapping conditions, printing the output along the way to make potential bug hunting easier, and finally doing a bunch of point checks on the results to see that they are all as we expect. The testing program is as follows:

```
#include <stdio.h>

#define true = 1
#define false = 0

typedef short bool;

typedef struct test_st {
    long a;
    long b;
} test_t;

extern int my_cmpr_swap(test_t *old, test_t *cur, test_t *mod);

void main(){
    test_t t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;

    t1.a=1;
    t1.b=2;

    t2.a=1;
    t2.b=2;

    t3.a=100;
    t3.b=200;

    t4.a = 1;
    t4.b = 2;

    t5.a = 1;
    t5.b = 2;

    t6.a = 100;
    t6.b = 200;

    t7.a = 1;
    t7.b = 2;

    t8.a = 5;
    t8.b = 10;

    t9.a = 100;
    t9.b = 200;

    t10.a = 50;
    t10.b = 250;

    t11.a = 50;
    t11.b = 250;

    t12.a = 912;
    t12.b = 42;

    bool result[4] = {my_cmpr_swap(&t1, &t2, &t3), my_cmpr_swap(&t6,&t5,&t4), my_cmpr_swap(&t7, t8  t9   my_cm

    char* resultString[4];
    //Assembling the string for true and false
    for (int i = 0; i < 4; i++)     {
```

```
        if(result[i]==1){
            resultString[i] = "true";
            continue;
        }
        else if (result[i] == 0){
            resultString[i] = "false";
            continue;
        }
            else resultString[i] = "error!";
    }

    printf("The function returned %s.\n", resultString[0]);
    printf("t1 is: (%ld,%ld)\n", t1.a,t1.b);
    printf("t2 is: (%ld,%ld)\n", t2.a,t2.b);
    printf("t3 is: (%ld,%ld)\n \n", t3.a,t3.b);
    printf("Changing the order, to compare t3 with t2 and swap with t1 \n (Using clones (t4,t5,
    printf("t4 is: (%ld,%ld)\n", t4.a, t4.b);
    printf("t5 is: (%ld,%ld)\n", t5.a, t5.b);
    printf("t6 is: (%ld,%ld)\n \n", t6.a, t6.b);
    printf("And just to make sure the false run before didn't swap the two identical structs, w
    printf("t7 is: (%ld,%ld)\n", t7.a, t7.b);
    printf("t8 is: (%ld,%ld)\n", t8.a, t8.b);
    printf("t9 is: (%ld,%ld)\n \n", t9.a, t9.b);
    printf("Finally we perform one last true swap to ensure the swapping order is correct \n We
    printf("t10 is: (%ld,%ld)\n", t10.a, t10.b);
    printf("t11 is: (%ld,%ld)\n", t11.a, t11.b);
    printf("t12 is: (%ld,%ld)\n \n", t12.a, t12.b);

    //Few crash tests for a quick overview
    if (
        resultString[0] == "true" &&
        resultString[1] == "false" &&
        resultString[2] == "false" &&
        resultString[3] == "true" &&
        t2.a == t3.a && t2.b == t3.b &&
        t5.a != t6.a && t5.b != t6.b &&
        t7.a != t8.a && t7.b != t8.b &&
        t6.a != t8.a && t6.b != t8.b &&
        t12.a == t10.a && t12.b == t10.b
    ) {
        printf("Yay! It seems to be working! \n \n");
    } else {
        printf("Think you better check your code \n \n");
    }
}
```

Running the test program, we get the output:

```
[Caspers-MacBook-Pro:~ casper$ ./test-extended
The function returned true.
t1 is: (1,2)
t2 is: (100,200)
t3 is: (100,200)

Changing the order, to compare t3 with t2 and swap with t1
 (Using clones (t4,t5,t6) to not change results from before)
 We get result: false
t4 is: (1,2)
t5 is: (1,2)
t6 is: (100,200)

And just to make sure the false run before didn't swap the two identical structs, we try with all different inputs on a false
 We get result: false
t7 is: (1,2)
t8 is: (5,10)
t9 is: (100,200)

Finally we perform one last true swap to ensure the swapping order is correct
 We get result: true
t10 is: (912,42)
t11 is: (50,250)
t12 is: (912,42)

Yay! It seems to be working!
```

Based on this, we believe to have a good understanding of the assembly code, and feel confident it works as we expect.

2 Testing The Main Program

A key part of this task is to avoid the threads overwriting each others' work. To verify that the program works as we expect we run it with various numbers of threads and see if we get consistent and correct output. We also time the code, to see how our locking mechanism affects the performance scaling with the number of threads.

