

Contents

Part A: Memory leaks and tools to find them	2
Questions:	2
Programs:	3
First: Program with leak.....	3
First: Result having the memory leak:	4
Second: Program with fixed memory leak:.....	5
Second: Result having fixed memory leak.....	6
Third: Last program with even worse memory usage:.....	7
Third: Result of third program.	8
Part B: Lottery Scheduling from the book	9
First part.....	9
Second part:	14

Part A: Memory leaks and tools to find them

Questions:

1. Explain what sudo did in either case and why we needed it?
 - a. Sudo gives us a higher level access to the operations. In our case we wanted to install valgrind so we needed that access level.
2. Write a program in C / C++ that allocates memory using malloc() but forgets to free it before exiting. What happens when this program runs? How about valgrind (with the command: valgrind --leak-check=yes MYPROGRAM)?
 - a. First Program:
 - i. We left one memory leak and we printed the allocation on of that memory
 - ii. As we ran valgrind we noticed that we had 4 definitely lost leaks.
3. Create other test cases for valgrind. Explain why you choose them and the expected results. In other words, change your code, recompile and do step two again.
 - a. Second Program
 - i. Without leaks to compare valgrind results.
 - ii. We freed the memory and we saw that there is not a memory leak as expected.
 - b. Third Program
 - i. We made a while infinite loop that allocates memory without freeing. This we had to stop the running of the program (control + c) since it was infinite and could result in damage.
 - ii. We saw 52 leaks in 13 blocs that could have been more if we let the program even more time. As expected, the number of leaks increased.

Programs:

First: Program with leak

```
GNU nano 6.2                               myprog3.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Allocate memory for an integer
    int *ptr = (int *)malloc(sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit with an error code
    }

    // Store a value in the allocated memory
    *ptr = 42;

    // Print the value
    printf("The value in the allocated memory is: %d\n", *ptr);

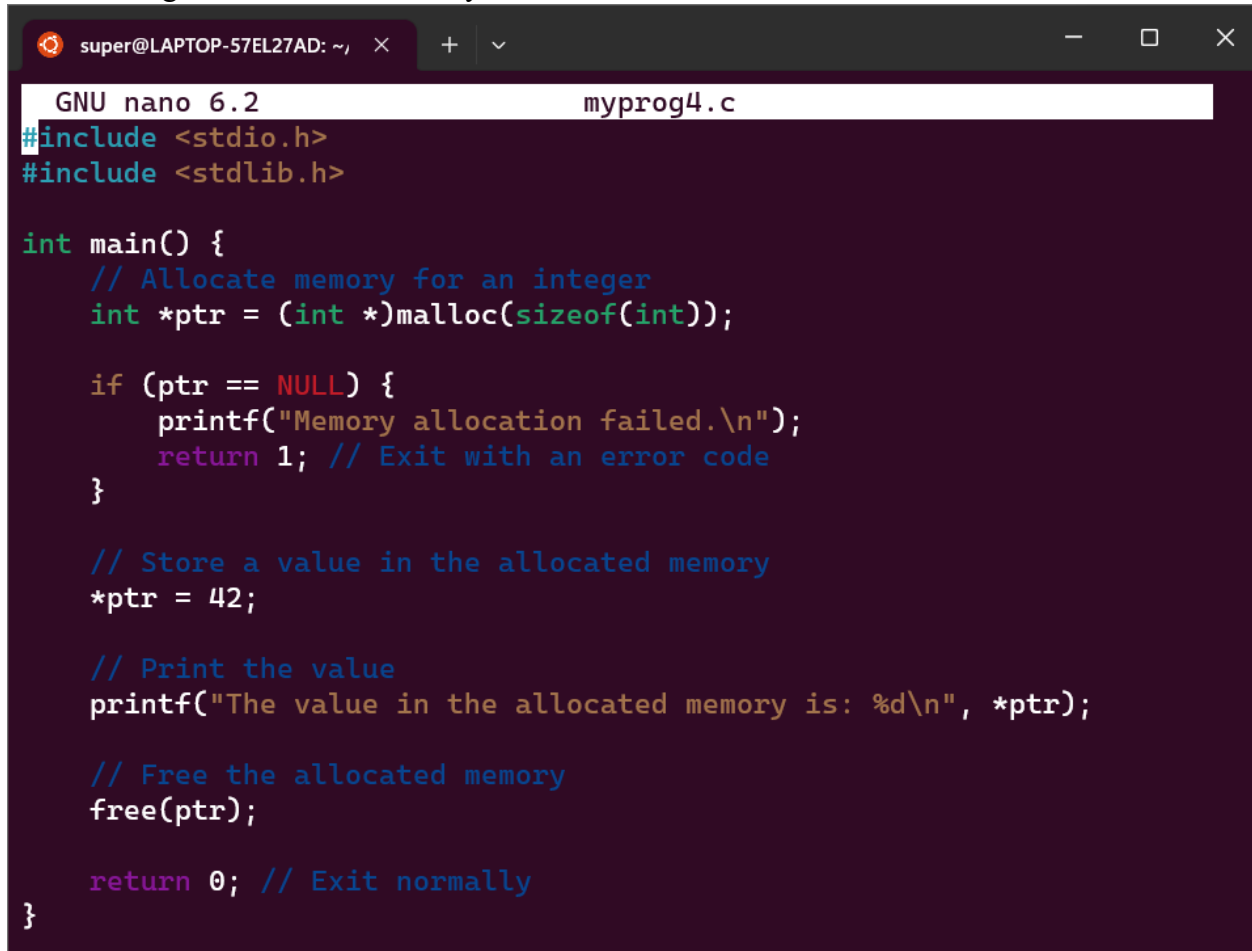
    // Program exits without freeing the allocated memory

    return 0; // Exit normally
}
```

