Report: Observing the Overhead from Swapping Pages in Memory
CS574
Long Tran

Introduction

Paging allows all processes in an operating system to have access to the full memory address space.
Paging is done by cutting the address space into fixed-size chunks (called pages) and only putting the pages under use in memory while other pages will be stored in the secondary storage.
When a running process is accessing data from a page that is not in memory and the memory is full, a page in the memory will be moved to the secondary storage, and the requested page will be put on the available spot. This swapping of pages incur an overhead which increases the running time of the running process.
This overhead comes from the extra operations needed to swap the pages and the much slower transfer speed of the secondary storage compared to the speed of a CPU.

The goal of this experiment is to observe that overhead.

Methodology

Experiment Design

The experiment was done as follows:

1) Use the “top” command to see the maximum amount of memory, the amount of free memory, and the amount of swap space.
2) Run a program that will create the maximum amount of memory as follows:
   While we have not reached the maximum amount of memory {
   Fill any created memory to keep them on RAM.
   Continue to fill any created memory on a separate parallel thread.
   Start timer.
   Create 1 GB of memory and fill it.
   End Timer
   Get the duration of creating and filling that 1GB of memory.
   }

The idea of this program is that the time taken to create and fill 1GB of memory when the memory has not reached the maximum will be faster than the time taken to do so when the memory has reached the limit.
This is because creating and filling 1GB of memory when the memory has reached the limit will require pages swapping.

The tested platforms
The experiment was executed on 3 different platforms:
1. poobah.cs.nmsu.edu machine:
   - OS: OpenSUSE Leap 15.5.
   - CPU: Intel i7-2600 (from [1] and [2]):
     - 4 cores, 8 threads, 1.6GHz-3.40 GHz per core.
     - L1 Cache: 128 KiB.
     - L2 Cache: 1 MiB.
     - L3 Cache: 8 MiB.

2. Macbook Air M2:
   - OS: macOS 14.3 (23D56).
   - CPU: Apple Silicone M2 (from [3] and [4]):
     - 4 efficiency cores:
       - L1 Instruction Cache: 128 KB.
       - L1 Data Cache: 64 KB.
       - L2 Cache: 4 MB.
     - 4 performance cores (higher clock frequency):
       - L1 Instruction Cache: 198 KB.
       - L1 Data Cache: 128 KB.
       - L2 Cache: 16 Mb.
     - In total, there are 8 cores/8 threads.
     - System Cache: 8 MB.

3. Ubuntu Laptop:
   - OS: Ubuntu 22.04.03 LTS
   - CPU: AMD Ryzen 7 5800H with Radeon Graphics (from [5] and [6]):
     - 8 cores, 16 logical processors (threads), 3.2GHz-4.4GHz.
     - L2 Cache: 4 MB.
     - L3 Cache: 16 MB.

Time Measurement
The time reported for each Filled Index (see “Raw Data” section to know what “Filled Index” is) was an average of 5 runs. The standard deviation of the 5 runs was also reported. The time was measured in nanoseconds using the clock_gettime() function with CLOCK_MONOTONIC.

Determining the Available Amount of Memory

The maximum amount of memory, the average amount of free memory, and the size of the swap space for each platform was determined using the “top” commands. For MacOs, it was “top -S”.
For each platform, the average amount of free memory reported was an average of the amount of free memory in the 5 runs. The units used in this report follow this convention: 1 GB = 1024 MB = 1024 * 1024 KB = 1024 * 1024 * 1024 B.

Raw Data
Note: “Filled GB” is the index number of the GB filled. For example, the time taken of “Filled GB” 5 means that 4 GB has already been created and filled and the time measured was for creating and filling the 5th 1GB of memory.
Time was measured in nanoseconds. The average and standard deviation was reported for each “Filled GB”.

Table 1: The data of running the experiment on MacOs Platform (Nanoseconds)

Filled GB	Time Taken (RUN 1)	Time Taken (RUN 2)	Time Taken (RUN 3)	Time Taken (RUN 4)	Time Taken (RUN 5)	AVERAGE	STANDARD DEVIATION
1	1050523000	1021337000	1038228000	1054290000	1037907000	1040457000	12939489.81
2	1041538000	1015166000	1026081000	1026642000	1037404000	1029366200	10399740.73
3	1044062000	1052616000	1055875000	1047173000	1102020000	1060349200	23744270.63
4	1185593000	1134873000	1159942000	1158510000	1146077000	1156999000	18956602.85
5	1212189000	1209664000	1203748000	1222367000	1196073000	1208808200	9795333.108
6	1191287000	1174657000	1170286000	1156638000	1169885000	1172550600	12459276.48
7	1165478000	1182079000	1176599000	1181377000	1191152000	1179337000	9365400.605
8	1172147000	1162302000	1175727000	1179730000	1173216000	1172624400	6466171.997
9	1178776000	1190446000	1187799000	1185481000	1186279000	1185756200	4337035.358
Free	2.38281	2.840820	2.6875	2.8378906	2.099609	2.56972	0.329920899

