

## 1 Objective

The main goal of the lab #1 is to learn about methods that measure a program's behavior by specifically using `mmap()`. There are many ways to trace operations of an OS and the hardware, but in this lab, **perf** and **getrusage()** are mainly exploited. Several experiments to reach the objective is performed by using **mmap()**.

`mmap()` tries to generate a new mapping in the virtual address space of the calling process. Users can specify not only the length of the mapping, but also detailed mapping methods: i.e. file-backed or anonymous area mapping. By changing a combination of the parameters based on an objective, both hardware and software operations for the memory mapping also change, these cause trade-offs among memory consumption, processing time, and functions. Therefore, it is important to consider the trade-offs and make proper configuration choices for successful mapping. This report ferrets out internal system actions by executing several kinds of experiments. Perf and `getrusage()` are good ways to accomplish this goal since it shows not only hardware event counts, but also resource consumption statistics. This report also introduces how to use them in order to extract meaningful information. Finally, it summarizes aggregated results and analyzes them.

## 2 Hardware/Software Specification

The host machine consists of a single socket system with Intel i5-2467M CPUs with 2 cores and 4 hyper-threads. Also, it has 8GB of DRAM and 3072K last level cache and Transparent Huge Pages (THP) is disabled. TLB has 4-way 32-entries of 2M/4M pages. The VM is set as a single socket with 2 cores. In this experiment, I disabled hyper threads in order to use more HW event counters. It also exploits 4GB DRAM whose THP is disabled and 32KB Last Level Cache (LLC). VM has the same TLB spec. Those settings are half of the host machine's resources since the host machine also required HW resources for real-time VM analysis such as GDB.

To construct the VM, I used Virtual Machine Builder 1.5.1 which is graphic based virtual machine manager application. Users can generate a virtual machine just by passing ISO file path. The generated VM runs based on KVM and QEMU of 3.1.50 version. KVM supports a full-virtualization system for Linux. QEMU supports both machine emulation and virtualization, but in this context, it virtualizes x86-64 architecture. I installed related packages through Ubuntu Advanced Packaging Tool (APT) except QEMU: `qemu-kvm`, `libvirt-daemon-system`, `libvirt-clients`, `bridge-utils` and `libvirt-bin`. I had to clone QEMU source code from the git repository, modified a source code and compiled it directly because of the compatibility issue between QEMU and GDB of the recent Linux kernel. This issue is treated in the later section: "Running a VM in KVM". I installed Ubuntu 18.04.1 LTS that adopts Linux kernel 4.15 for the initial VM construction. Then, I replaced the kernel with Linux kernel 4.20, the more recent version. Inside the VM, `vim`, `gcc 7.3`, `g++ 7.3` and `openssh-client` are used for the analysis. Finally, `guestmount`, `guestfish`, and `fusemount` are used to mount and un-mount the VM disk.

In this lab, I mainly carried out performance evaluation of several environment. To do this, `perf` of version 4.20.6, `gdb` of version 8.1.0 and, `objdump` and `readelf` of version 2.3 were used.

## 3 Understanding Memory Map

### 3.1 Analysis of `/proc/self/maps`

This section explains how to read and understand '`/proc/self/maps`'. '`procfs` (proc filesystem)' provides a process or system information as a file format in Unix based OS; Each file size is '0' since files actually do not exist in a disk or in the memory, but each information is constructed during a runtime whenever users try to access or read the files. Among the files, '`/proc/self/maps`' contains a region of contiguous virtual memory in the current process. Since it is treated as a file, we can easily read them by using general file I/O codes in C: `open()` and `read()`. (the used code is attached to the submission.) Below is a result of printing the '`/proc/self/maps`'.