First: Result having the memory leak:

```
super@LAPTOP-57EL27AD: ~/super/HW3$ ./myprog3
The value in the allocated memory is: 42
super@LAPTOP-57EL27AD: ~/super/HW3$ valgrind --leak-check=yes ./myprog3
==14986== Memcheck, a memory error detector
==14986== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14986== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14986== Command: ./myprog3
==14986==
The value in the allocated memory is: 42
==14986==
==14986== HEAP SUMMARY:
==14986==      in use at exit: 4 bytes in 1 blocks
==14986==    total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==14986==
==14986== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14986==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==14986==    by 0x10919E: main (in /home/super/super/HW3/myprog3)
==14986==
==14986== LEAK SUMMARY:
==14986==    definitely lost: 4 bytes in 1 blocks
==14986==    indirectly lost: 0 bytes in 0 blocks
==14986==    possibly lost: 0 bytes in 0 blocks
==14986==    still reachable: 0 bytes in 0 blocks
==14986==    suppressed: 0 bytes in 0 blocks
==14986==
==14986== For lists of detected and suppressed errors, rerun with: -s
==14986== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
super@LAPTOP-57EL27AD: ~/super/HW3$
```

Second: Program with fixed memory leak:



```
GNU nano 6.2 myprog4.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Allocate memory for an integer
    int *ptr = (int *)malloc(sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit with an error code
    }

    // Store a value in the allocated memory
    *ptr = 42;

    // Print the value
    printf("The value in the allocated memory is: %d\n", *ptr);

    // Free the allocated memory
    free(ptr);

    return 0; // Exit normally
}
```

Second: Result having fixed memory leak

```
super@LAPTOP-57EL27AD:~/super/HW3$ nano myprog3.c
super@LAPTOP-57EL27AD:~/super/HW3$ touch myprog4.c
super@LAPTOP-57EL27AD:~/super/HW3$ nano myprog4.c
super@LAPTOP-57EL27AD:~/super/HW3$ gcc -o myprog4 myprog4.c
super@LAPTOP-57EL27AD:~/super/HW3$ ./myprog4
The value in the allocated memory is: 42
super@LAPTOP-57EL27AD:~/super/HW3$ valgrind --leak-check=yes ./myprog4
==15008== Memcheck, a memory error detector
==15008== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==15008== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyrigh
t info
==15008== Command: ./myprog4
==15008==
The value in the allocated memory is: 42
==15008==
==15008== HEAP SUMMARY:
==15008==      in use at exit: 0 bytes in 0 blocks
==15008==    total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==15008==
==15008== All heap blocks were freed -- no leaks are possible
==15008==
==15008== For lists of detected and suppressed errors, rerun with: -s
==15008== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
super@LAPTOP-57EL27AD:~/super/HW3$ █
```

Third: Last program with even worse memory usage.

