Chapter 10

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.

Job name:a run_time:30 working_set_size:200
Scheduler central queue: ['a']
Finished time 30

The finished time is 30. Because the default quantum is 10 and there is only one job in the CPU. The run-time of the job is 30. Also, because the CPU's catch is 100 which is smaller than the job's work set size 200, it means the CPU doesn't have chance to warm up and get higher speed. Therefore, the finished time is equal to the run time 30.

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.

Job name:a run_time:30 working_set_size:200
Scheduler central queue: ['a']
Finished time 20

The finished time is 20. Because we add -M 300 flag which means the CPU cathe size is 300. The first 10 run time can warm up the CPU. Then the CPU can run two time faster by the default warm up rate. We can have 10 + 20/2 = 20 units.

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?
Scheduler central queue: ['a']

0 a [29]
1 a [28]
2 a [27]
3 a [26]
4 a [25]

```
  5   a [ 24]
  6   a [ 23]
  7   a [ 22]
  8   a [ 21]
  9   a [ 20]
----------
 10   a [ 18]
 11   a [ 16]
 12   a [ 14]
 13   a [ 12]
 14   a [ 10]
 15   a [  8]
 16   a [  6]
 17   a [  4]
 18   a [  2]
 19   a [  0]
```

In the first 10 time ticks, the process run 1 per time tick. In the second 10 time ticks, the process run 2 per time tick.

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?
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 [ 18] cache[w]
 11   a [ 16] cache[w]
 12   a [ 14] cache[w]
 13   a [ 12] cache[w]
 14   a [ 10] cache[w]
 15   a [  8] cache[w]
```