Memory (GB)	25	313		25	375	6563	6

Table 2: The data of running the experiment on Ubuntu Platform (Nanoseconds)

Filled GB	**Time Taken (RUN 1)**	**Time Taken (RUN 2)**	**Time Taken (RUN 3)**	**Time Taken (RUN 4)**	**Time Taken (RUN 5)**	**AVERAGE**	**STANDARD DEVIATION**
1	2065857881	1993943853	1959956223	1989105389	1965150389	1994802747	42354374.7
2	2026443374	1971493095	1939942227	1946664032	1942691250	1965446796	36325715.4
3	1982596880	1986861668	1939940551	1944103021	1970500510	1964800526	21692735.48
4	1989310932	1978505508	1935424963	1939731935	1950045955	1958603859	24008583.99
5	2020564554	1979399197	1940239751	1940493903	1953293785	1966798238	34019268.17
6	1998751028	1976774702	1940054741	1945637714	1953705709	1962984779	24403867.47
7	1999830145	2002310559	1944483168	1945332786	1950014108	1968394153	29916266.87
8	2031604854	1984379719	1945347593	1954273238	1956069906	1974335062	35203303.38
9	1995017523	1990074926	1946207758	1970552681	1956744921	1971719562	20953329.1
10	199249276	2009776	19515121	19862499	1959569	1979	24034989.

	1	801	33	28	443	920213	86
11	2005216235	2003570355	1958029163	1980234359	1971106803	1983631383	20538664.67
12	1998730355	2080499826	1955967518	1967048675	1961201773	1992689629	51828028.38
13	2162768592	2470927660	2086217242	2091677294	2112446065	2184807371	162772779.5
Free Memory (GB)	12.28027344	11.35351563	12.31738281	12.26855469	11.88242188	12.02042969	0.4225431718

Table 3: The data of running the experiment on Poobah Platform (Nanoseconds)

Filled GB	**Time Taken (RUN 1)**	**Time Taken (RUN 2)**	**Time Taken (RUN 3)**	**Time Taken (RUN 4)**	**Time Taken (RUN 5)**	**AVERAGE**	**STANDARD DEVIATION**
1	2418246560	2412979933	2411416184	2405162074	2405551261	2410671202	5473471.773
2	2426998017	2421246761	2420951481	2413462646	2413887232	2419309227	5682188.554
3	2438690096	2428667878	2429249118	2424139761	2424901682	2429129707	5796644.273
4	2426120234	2459931925	2443621024	2436001990	2443040959	2441743226	12380212.82
5	2456173214	2451200654	2454189763	2449522029	2446514307	2451519993	3805616.488
6	2469263854	2456137201	2457159232	2450756790	2449289529	24565213	7881231.805

						21	
7	2473630845	2467134666	2467697543	2460969176	2459278935	2465742233	5756487.651
8	2483212378	2476156213	2475604924	2469136226	2466974969	2474216942	6419987.984
9	2491829993	2483923137	2484353160	2477157754	2478320613	2483116931	5844059.731
10	2499771356	2491071330	2493853227	2487637851	2484168303	2491300413	5970920.513
11	2506114101	2501297974	2503531286	2498930372	2495408957	2501056538	4128479.306
12	2499536873	2514420328	2514645853	2509738514	2507391209	2509146555	6201463.604
13	2535617337	2524070908	2526020993	2519932512	2518610404	2524850431	6727332.32
14	2672939138	2530249447	2528961999	2525592227	2521571134	2555862789	65533945.04
15	2781744722	2737496353	2633822825	2717829606	2657870017	2705752705	59989492.46
16	2612626243	2626619171	2654115301	2625164141	2624148362	2628534644	15343696.03
17	2975523832	2487011005	2477485630	2461939637	2509475914	2582287204	220499994.6
Free Memory	15.24316406	15.20703125	15.23632813	15.22949219	15.21191406	15.225585	0.01595305613