Figure 1: Printing '`/proc/self/maps`'

```
lhc@lhc-vm:~/Practice/lab1/final$ ./lab1 -p 0
Selected problem type: 0
562510ae0000-562510ae3000 r-xp 00000000 08:01 921058 /home/lhc/Practice/lab1/final/lab1
562510ce3000-562510ce4000 r--p 00003000 08:01 921058 /home/lhc/Practice/lab1/final/lab1
562510ce4000-562510ce5000 rw-p 00004000 08:01 921058 /home/lhc/Practice/lab1/final/lab1
562510db2000-562510dd3000 rw-p 00000000 00:00 0 [heap]
7f3bf4fee000-7f3bf51d5000 r-xp 00000000 08:01 1185640 /lib/x86_64-linux-gnu/libc-2.27.so
7f3bf51d5000-7f3bf53d5000 ---p 001e7000 08:01 1185640 /lib/x86_64-linux-gnu/libc-2.27.so
7f3bf53d5000-7f3bf53d9000 r--p 001e7000 08:01 1185640 /lib/x86_64-linux-gnu/libc-2.27.so
7f3bf53d9000-7f3bf53db000 rw-p 001eb000 08:01 1185640 /lib/x86_64-linux-gnu/libc-2.27.so
7f3bf53db000-7f3bf53df000 rw-p 00000000 00:00 0
7f3bf53df000-7f3bf5406000 r-xp 00000000 08:01 1185612 /lib/x86_64-linux-gnu/ld-2.27.so
7f3bf55f1000-7f3bf55f3000 rw-p 00000000 00:00 0
7f3bf5606000-7f3bf5607000 r--p 00027000 08:01 1185612 /lib/x86_64-linux-gnu/ld-2.27.so
7f3bf5607000-7f3bf5608000 rw-p 00028000 08:01 1185612 /lib/x86_64-linux-gnu/ld-2.27.so
7f3bf5608000-7f3bf5609000 rw-p 00000000 00:00 0
7ffe24859000-7ffe2487a000 rw-p 00000000 00:00 0 [stack]
7ffe249b6000-7ffe249b9000 r--p 00000000 00:00 0 [vvar]
7ffe249b9000-7ffe249bb000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Each row of '`/proc/self/maps`' demonstrates a region of the virtual memory in the current executing process. Each process or library consists of several memory regions that are split based on permission: read, read-execute and read-write. The six columns mean memory address, permission, offset, dev, inode and pathname of the corresponding region. The most interesting fact that can be found on the printed results is that the process is partitioned into three regions. Moreover, the regions have different permission even though they are a part of one process. I wanted to understand how each region was mapped onto memory sections and how they are used. So, I decided to dig deeper '`/proc/self/maps`'.

To do this, I exploited `readelf`; `readelf` with '`-s`' option prints each section header information of the file including virtual address to be used. Before doing this, I compiled the code again with PIE disabling option. The PIE loads the segments' base address arbitrarily. Therefore, if a binary is compiled with the PIE option, statically known virtual address is changed during the runtime and it is hard to expect the actually mapped virtual address. Below is the commands that I exploited in order to compare `readelf` and '`/proc/self/maps`'.