```
  16   a [  6] cache[w]
  17   a [  4] cache[w]
  18   a [  2] cache[w]
  19   a [  0] cache[w]
```

In the 10th time tick, cache changes to warm.
If I change -w as 7, the cache would change to warm earlier which is in the 7th time tick.
If I change -w as 12, the cache would change to warm later which is in the 12th time tick.

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

The finished time is (100 + 100 + 100)/2 = 300. Because we have 2 CPUs and the quantum is 10 which means each CPU never have chance to warm up and get faster speed to run the next process.

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?

The finished time is 110. Because process 'a' is on CPU0, and it finished at 55th time tick (10 + 90/2) = 55). On the CPU1, b and c can get speed up after 10 time units for each. Thus the run time is 20 + (90/2 * 2) = 110. Therefore, the overall finished time is 110. Compared with other combinations like ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:0, or ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:0,c:1, the combination of this question is faster. Because for other combinations, the total work set size is larger than the CPU cache size, which means the warm speed up can not work.

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.

./multi.py -n 1 -L a:100:100,b:100:100,c:100:100 -M 50 -T -C -c
Finished time 300
Because there is only 1 CPU, time units = 100 * 3 = 300.

./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -M 50 -T -C -c
Finished time 150
Because there are 2 CPUs and cache size is smaller than 150 which means it can not be warmed to speed up, time units = 100 * 3 /2 = 150.

./multi.py -n 3 -L a:100:100,b:100:100,c:100:100 -M 50 -T -C -c
Finished time 100
Because there are 3 CPUs and cache size is smaller than 100 which means it can not be warmed to speed up, time units = 100 * 3 /3 = 100.

./multi.py -n 1 -L a:100:100,b:100:100,c:100:100 -M 100 -T -C -c
Finished time 300
Because there is only 1 CPU. Even though the catch is larger, there are 3 jobs to run round robin in the CPU. They have to switch to the next after 10 time ticks, which means it can not be warmed to speed up. The time units = 100 * 3 = 300.

./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -M 100 -T -C -c
Finished time 150
Because there are 2 CPUs. Even though the catch is larger, there are 3 jobs to run round robin in the CPU. They have to switch to the next after 10 time ticks, which means it can not be warmed to speed up. time units = 100 * 3 /2 = 150.

./multi.py -n 3 -L a:100:100,b:100:100,c:100:100 -M 100 -T -C -c
Finished time 55
Because there are 3 CPUs. We can run one job on each CPU. The time units = 10 + 90/2 = 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?

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -c -t -T -C
Finished time is 100
At the beginning, a and c run on CPU0, b runs on CPU1. For b, 10 + 90/2 = 55, it can done in 55th time unit. Then in the next quantum, which is the 61th time unit, CPU1 is idle and it peeks CPU0 and grab process a to run. Then a left 70 to run, 10 + 60/2 = 40. For CPU1, the finished time is 55(b) + 5(idle) + 40(a) = 100. For CPU0, the finished time is 30 (a) + 30(c) + 70/2(c) = 95. Theus, total finished time 100.

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -P 0 -c -t -T -C
When -P is 0, we turn the peek mode off. Finished time is 300/2 = 150.

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -P 5 -c -t -T -C
Finshed time is 90. Because the idle time is gone. Once b is done, CPU1 would grab a to run in the next round robin switch.

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -P 10 -c -t -T -C
Finshed time is 100. Same situation with the 1st one.

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -P 20 -c -t -T -C
Finshed time is 100. Same situation with the 1st one.

./multi.py -n 3 -L a:100:100,b:100:50,c:100:50 -p -P 5 -c -t -T -C
Finished time is 55.

./multi.py -n 3 -L a:100:100,b:100:50,c:100:50 -p -P 10 -c -t -T -C
Finished time is 55.

9. Finally, feel free to just generate random workloads and see if you can predict their performance on different numbers of processors, cache sizes, and scheduling options. If you do this, you'll soon be a multi-processor scheduling master, which is a pretty awesome thing to be. Good luck!

./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -p -c -t -T -C -P 40
Finshed time is 115.

./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -p -c -t -M 200 -T -C -P 40
Finshed time is 100.

./multi.py -n 4 -L a:100:100,b:100:100,c:100:100 -p -c -t -M 200 -T -C -P 40
Finshed time is 60.

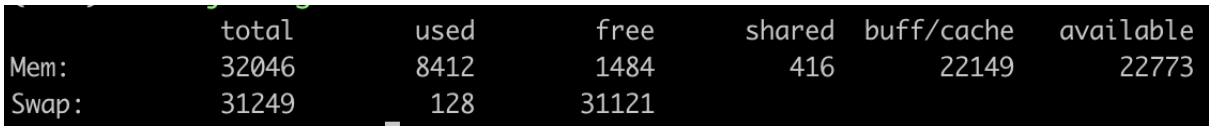
./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -p -c -t -M 200 -T -C -P 5
Finshed time is 90.

Larger cache, shorter finish time. More CPUs, shorter finish time. More often peek, shorter finish time.

Chapter 13

1. The first Linux tool you should check out is the very simple tool free. First, type man free and read its entire manual page; it's short, don't worry!

2. Now, run free, perhaps using some of the arguments that might be useful (e.g., -m, to display memory totals in megabytes). How much memory is in your system? How much is free? Do these numbers match your intuition?



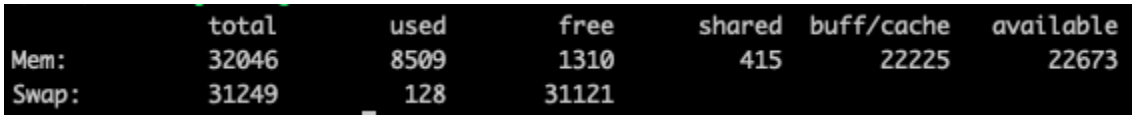
	total	used	free	shared	buff/cache	available
Mem:	32046	8412	1484	416	22149	22773
Swap:	31249	128	31121			

The memory size is 32046 MB. The free memory size is 1484 MB.

3. Next, create a little program that uses a certain amount of memory, called memory-user.c. This program should take one commandline argument: the number of megabytes of memory it will use. When run, it should allocate an array, and constantly streamthrough the array, touching each entry. The program should do this indefinitely, or, perhaps, for a certain amount of time also specified at the command line.

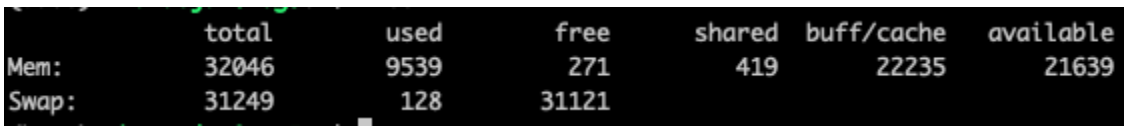
4. Now, while running your memory-user program, also (in a different terminal window, but on the same machine) run the free tool. How do the memory usage totals change when your program is running? How about when you kill the memory-user program? Do the numbers match your expectations? Try this for different amounts of memory usage. What happens when you use really large amounts of memory?

Before run:



	total	used	free	shared	buff/cache	available
Mem:	32046	8509	1310	415	22225	22673
Swap:	31249	128	31121			

After run ./a.out 1024 60:



	total	used	free	shared	buff/cache	available
Mem:	32046	9539	271	419	22235	21639
Swap:	31249	128	31121			

Kill:

```
               total        used        free      shared  buff/cache   available
Mem:           32046        8719        1310         415       22016       22463
Swap:          31249         129       31120
```

With larger amount of usage ./a.out 1200 60:

```
               total        used        free      shared  buff/cache   available
Mem:           32046        9942         260         419       21843       21236
Swap:          31249         129       31120
```

With smaller amount of usage ./a.out 512 60:

```
               total        used        free      shared  buff/cache   available
Mem:           32046        9260         931         419       21854       21918
Swap:          31249         129       31120
```

5. Let’s try onemore tool, known as pmap. Spend some time, and read the pmap manual page in detail.

6. To use pmap, you have to know the process IDof the process you’re interested in. Thus, first run ps auxw to see a list of all processes; then, pick an interesting one, such as a browser. You can also use your memory-user program in this case (indeed, you can even have that program call getpid() and print out its PID for your convenience).

```
29741:   ./a.out 1000 60
000055d89725e000      4K r-x-- a.out
000055d89745e000      4K r---- a.out
000055d89745f000      4K rw--- a.out
000055d899346000    132K rw---   [ anon ]
00007f99e2725000 1024004K rw---   [ anon ]
00007f9a20f26000   1948K r-x-- libc-2.27.so
00007f9a2110d000   2048K ----- libc-2.27.so
00007f9a2130d000     16K r---- libc-2.27.so
00007f9a21311000      8K rw--- libc-2.27.so
00007f9a21313000     16K rw---   [ anon ]
00007f9a21317000    164K r-x-- ld-2.27.so
00007f9a2151e000      8K rw---   [ anon ]
00007f9a21540000      4K r---- ld-2.27.so
00007f9a21541000      4K rw--- ld-2.27.so
00007f9a21542000      4K rw---   [ anon ]
00007fff6f7d2000    132K rw---   [ stack ]
00007fff6f7fc000     12K r----   [ anon ]
00007fff6f7ff000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total          1028520K
```

7. Now run pmap on some of these processes, using various flags (like -X) to reveal many details about the process. What do you see? How many different entities make up a modern address space, as opposed to our simple conception of code/stack/heap?

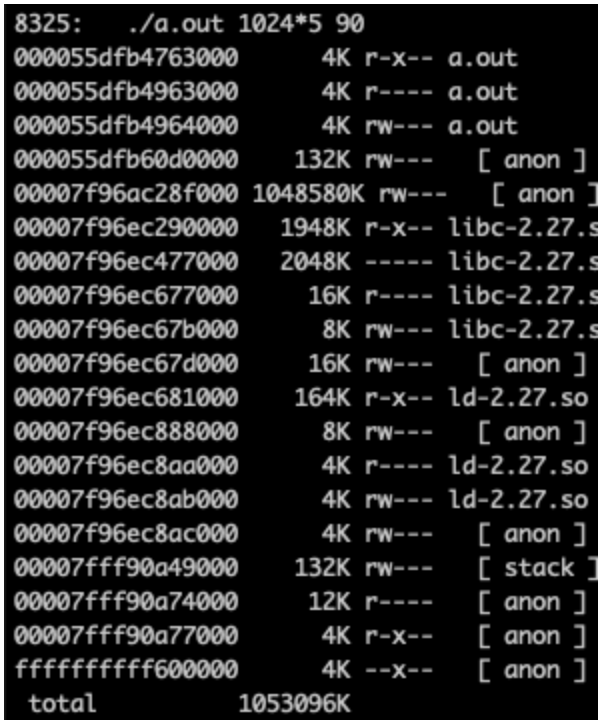
```
      Address Perm   Offset Device   Inode    Size     Rss     Pss Referenced Anonymous LazyFree ShmemPmdMapped FilePmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked THPeligible Mapping
   5615566f5000 r-xp 00000000 08:03 55456292       4       4       4          4         0        0             0             0              0              0    0       0      0           0 a.out
   5615568f5000 r--p 00000000 08:03 55456292       4       4       4          4         4        0             0             0              0              0    0       0      0           0 a.out
   5615568f6000 rw-p 00001000 08:03 55456292       4       4       4          4         4        0             0             0              0              0    0       0      0           0 a.out
   561556d31000 rw-p 00000000 00:00        0     132       4       4          4         4        0             0             0              0              0    0       0      0           0 [heap]
   7f8fc74b2000 rw-p 00000000 00:00        0 1024004 1024004 1024004    1024004   1024004        0             0             0              0              0    0       0      0           0
   7f9005cb3000 r-xp 00000000 08:03 12716291    1948    1332       5       1332         0        0             0             0              0              0    0       0      0           0 libc-2.27.so
   7f9005e9a000 ---p 001e7000 08:03 12716291    2048       0       0          0         0        0             0             0              0              0    0       0      0           0 libc-2.27.so