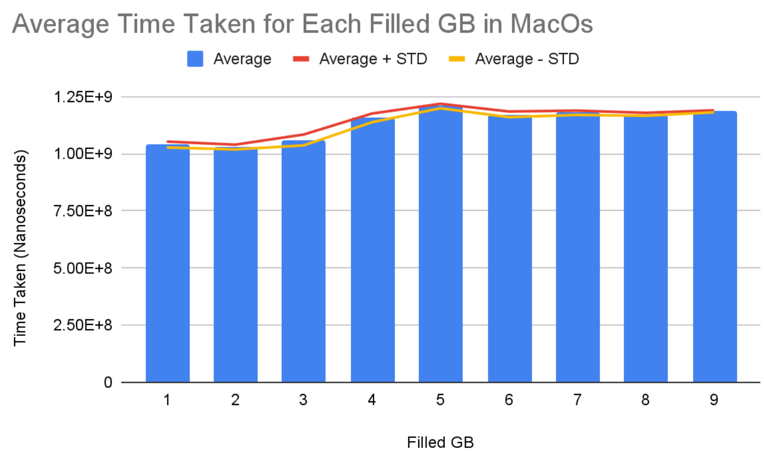
(GB)						94	

Graphs

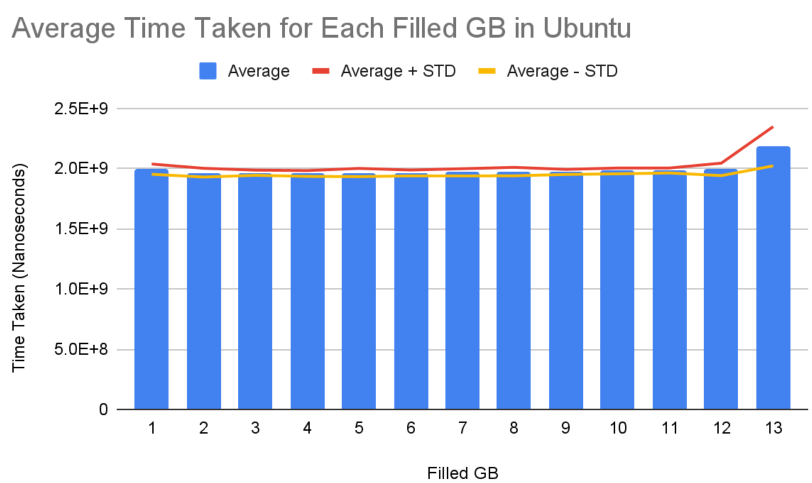
Note: See "Raw Data" section to understand what "Filled GB" is.
Also, in the graphs below:

1) *"Average" is the average runtime of each "Filled GB"*
2) *"Average + STD" is the average runtime plus the standard deviation of each "Filled GB".*
3) *"Average - STD" is the average runtime minus the standard deviation of each "Filled GB".*

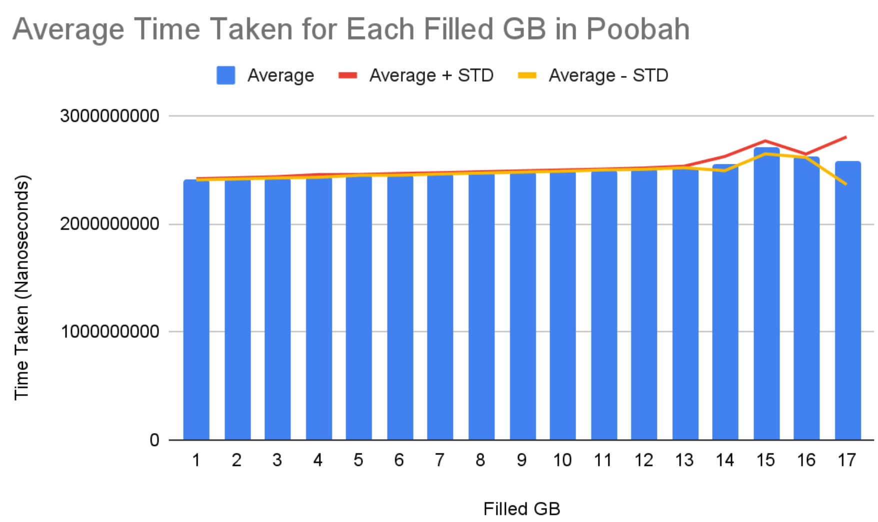
Graph 1: Average Time Taken for Each Filled GB in MacOs



Graph 2: Average Time Taken for Each Filled GB in Ubuntu



Graph 3: Average Time Taken for Each Filled GB in Poobah



Aggregated Data

Table 4: The Maximum Amount of Memory, the Average Amount of Free Memory, and the Amount of Swap Space for each platform.