```
# gcc -o lab1_test -no-pie lab1.c
# readelf -l lab1_test
```

In addition, below figures are the results of the above commands. The upper part is the result of '`/proc/self/maps`' and the lower part is the result of '`readelf -l`'. My process named '`lab1_test`' consists of three memory areas. Address ranges printed by '`/proc/self/maps`' are marked by blue, green, and yellow boxes. Below color boxes highlights corresponding section headers that are included within the address range. The first memory area of blue box in

the ‘/proc/self/maps’ consists of .interp, .note.ABI-tag, .note.gnu.build-id, .gnu.hash, .dynsym, .rodata, .eh\_frame\_hdr and .eh\_frame sections. The second memory area in the green box is composed by .init\_array, .fini\_array, .dynamic, .got and .got.plt sections. Lastly, the orange box memory area consists of .data and .bss sections.

Figure 2: Comparison readelf and ‘/proc/self/maps’

/proc/self/maps							
00400000-00403000	r-xp	00000000	08:01 922092	/home/lhc/Practice/lab1/final/lab1_test			
00602000-00603000	r--p	00002000	08:01 922092	/home/lhc/Practice/lab1/final/lab1_test			
00603000-00604000	rw-p	00003000	08:01 922092	/home/lhc/Practice/lab1/final/lab1_test			
Section Headers:				readelf -s lab1_test			
[Nr]	Name	Type	Address	Offset			
	Size	EntSize	Flags	Link	Info	Align	
[ 0]		NULL	0000000000000000	00000000			
[ 1]	.interp	PROGBITS	0000000000000238	00000238			
[ 2]	.note.ABI-tag	NOTE	0000000000000254	00000254			
[ 3]	.note.gnu.build-id	NOTE	0000000000000274	00000274			
[ 4]	.gnu.hash	GNU_HASH	0000000000000298	00000298			
[ 5]	.dynsym	DYNSYM	00000000000002c8	000002c8			
[ 6]	.dynstr	STRTAB	0000000000000070	00000070			
[ 7]	.gnu.version	VERSYM	00000000000007c6	000007c6			
[ 8]	.gnu.version_r	VERNEED	0000000000000818	00000818			
[ 9]	.rela.dyn	RELA	0000000000000858	00000858			
[10]	.rela.plt	RELA	00000000000008d0	000008d0			
[11]	.init	PROGBITS	0000000000000be8	00000be8			
[12]	.plt	PROGBITS	0000000000000c00	00000c00			
[13]	.text	PROGBITS	0000000000000e20	00000e20			
[14]	.fini	PROGBITS	000000000000021f0	000021f0			
[15]	.rodata	PROGBITS	0000000000000200	00002200			
[16]	.eh_frame_hdr	PROGBITS	0000000000000278	00002878			
[17]	.eh_frame	PROGBITS	0000000000000358	00002950			
[18]	.init_array	INIT_ARRAY	00000000000002e10	00002e10			
[19]	.fini_array	FINI_ARRAY	00000000000002e18	00002e18			
[20]	.dynamic	DYNAMIC	00000000000002e20	00002e20			
[21]	.got	PROGBITS	00000000000002ff0	00002ff0			
[22]	.got.plt	PROGBITS	00000000000003000	00003000			
[23]	.data	PROGBITS	00000000000003120	00003120			
[24]	.bss	NOBITS	00000000000003160	00003150			
[25]	.comment	PROGBITS	0000000000000000	00003150			
[26]	.syntab	SYMTAB	0000000000000001	00003180			
[27]	.strtab	STRTAB	0000000000000000	00003d20			
[28]	.shstrtab	STRTAB	0000000000000000	000042cc			

### 3.2 Analysis of libc start address

When you check the ‘/proc/self/maps’ result, you can easily find that **libc** has very different start address compared to the base address of the process, lab\_1. This occurs because of ASLR (Address Space Layout Randomization) technique. libc is one of the shared libraries and several processes exploit one library object. Therefore, it should be protected from a malicious process. ASLR randomizes the location of the libc’s object and prevents from buffer-overflow attacks. However, since it only randomizes virtual memory, the physical address of the libc is fixed.

## 4 Getting resource usages by getrusage()

getrusage() is a useful function that aggregates resource usage statistics such as CPU time used, several kinds of memory usage and the number of page reclaim/faults and IPC/block I/O, etc. I called the function reading ‘/proc/self/maps’ program mentioned above. (In addition, all other programs that will be introduced also uses getrusage() in order to understand system behaviors.) Figure 3 shows the results of the getrusage().

Since the process that prints /proc/self/maps is simple and light, small amount of resources were consumed. For example, execution time for user and kernel space were almost ‘0’ and hardware page faults did not occur.

Figure 3: `getrusage()` of printing `/proc/self/maps`

```
**** Getrusage Results ****
[user CPU time used],seconds,0,microseconds,0
[system CPU time used],seconds,0,microseconds,9883
[maximum resident set size],2628
[page reclaim (soft page faults)],79
[page faults (hard page faults)],0
[block input operations],0
[block output operations],0
[the number of swaps],0
[the number of voluntary context switches],0
[the number of involuntary context switches],0
```

## 5 `perf_event_open()`

‘perf’ is a representative tool which monitors performance of the system. Internally, it calls one of system call named `perf_event_open()` in order to set up performance monitoring. Therefore, by properly calling `perf_event_open()`, we also can trace and get specific HW events information for the executing process. This section introduces overall processes which are from how to build the perf tools on VM to how to trace and analyze measured performance information.

### 5.1 How to build the perf tools on VM

‘perf’ can be easily built on VM by using package tools of OS; in case of Ubuntu, we can install it by using the below command.