   7f900609a000 r--p 001e7000 08:03 12716291      16      16      16         16        16        0             0             0              0              0    0       0      0           0 libc-2.27.so
   7f900609e000 rw-p 001eb000 08:03 12716291       8       8       8          8         8        0             0             0              0              0    0       0      0           0 libc-2.27.so
   7f90060a0000 rw-p 00000000 00:00        0      16      12      12         12        12        0             0             0              0              0    0       0      0           0
   7f90060a4000 r-xp 00000000 08:03 12716284     164     164       0        164         0        0             0             0              0              0    0       0      0           0 ld-2.27.so
   7f90062ab000 rw-p 00000000 00:00        0       8       8       8          8         8        0             0             0              0              0    0       0      0           0
   7f90062cd000 r--p 00029000 08:03 12716284       4       4       4          4         4        0             0             0              0              0    0       0      0           0 ld-2.27.so
   7f90062ce000 rw-p 0002a000 08:03 12716284       4       4       4          4         4        0             0             0              0              0    0       0      0           0 ld-2.27.so
   7f90062cf000 rw-p 00000000 00:00        0       4       4       4          4         4        0             0             0              0              0    0       0      0           0
   7ffd95b2b000 rw-p 00000000 00:00        0     132      12      12         12        12        0             0             0              0              0    0       0      0           0 [stack]
   7ffd95bc5000 r--p 00000000 00:00        0      12       0       0          0         0        0             0             0              0              0    0       0      0           0 [vvar]
   7ffd95bc8000 r-xp 00000000 00:00        0       4       4       0          4         0        0             0             0              0              0    0       0      0           0 [vdso]
ffffffffff600000 --xp 00000000 00:00        0       4       0       0          0         0        0             0             0              0              0    0       0      0           0 [vsyscall]
                                                ====== ======= ======= ========== ========= ======== ============= ============= ============== ============== ==== ======= ====== ===========
                                             1028520 1025588 1024093    1025588   1024084        0             0             0              0              0    0       0      0           0 KB
```

There are .so and vvar, vdso, vsyscall besides code, stack and heap.

8. Finally, let’s run pmap on your memory-user program, with different amounts of used memory. What do you see here? Does the output from pmap match your expectations?

```
7243:   ./a.out 1024 90
000055d2aa416000      4K r-x-- a.out
000055d2aa616000      4K r---- a.out
000055d2aa617000      4K rw--- a.out
000055d2aaab6000    132K rw---   [ anon ]
00007f75bcadb000 1048580K rw---   [ anon ]
00007f75fcadc000   1948K r-x-- libc-2.27.so
00007f75fccc3000   2048K ----- libc-2.27.so
00007f75fcec3000     16K r---- libc-2.27.so
00007f75fcec7000      8K rw--- libc-2.27.so
00007f75fcec9000     16K rw---   [ anon ]
00007f75fcecd000    164K r-x-- ld-2.27.so
00007f75fd0d4000      8K rw---   [ anon ]
00007f75fd0f6000      4K r---- ld-2.27.so
00007f75fd0f7000      4K rw--- ld-2.27.so
00007f75fd0f8000      4K rw---   [ anon ]
00007fffc74c5000    132K rw---   [ stack ]
00007fffc75fb000     12K r----   [ anon ]
00007fffc75fe000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total          1053096K
```