Note: This program was deleted after being used because it could be problematic to run without necessary caution.

```
GNU nano 6.2                               myprog5.c *
#include <stdio.h>
#include <stdlib.h>

int main() {
    while (1) {
        // Allocate memory for an integer
        int *ptr = (int *)malloc(sizeof(int));

        if (ptr == NULL) {
            printf("Memory allocation failed.\n");
            return 1; // Exit with an error code
        }

        // Store a value in the allocated memory
        *ptr = 42;

        // No memory is ever freed

        // Sleep for a while to make it more obvious
        sleep(1);
    }

    return 0; // This line will never be reached
}
```

Third: Result of third program.

```
super@LAPTOP-57EL27AD: ~/super/HW3$ gcc -o myprog5 myprog5.c
myprog5.c: In function 'main':
myprog5.c:20:9: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   20 |         sleep(1);
      |         ^~~~~~
super@LAPTOP-57EL27AD: ~/super/HW3$ valgrind --leak-check=yes ./myprog5==15024==
Memcheck, a memory error detector
==15024== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15024== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==15024== Command: ./myprog5
==15024==
^C==15024==
==15024== Process terminating with default action of signal 2 (SIGINT)==15024==
==15024==   at 0x494878A: clock_nanosleep@@GLIBC_2.17 (clock_nanosleep.c:78)
==15024==   by 0x494D676: nanosleep (nanosleep.c:25)
==15024==   by 0x494D5AD: sleep (sleep.c:55)
==15024==   by 0x1091D8: main (in /home/super/super/HW3/myprog5)
==15024==
==15024== HEAP SUMMARY:
==15024==   in use at exit: 56 bytes in 14 blocks
==15024==   total heap usage: 14 allocs, 0 frees, 56 bytes allocated
==15024==
==15024== 52 bytes in 13 blocks are definitely lost in loss record 2 of 2
==15024==   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==15024==   by 0x10919E: main (in /home/super/super/HW3/myprog5)
==15024==
==15024== LEAK SUMMARY:
==15024==   definitely lost: 52 bytes in 13 blocks
==15024==   indirectly lost: 0 bytes in 0 blocks
==15024==   possibly lost: 0 bytes in 0 blocks
==15024==   still reachable: 4 bytes in 1 blocks
==15024==   suppressed: 0 bytes in 0 blocks
==15024== Reachable blocks (those to which a pointer was found) are not shown.
==15024== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==15024==
```


Part B: Lottery Scheduling from the book

First part

1. Compute the solutions for simulations with 3 jobs and random seeds of 1, 2, and 3.

```
super@LAPTOP-57EL27AD:~/super/HW3/ostep-homework/cpu-sched-lottery$ ./lottery.py -j 3 -s 1
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

Here is the job list, with the run time of each job:
Job 0 ( length = 1, tickets = 84 )
Job 1 ( length = 7, tickets = 25 )
Job 2 ( length = 4, tickets = 44 )

Here is the set of random numbers you will need (at most):
Random 651593
Random 788724
Random 93859
Random 28347
Random 835765
Random 432767
Random 762280
Random 2106
Random 445387
Random 721540
Random 228762
Random 945271
```

```
super@LAPTOP-57EL27AD:~/super/HW3/ostep-homework/cpu-sched-lottery$ ./lottery.py -j 3 -s 2
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 2

Here is the job list, with the run time of each job:
  Job 0 ( length = 9, tickets = 94 )
  Job 1 ( length = 8, tickets = 73 )
  Job 2 ( length = 6, tickets = 30 )

Here is the set of random numbers you will need (at most):
Random 605944
Random 606802
Random 581204
Random 158383
Random 430670
Random 393532
Random 723012
Random 994820
Random 949396
Random 544177
Random 444854
Random 268241
Random 35924
Random 27444
Random 464894
Random 318465
Random 380015
Random 891790
Random 525753
Random 560510
Random 236123
Random 23858
```

```
super@LAPTOP-57EL27AD:~/super/HW3/ostep-homework/cpu-sched-lottery$ ./lottery.py -j 3 -s 3
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 3

Here is the job list, with the run time of each job:
  Job 0 ( length = 2, tickets = 54 )
  Job 1 ( length = 3, tickets = 60 )
  Job 2 ( length = 6, tickets = 6 )

Here is the set of random numbers you will need (at most):
Random 13168
Random 837469
Random 259354
Random 234331
Random 995645
Random 470263
Random 836462
Random 476353
Random 639068
Random 150616
Random 634861
```