	MacOs	**Ubuntu**	**Poobah**
Maximum Amount of Memory	7619 MB	13,817.7 MB	15,949.07 MB
Average Amount of Free Memory	2.56 GB	12.0 GB	15.2 GB
Amount of Swap Space	2 GB	2 GB	4 GB

Table 5: The average time taken to allocate new memory when there is available free memory vs when there is not enough available free memory for MacOs.

Allocating Scenario	**Average Time Taken (Nanoseconds)**
Available Free Memory	1034911600
Not Enough Available Free Memory	1162346371

Table 6: The average time taken to allocate new memory when there is available free memory vs when there is not enough available free memory for Ubuntu.

Allocating Scenario	**Average Time Taken (Nanoseconds)**
Available Free Memory	1973677246
Not Enough Available Free Memory	2184807371

Table 7: The average time taken to allocate new memory when there is available free memory vs when there is not enough available free memory for Poobah.

Allocating Scenario	**Average Time Taken (Nanoseconds)**
Available Free Memory	2499227676
Not Enough Available Free Memory	2605410924

Table 8: The time differences of allocating 1GB of memory when there is available free memory vs when there is not enough available free memory on MacOs, Ubuntu, and Poobah.

Platforms	**Time Differences (Milliseconds)**
MacOs	127.4347714
Ubuntu	211.1301251
Poobah	106.1832478

Example Outputs of Running "top" on the 3 Platforms

MacOs

```
Processes: 326 total, 3 running, 323 sleeping, 1903 threads                   14:23:16
Load Avg: 3.50, 2.99, 2.33  CPU usage: 12.93% user, 7.13% sys, 79.92% idle
SharedLibs: 505M resident, 104M data, 27M linkedit.
MemRegions: 126952 total, 355M resident, 69M private, 604M shared.
PhysMem: 6547M used (1280M wired, 3114M compressor), 1071M unused.
VM: 132T vsize, 4773M framework vsize, 27170(4) swapins, 81532(0) swapouts.
Swap: 850M + 1198M free. Purgeable: 45M 15262(0) pages purged.
Networks: packets: 1734140/1792M in, 721888/319M out.
Disks: 2485367/45G read, 708748/10G written.
```

Ubuntu

```
top - 13:47:19 up 52 min,  1 user,  load average: 0.29, 0.73, 0.87
Tasks: 375 total,   2 running, 373 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.4 us,  2.0 sy,  0.0 ni, 89.1 id,  1.5 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem :  13817.7 total,  11626.7 free,   1896.8 used,    294.1 buff/cache
MiB Swap:   2048.0 total,   1241.5 free,    806.5 used.  11638.3 avail Mem
```

Poobah

```
top - 13:51:48 up  8:51,  4 users,  load average: 0.00, 0.23, 0.57
Tasks: 200 total,   1 running, 199 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 15949.07+total, 15572.86+free,  459.914 used,  160.102 buff/cache
MiB Swap: 4096.457 total, 3795.594 free,  300.863 used. 15489.15+avail Mem
```

Analysis

From Graph 1 about MacOs (with around 2.6GB of free memory), we can see that allocating an extra 1GB of RAM and writing to that new memory when the RAM is full takes a slightly longer duration than doing so when there is still enough available space.

According to table 8, with MacOs, we can see that the average time taken to allocate an extra 1GB of memory when there is not enough free memory is 127,434,771 nanoseconds or around 127 milliseconds longer than doing so when there is enough free memory. 127 milliseconds is a very long duration in terms of CPU speed. Thus, this extra duration is likely to come from the pages swapping process and the much slower speed of secondary storage (SSD in MacOs).

We can see similar trends in Ubuntu (with around 12 GB of free memory) and Poobah (with around 15.2 GB of free memory) from Graph 2 and 3. More specifically, according to table 8, the extra time caused by pages swapping in Ubuntu and Poobah are 211 and 106 milliseconds, respectively.

Hence, there is definitely overheads from page swapping. However, the amount seems to not be very significant compared to the amount of memory allocated (1 GB). This must be due to the fast secondary storage technology (SSD) that is used on the tested platforms.

I also observed that in Ubuntu, I always got killed when trying to create a 14th 1 GB of memory (only 1GB more than the maximum amount of memory), while it was not the case for the others (successfully created 1 GB more than the maximum amount of memory).
According to the “Learn Linux” website, the reason for the process to die in Ubuntu was because I filled up more than the sum of both the available amount of free memory and the free swap space.
Although not shown in the data section, as can be seen from the above picture of running “top”, the amount of free swap space in Ubuntu before each run is only around 1GB while the amount of free memory is around 12 GB. Thus, the system can only contain around 13 GB worth of pages at runtime. Hence, when trying to create the 14th GB of memory, the process was killed.