```
8993:   ./a.out 512 60
000055b0af1f5000      4K r-x-- a.out
000055b0af3f5000      4K r---- a.out
000055b0af3f6000      4K rw--- a.out
000055b0b0918000    132K rw---   [ anon ]
00007f091ffc0000 524292K rw---   [ anon ]
00007f093ffc1000   1948K r-x-- libc-2.27.so
00007f09401a8000   2048K ----- libc-2.27.so
00007f09403a8000     16K r---- libc-2.27.so
00007f09403ac000      8K rw--- libc-2.27.so
00007f09403ae000     16K rw---   [ anon ]
00007f09403b2000    164K r-x-- ld-2.27.so
00007f09405b9000      8K rw---   [ anon ]
00007f09405db000      4K r---- ld-2.27.so
00007f09405dc000      4K rw--- ld-2.27.so
00007f09405dd000      4K rw---   [ anon ]
00007ffc51128000    132K rw---   [ stack ]
00007ffc5118d000     12K r----   [ anon ]
00007ffc51190000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total           528808K
```



For anon, the value is different.

Chapter 14

1 First, write a simple program called null.c that creates a pointer to an integer, sets it to NULL, and then tries to dereference it. Com- pile this into an executable called null. What happens when you run this program?

gcc null.c -o null

./null

```
Segmentation fault (core dumped)
```

2 Next, compile this program with symbol information included(with the -g flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the de- bugger by typing gdb null and then, once gdb is running, typing run. What does gdb show you?

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555555546fc in main ()
```

3 Finally, use the valgrind tool on this program. We'll use the memcheck tool that is a part of valgrind to analyze what happens. Run this by typing in the following: valgrind --leak-check=yes null. What happens when you run this? Can you interpret the output from the tool?

valgrind –leak-check=yes ./null

```
==16036== Memcheck, a memory error detector
==16036== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16036== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==16036== Command: ./null
==16036==
==16036== Invalid read of size 4
==16036==    at 0x1086FC: main (in /home/zhuzuojun/learn/ostep/ch14/null)
==16036==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==16036==
==16036==
==16036== Process terminating with default action of signal 11 (SIGSEGV)
==16036==  Access not within mapped region at address 0x0
==16036==    at 0x1086FC: main (in /home/zhuzuojun/learn/ostep/ch14/null)
==16036==  If you believe this happened as a result of a stack
==16036==  overflow in your program's main thread (unlikely but
==16036==  possible), you can try to increase the size of the
==16036==  main thread stack using the --main-stacksize= flag.
==16036==  The main thread stack size used in this run was 8388608.
==16036==
==16036== HEAP SUMMARY:
==16036==     in use at exit: 4 bytes in 1 blocks
==16036==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==16036==
==16036== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16036==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16036==    by 0x1086EB: main (in /home/zhuzuojun/learn/ostep/ch14/null)
==16036==
==16036== LEAK SUMMARY:
==16036==    definitely lost: 4 bytes in 1 blocks
==16036==    indirectly lost: 0 bytes in 0 blocks
==16036==      possibly lost: 0 bytes in 0 blocks
==16036==    still reachable: 0 bytes in 0 blocks
==16036==         suppressed: 0 bytes in 0 blocks
==16036==
==16036== For counts of detected and suppressed errors, rerun with: -v
==16036== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

There is a memory leak in this program. The program crashes when trying to dereference pointer p. Free() does not run. Then 4 bytes in 1 block are lost.

4 Write a simple program that allocates memory using malloc() but forgets to free it before exiting. What happens when this program runs? Can you use gdb to find any problems with it? How about valgrind (again with the --leak-check=yes flag)?

```
[Inferior 1 (process 23494) exited normally]
```

gdb does not find anything wrong.