2. Now run with two specific jobs: each of length 10, but one (job 0) with just 1 ticket and the other (job 1) with 100 (e.g., -l 10:1,10:100). What happens when the number of tickets is so imbalanced? Will job 0 ever run before job 1 completes? How often? In general, what does such a ticket imbalance do to the behavior of lottery scheduling?
 - a. I think job 0 would barely get a chance to run since the scheduling is mostly used for job 1. This could lead to “unfairness” in case both jobs represent equal importance. Most of the times Job 1 will be completed before job 0 since it is most probably the one that is going to be selected to run.

```
super@LAPTOP-57EL27AD:~/super/HW3/ostep-homework/cpu-sched-lottery$ ./lottery.py -l 10:1,10:100
ARG jlist 10:1,10:100
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 0

Here is the job list, with the run time of each job:
  Job 0 ( length = 10, tickets = 1 )
  Job 1 ( length = 10, tickets = 100 )

Here is the set of random numbers you will need (at most):
Random 844422
Random 757955
Random 420572
Random 258917
Random 511275
Random 404934
Random 783799
Random 303313
Random 476597
Random 583382
Random 908113
Random 504687
Random 281838
Random 755804
Random 618369
Random 250506
Random 909747
Random 982786
Random 810218
Random 902166
```

3. When running with two jobs of length 100 and equal ticket allocations of 100 (-1 100:100,100:100), how unfair is the scheduler? Run with some different random seeds to determine the (probabilistic) answer; let unfairness be determined by how much earlier one job finishes than the other.
 - a. As both jobs have the same length and ticket allocations, we would suppose that the scheduling is entirely fair. Of course, this could change as probabilistic differences related to the seed. So as a coin, if we ran the program just one time, we would conclude that is biased. As we approximate the 'n' number of tries to infinite we would consider fully fair.

4. How does your answer to the previous question change as the quantum size (-q) gets larger?
 - a. As we increase the quantum size the fairness increases as we give different times to reevaluate the scheduling allocations.

Second part:

1. To start things off, let's learn how to use the simulator to study how to build an effective multi-processor scheduler. The first simulation will run just one job, which has a run-time of 30, and a working-set size of 200. Run this job (called job 'a' here) on one simulated CPU as follows: `./multi.py -n 1 -L a:30:200`. How long will it take to complete? Turn on the `-c` flag to see a final answer, and the `-t` flag to see a tick-by-tick trace of the job and how it is scheduled.

```
super@LAPTOP-57EL27AD: ~/super/HW3/ostep-homework/cpu-sched-multi$ ./multi.py n 1 -L a:30:200 -c -t
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 2
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 100
ARG random_order False
ARG trace True
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']

  0  a  -
  1  a  -
  2  a  -
  3  a  -
  4  a  -
  5  a  -
  6  a  -
  7  a  -
  8  a  -
  9  a  -
-----
```

```
-----
10  a  -
11  a  -
12  a  -
13  a  -
14  a  -
15  a  -
16  a  -
17  a  -
18  a  -
19  a  -
-----
20  a  -
21  a  -
22  a  -
23  a  -
24  a  -
25  a  -
26  a  -
27  a  -
28  a  -
29  a  -

Finished time 30

Per-CPU stats
CPU 0  utilization 100.00 [ warm 0.00 ]
CPU 1  utilization 0.00 [ warm 0.00 ]
```

2. Now increase the cache size so as to make the job's working set (size=200) fit into the cache (which, by default, is size=100); for example, run `./multi.py -n 1 -L a:30:200 -M 300`. Can you predict how fast the job will run once it fits in cache? (hint: remember the key parameter of the warm rate, which is set by the `-r` flag) Check your answer by running with the solve flag (`-c`) enabled.
 - a. As I increase the warm rate, I get a time closer to 11. As it was with 1 the time were 50. In this case the cache fits within the memory so we could expect that the answer is mostly the warm ratio mapped.
3. One cool thing about `multi.py` is that you can see more detail about what is going on with different tracing flags. Run the same simulation as above, but this time with time left tracing enabled (`-T`). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?
 - a. We see that the second column is the remaining runtime, and it decreases monotonically one unit each clock tic. Once we get to the amount necessary for warm up it would go down to finishing the program.
4. Now add one more bit of tracing, to show the status of each CPU cache for each job, with the `-C` flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a 'w' (if the cache is warm for that job). At what point does the cache become warm for job 'a' in this simple example? What happens as you change the warmup time parameter (`-w`) to lower or higher values than the default?
 - a. In this last example it became warm in the last tree spaces.

```
Scheduler central queue: ['a']

0  a [ 29] cache[ ]
1  a [ 28] cache[ ]
2  a [ 27] cache[ ]
3  a [ 26] cache[ ]
4  a [ 25] cache[ ]
5  a [ 24] cache[ ]
6  a [ 23] cache[ ]
7  a [ 22] cache[ ]
8  a [ 21] cache[ ]
9  a [ 20] cache[w]
-----
10 a [  5] cache[w]
11 a [  0] cache[w]

Finished time 12

Per-CPU stats
CPU 0 utilization 100.00 [ warm 16.67 ]
```