At the same time, on MacOs and Poobah, the processes were not killed because the free amount of swap space was much bigger than 1GB.

However, I was not able to kill the process on MacOs even after creating 30GB of memory, which is 3 times the sum of the maximum amount of memory and the size of the swap space.
I think this is because of the memory compression mechanism of MacOs.
According to Nelson (2020) on Lifewire, MacOs will compress inactive memory pages on the RAM without paging out to the disk. Thus, the simplistic nature of the data I stored in memory (an array of characters ‘a’) allows MacOs to easily compress memory to make space for 30GB of data. This could also be because my mechanism of keeping the pages active is not good enough for a large amount of memory, causing pages to be compressed.

Also, I was not able to find any additional sources that did the same experiment to compare my observations of the overheads of page swapping.

Conclusion

To conclude, page swapping will cause delay in the running process and when a process fills up both the memory and the swap space, the process will get killed. However, the delay from the overhead is not a lot due to the fast SSD used as secondary storage in modern systems. Also, due to memory compression, it is difficult to fill up so much memory that the experiment’s process gets killed in MacOs.

References

Paging and Swapping. *Learn Linux*. Retried on March 24 from
https://www.learnlinux.org.za/courses/build/internals/ch05s03#:~:text=If%2C%20after%20all%20of%20that,the%20kernel%20starts%20killing%20processes.
Nelson T (2020). *Lifewire.* Retrieved from
https://www.lifewire.com/understanding-compressed-memory-os-x-2260327

Appendix A - Code

```
// A program to measure the amount of time taken caused by page swapping.
// CS574
// Author: Long Tran
// Date: Feb 10, 2024
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <math.h>
```

```c
#include <string.h>

enum unit {
   GB,
   B
};

long max_size = 0;
enum unit max_unit = 0;
long chunk_size = 0;
enum unit chunk_unit = 0;

long max_size_in_bytes = 0;
long chunk_increase_in_bytes = 0;

long long get_duration(struct timespec start, struct timespec end) {
  long j = end.tv_sec - start.tv_sec;
  long k = end.tv_nsec - start.tv_nsec;
  return j * 1000000000 + k;
} // end get_duration

char ** chunks = NULL;

void * touch_memory(void *arg) {
   srand(time(NULL));
   int index = *((int *) arg);
   free(arg);

   for (int i = 0; i < index; ++i) {
       memset(chunks[i], 'a', chunk_increase_in_bytes);
   } // end for i
   return NULL;
} // end touch_memory

void start_experiment() {
   int num_chunks = max_size_in_bytes / chunk_increase_in_bytes;
   pthread_t thread_id;
   chunks = (char **) malloc(num_chunks * sizeof(char *));

   for (int i = 0; i < num_chunks; ++i) {
```

```
        for (int j = 0; j < i; ++j) {
            for (int k = 0; k < chunk_increase_in_bytes; ++k) {
                chunks[j][k] = ('a' + j) % 26;
            } // end for k
        } // end for j

        int * value = malloc(sizeof(int));
        *value = i;
        pthread_create(&thread_id, NULL, touch_memory, (void*) value);

        // timer start here
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);
        chunks[i] = (char *) malloc(chunk_increase_in_bytes);
        for (int j = 0; j < chunk_increase_in_bytes; ++j) {
            chunks[i][j] = 'a';
        } // end for j
        // timer end here
        clock_gettime(CLOCK_MONOTONIC, &end);
        long long duration = get_duration(start, end);
        printf("%d, %lld\n", i, duration);

        pthread_join(thread_id, NULL);
    } // end for i

    for (int i = 0; i < num_chunks; ++i) {
        free(chunks[i]);
    } // end for i
    free(chunks);
}

int main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <GB Max Units to Allocate> > <Chunk Size
Increase> <Unit>\n", argv[0]);
        fprintf(stderr, "Usage: %s <Unit>: gb or b\n", argv[0]);
        return 1;
    } // end if
    max_size = atoi(argv[1]);
```

```
    chunk_size = atoi(argv[2]);

    if (strcmp(argv[3], "gb") == 0) {
        chunk_unit = GB;
    } else if (strcmp(argv[3], "b") == 0) {
        chunk_unit = B;
    } else {
        fprintf(stderr, "Usage: %s <Unit>: gb or b\n", argv[0]);
        return 1;
    } // end if

    if (chunk_unit == GB) {
        chunk_increase_in_bytes = chunk_size * 1024 * 1024 * 1024;
    } else {
        chunk_increase_in_bytes = chunk_size;
    } // end if

    max_size_in_bytes = max_size * 1024 * 1024 * 1024;

    start_experiment();

    // exit
    return 0;
} // end main
```