```
==23765== Memcheck, a memory error detector
==23765== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==23765== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==23765== Command: ./no-free
==23765==
==23765==
==23765== HEAP SUMMARY:
==23765==     in use at exit: 4 bytes in 1 blocks
==23765==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==23765==
==23765== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23765==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23765==    by 0x10865B: main (no-free.c:4)
==23765==
==23765== LEAK SUMMARY:
==23765==    definitely lost: 4 bytes in 1 blocks
==23765==    indirectly lost: 0 bytes in 0 blocks
==23765==      possibly lost: 0 bytes in 0 blocks
==23765==    still reachable: 0 bytes in 0 blocks
==23765==         suppressed: 0 bytes in 0 blocks
==23765==
==23765== For counts of detected and suppressed errors, rerun with: -v
==23765== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind finds the memory leak because p is freed after usage.

5 Write a program that creates an array of integers called data of size 100 using malloc; then, set data[100] to zero. What happens when you run this program? What happens when you run this program using valgrind? Is the program correct?

gdb 5

run

```
[Inferior 1 (process 3691) exited normally]
```

gdb does not find anything wrong.

valgrind –leak-check=yes ./5

valgrind finds the invalid write.

6 Create a program that allocates an array of integers(as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use valgrind on it?

gcc -g 6.c -o 6

valgrind --leak-check=yes ./6

```
==6443== Memcheck, a memory error detector
==6443== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6443== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6443== Command: ./6
==6443==
==6443== Invalid read of size 4
==6443==    at 0x108700: main (6.c:7)
==6443==  Address 0x522f040 is 0 bytes inside a block of size 400 free'd
==6443==    at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6443==    by 0x1086FB: main (6.c:6)
==6443==  Block was alloc'd at
==6443==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6443==    by 0x1086EB: main (6.c:5)
==6443==
0
==6443==
==6443== HEAP SUMMARY:
==6443==     in use at exit: 0 bytes in 0 blocks
==6443==   total heap usage: 2 allocs, 2 frees, 1,424 bytes allocated
==6443==
==6443== All heap blocks were freed -- no leaks are possible
==6443==
==6443== For counts of detected and suppressed errors, rerun with: -v
==6443== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind finds the bug of an invalid read.

7 Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

```
free(): invalid pointer
Aborted (core dumped)
```

No, it does not need tools to find this type of problem.

8. Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use realloc() to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use realloc() to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use valgrind to help you find bugs.

```
==22212==
p0: 0
p1: 1
p2: 2
==22212== Invalid write of size 4
==22212==    at 0x1087CD: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Address 0x522f04c is 0 bytes after a block of size 12 free'd
==22212==    at 0x4C33D2F: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x1087C0: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Block was alloc'd at
==22212==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x108732: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==
==22212== Invalid read of size 4
==22212==    at 0x1087D7: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Address 0x522f040 is 0 bytes inside a block of size 12 free'd
==22212==    at 0x4C33D2F: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x1087C0: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Block was alloc'd at
==22212==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x108732: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==
p0: 0
==22212== Invalid read of size 4
==22212==    at 0x1087F4: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Address 0x522f044 is 4 bytes inside a block of size 12 free'd
==22212==    at 0x4C33D2F: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x1087C0: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Block was alloc'd at
==22212==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x108732: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==
p1: 1
==22212== Invalid read of size 4
==22212==    at 0x108811: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Address 0x522f048 is 8 bytes inside a block of size 12 free'd
==22212==    at 0x4C33D2F: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x1087C0: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Block was alloc'd at
==22212==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x108732: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==
p2: 2
==22212== Invalid read of size 4
==22212==    at 0x10882E: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Address 0x522f04c is 0 bytes after a block of size 12 free'd
==22212==    at 0x4C33D2F: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x1087C0: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==  Block was alloc'd at
==22212==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22212==    by 0x108732: main (in /home/zhuzuojun/learn/ostep/ch14/8)
==22212==
p3: 3
==22212==
==22212== HEAP SUMMARY:
==22212==     in use at exit: 0 bytes in 0 blocks
==22212==   total heap usage: 3 allocs, 3 frees, 1,040 bytes allocated
==22212==
==22212== All heap blocks were freed -- no leaks are possible
==22212==
==22212== For counts of detected and suppressed errors, rerun with: -v
==22212== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

There are valgrind error after realloc.

In vector, it takes O(1) to peek or add a value in the end. realloc() is like in a linked list, it takes O(n) to peek to add a value in the end.