All of the following is performed on an Intel Core i7 4770HQ with 4 physical and 8 logical cores

```
Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 1
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real    0m16.723s
user    0m16.701s
sys     0m0.011s
Caspers-MacBook-Pro:latestWork casper$
```

```
Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 2
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real    0m10.540s
user    0m16.911s
sys     0m0.011s
Caspers-MacBook-Pro:latestWork casper$
```

```
Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 3
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real    0m7.745s
user    0m17.608s
sys     0m0.076s
Caspers-MacBook-Pro:latestWork casper$
```

```
Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 4
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real    0m5.889s
user    0m17.702s
sys     0m0.027s
Caspers-MacBook-Pro:latestWork casper$
```

```
[Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 5
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real	0m5.181s
user	0m19.704s
sys	0m0.026s
Caspers-MacBook-Pro:latestWork casper$
```

```
[Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 6
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real	0m4.695s
user	0m21.254s
sys	0m0.024s
Caspers-MacBook-Pro:latestWork casper$
```

```
[Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 7
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real	0m4.285s
user	0m22.596s
sys	0m0.099s
Caspers-MacBook-Pro:latestWork casper$
```

```
[Caspers-MacBook-Pro:latestWork casper$ time ./MacBinary 8
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.
real	0m4.042s
user	0m24.150s
sys	0m0.045s
Caspers-MacBook-Pro:latestWork casper$
```

As we can see, we get consistent results for all the executions, no matter the number of threads, signifying that the program functions correctly. We have also validated that the number of primes we've computed is correct, though it should be noted that we aren't counting 2 and 5 since they only occur once, so the real number of primes is 2 higher than our program says as a total in all cases, but this is intentional behaviour. As expected, the distribution of primes ending in digits 1, 3, 7 or 9 is also fairly uniform.
The performance scaling with the number of threads is nearly as good as in our last hand-in where we had no need for a locking mechanism. In fact, going from 1 to 2 threads still yields $\approx$ 59% speed increase, which is almost identical to the last hand-in. Though going from 2 to 4 yields a speedup of "only" $\approx$ 79%, where it was $\approx$ 81% in the last hand-in; Though still a relatively minor difference. Thus we can conclude that the threads do not spend a significant amount oftime waiting on a swap.

Finally, we'll just run some quick tests with different MAX values, to verify we also get the correct number of primes for other boundaries.

```
[Caspers-MacBook-Pro:~ casper$ ./cmp_swp_template 8
The result was 78496 primes numbers ending with 1,3,7 or 9 in the range of [0,999999].
Primes numbers ending with 1: 19617
.Primes numbers ending with 3: 19665
.Primes numbers ending with 7: 19621
.Primes numbers ending with 9: 19593
.Caspers-MacBook-Pro:~ casper$
```

```
[Caspers-MacBook-Pro:~ casper$ ./cmp_swp_template 8
The result was 2 primes numbers ending with 1,3,7 or 9 in the range of [0,9].
Primes numbers ending with 1: 0
.Primes numbers ending with 3: 1
.Primes numbers ending with 7: 1
.Primes numbers ending with 9: 0
.Caspers-MacBook-Pro:~ casper$ |
```

```
[Caspers-MacBook-Pro:~ casper$ ./cmp_swp_template 8
The result was 5761453 primes numbers ending with 1,3,7 or 9 in the range of [0,99999999].
Primes numbers ending with 1: 1440298
.Primes numbers ending with 3: 1440474
.Primes numbers ending with 7: 1440495
.Primes numbers ending with 9: 1440186
.Caspers-MacBook-Pro:~ casper$ |
```

This should be compared against the official results here:

Table 1. Values of $\pi(x)$

n	x	$\pi(x)$	ref
1	10	4	
2	100	25	
3	1,000	168	
4	10,000	1,229	
5	100,000	9,592	
6	1,000,000	78,498	
7	10,000,000	664,579	
8	100,000,000	5,761,455	

Thoughh again, our program does not count 2 and 5 so we are two below the official results in all cases.
Aside from that however, we can see that our program produces the correct results in all the tests.

3 Checking For Memory Leaks

Last but not least, we'll use Valgrind to check that we don't have any memory leaks.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

root@x86:/this$ valgrind /home/Casper/latestWork/LinuxBinary 128
==62264== Memcheck, a memory error detector
==62264== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==62264== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==62264== Command: /home/Casper/latestWork/LinuxBinary 128
==62264==
The result was 664577 primes numbers ending with 1,3,7 or 9 in the range of [0,9999999].
Primes numbers ending with 1: 166104
.Primes numbers ending with 3: 166230
.Primes numbers ending with 7: 166211
.Primes numbers ending with 9: 166032
.==62264==
==62264== HEAP SUMMARY:
==62264==     in use at exit: 32 bytes in 2 blocks
==62264==   total heap usage: 645 allocs, 643 frees, 46,112 bytes allocated
==62264==
==62264== LEAK SUMMARY:
==62264==    definitely lost: 0 bytes in 0 blocks
==62264==    indirectly lost: 0 bytes in 0 blocks
==62264==      possibly lost: 0 bytes in 0 blocks
==62264==    still reachable: 32 bytes in 2 blocks
==62264==         suppressed: 0 bytes in 0 blocks
==62264== Rerun with --leak-check=full to see details of leaked memory
==62264==
==62264== For counts of detected and suppressed errors, rerun with: -v
==62264== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@x86:/this$ PS1=Yay!
Yay!
```

Looks like we have no un-accounted for memory and that all is good.