Beyond Physical Memory: Mechanisms

Annalise Tarhan

February 16, 2021

1 Run mem.c with only 1 MB of memory. How do the CPU usage statistics change while running mem? Do the numbers in the user time column make sense? How does this change when running more than one instance of mem at once?

Activity Monitor shows the user percentage of the CPU time increases dramatically when mem is run, and continues to increase for each additional instance of mem. While mem is running, the system percentage increases as well.

2 Run ./mem 1024 and watch how swpd (the amount of virtual memory used) and free (the amount of idle memory) change. Then kill the running program and watch again how the values change. What do you notice? How does the free column change when the program exits? Does the amount of free memory increase by the expected amount when mem exits?

The free column's values change from around 465000 to around 190000 when mem starts running, and changes back as expected when mem is killed. swpd is not available on macOS.

3 First, examine how much free memory is available, then run mem with arguments approaching that number. Watch the swap in/out columns. Do they ever give non-zero values? What happens to these values as the program enters the second loop and beyond as compared to the first loop? How much data are swapped in and out during the second, third, and subsequent loops?

Any argument over 15000 caused free to drop to its minimum, around 3500. swapins and swapouts stay at 0 for quite a while for any argument, but once the swaps start they continue long after mem exits. The first few loops are quiet, but after loop 3 with an argument of 20000, swapins frequently hits the hundreds or thousands. swapouts start later, are less consistent, and come in much larger batches, sometimes tens of thousands at a time.

4 Do the same experiments as above, but now watch other statistics such as CPU utilization and block I/O statistics. How do they change while mem is running?

Using Activity Monitor, I can see that the CPU usage increases from only a few percent of capacity to around 25% while mem is running, split roughly evenly between user and system processes. Strangely, Activity Monitor only shows minimal disk reads and no disk writes, even for absurdly large arguments.

5 Pick an input for mem that comfortably fits in memory. How long does each loop take? Now pick a size comfortably beyond the size of memory. How long do the loops take now? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Finally, how does the performance of the first loop compare to that of subsequent loops?

To compare apples to apples, divide the time each loop takes by the argument given. The smaller allocation's ratio is just under one for the first loop and half of that for subsequent loops. The larger allocation's is just over one for the first loop and even higher on later loops. The important difference is what happens after the first loop. When the array fit in memory, the time was halved. When it was too large, it took even longer. As you would expect, the bandwidth numbers for the small array were higher than for the large array, especially after the first loop.

6 Swap space isn't infinite. What happens if you try to run mem with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?

The system eventually kills the process when given an argument somewhere between 100000 and 150000. This is much, much larger than the available swap space, though.