5. At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn't this a multi-processor CPU scheduling chapter? Oh yeah! So let's start working with multiple jobs. Specifically, let's run the following three jobs on a two-CPU system (i.e., type `./multi.py -n 2 -L a:100:100,b:100:50,c:100:50`) Can you predict how long this will take, given a round-robin centralized scheduler? Use `-c` to see if you were right, and then dive down into details with `-t` to see a step-by-step and then `-C` to see whether caches got warmed effectively for these jobs. What do you notice?
- a. Well, we have 3 jobs with two CPU's. The first job is going to take 100, the second 50 and the third is going to take place in the first CPU so it would make sense it would take somewhere around 100 to 200 + warmup.

```
Finished time 150

Per-CPU stats
CPU 0  utilization 100.00 [ warm 0.00 ]
CPU 1  utilization 100.00 [ warm 0.00 ]
```

i.

6. Now we'll apply some explicit controls to study cache affinity, as described in the chapter. To do this, you'll need the `-A` flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. In this case, let's use it to place jobs 'b' and 'c' on CPU 1, while restricting 'a' to CPU 0. This magic is accomplished by typing this `./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1`; don't forget to turn on various tracing options to see what is really happening! Can you predict how fast this version will run? Why does it do better? Will other combinations of 'a', 'b', and 'c' onto the two processors run faster or slower?
- a. I would think this one would faster since we are giving the large job to one CPU and the other smaller jobs to the other.

```
Finished time 110

Per-CPU stats
CPU 0  utilization 50.00 [ warm 40.91 ]
CPU 1  utilization 100.00 [ warm 81.82 ]
```

i.

- b. As I put the "a" and "c" job to CPU 1 and "b" job in CPU 0 I would expect to take even longer than the last steps and to have very poor performance in CPU utilization.

```
Finished time 200

Per-CPU stats
CPU 0  utilization 27.50 [ warm 22.50 ]
CPU 1  utilization 100.00 [ warm 0.00 ]
```

i.

7. One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, a situation entitled super-linear speedup. To experiment with this, use the job description here (-L a:100:100,b:100:100,c:100:100) with a small cache (-M 50) to create three jobs. Run this on systems with 1, 2, and 3 CPUs (-n 1, -n 2, -n 3). Now, do the same, but with a larger per-CPU cache of size 100. What do you notice about performance as the number of CPUs scales? Use -c to confirm your guesses, and other tracing flags to dive even deeper.

- a. I would think that as we increase the CPU number, the Finished time would take less and less. As we increase the cache the time for one and two CPU's remains constant, but when we get to 3 CPU's the time drops to 55.

- b. Cache 50:

- i. 1 CPU

1. Finished time 300

- ii. 2 CPU

1. Finished time 150

- iii. 3 CPU

1. Finished time 100

- c. Cache 100:

- i. 1 CPU

1. Finished time 300

- ii. 2 CPU

1. Finished time 150

- iii. 3 CPU

1. Finished time 55

8. One other aspect of the simulator worth studying is the per-CPU scheduling option, the -p flag. Run with two CPUs again, and this three job configuration (-L a:100:100,b:100:50,c:100:50). How does this option do, as opposed to the hand-controlled affinity limits you put in place above? How does performance change as you alter the 'peek interval' (-P) to lower or higher values? How does this per-CPU approach work as the number of CPUs scales?

- a. Without changing -P we get 150 Finished time with 100 utilization in both CPU's.
b. As we change -P the result does not affect.
c. If we change CPU's to three and change -P's from 1 to ten or all the way up to 1000 the time remains the same.
d. If we did for CPU's the time is going to stay the same as with 3 but we would have one of them completely unutilized.