```
# apt-get install linux-tools-common linux-tools-generic linux-tools-‘uname -r’
```

However, I could not use the commands since my VM uses the recent Linux kernel, 4.20. Corresponding linux-tools are not deployed yet. Fortunately, Linux kernel provides ‘perf’ source code on its directory. I could build it and copy the generated binary to VM. By using below commands, perf was successfully installed on VM.

```
# cd linux-4.20.6/tools/perf
// binary file is generated
# make -j
# sudo guestmount -a <vm_image>.qcow2 -m /dev/sda1 <mount_dir>
// copy the binary to VM
# sudo cp perf <mount_dir>/usr/bin
# sudo fusermount -u <mount_dir>
```

Several kernel configuration should be set in order to build and appropriately use ‘perf’. Below are the required kernel configuration lists.

## Required Kernel Configuration for perf

```
# for perf_events:
CONFIG_PERF_EVENTS=y
# for stack traces:
CONFIG_FRAME_POINTER=y
# kernel symbols:
CONFIG_KALLSYMS=y
# tracepoints:
CONFIG_TRACEPOINTS=y
# kernel function trace:
CONFIG_FTRACE=y
# kernel-level dynamic tracing:
CONFIG_KPROBES=y
CONFIG_KPROBE_EVENTS=y
# user-level dynamic tracing:
CONFIG_UPROBES=y
CONFIG_UPROBE_EVENTS=y
# full kernel debug info:
CONFIG_DEBUG_INFO=y
# kernel lock tracing:
CONFIG_LOCKDEP=y
# kernel lock tracing:
CONFIG_LOCK_STAT=y
# kernel dynamic tracepoint variables:
CONFIG_DEBUG_INFO=y
```

## 5.2 Preparation to correctly monitor HW events

### Prerequisite Configuration

To correctly and successfully measure each HW events, we need to check whether a system environment is correctly set. First, permission for HW event counters is required. If a user does not have a sufficient permission, some HW event counters would not be seen and also could not be used. Therefore, I gave the top permission to my account. The permission of ‘perf’ is managed by ‘/proc/sys/kernel/perf\_event\_paranoid.’ Below command allow the current user to utilize all HW event counters provided by a system.

```
# sudo sh -c 'echo -1 < /proc/sys/kernel/perf_event_paranoid'
```

Second, we should check how many and what kinds of events can be measured. Not only the number of the counters, but also measurable event types are decided by a processor. We can check the number of counters by using below command.

```
# cpuid|grep 'number of counters per logical processor'
number of counters per logical processor = 0x4 (4)
```

One more thing I had to consider on this was hyperthreading. Most of Intel processors provide this technology which partitions a physical processor into more than two logical processors. I figured out that during the hyperthreading, HW event counters are also partitioned across the logical processors. For example, if a physical processor supports four event counters and hyperthreading is applied to it, each logical processor starts to own only two event counters. As a result, when I measured four events simultaneously, HW events were not counted at all and printed out '0' as the measured data. This was solved after disabling the hyperthreading option on the BIOS.

## Cache Flush

One of the objectives is to trace and understand the level-1 data cache miss rates according to system environments or options. Therefore, to do more correct experiments, we have to flush the cache. There are several ways to do that; a hardware vendor could support such a function, but in this lab, I cleaned the cache by reading a big memory buffer which is larger than VM's cache size. Below is the source code that flushes the data cache.

```
void cache_flush(void) {
    char *tmp = (char *) malloc (L1D_CACHE_SIZE*sizeof(char));
    for (int i = 0; i < L1D_CACHE_SIZE; i++)
        tmp[i] = simplerand();
    free(tmp);
}
```

## Process Locking

This lab requires to understand cache and TLB miss according to variable factors including memory size or types of memory operations. Therefore, narrowing down to a single core is a good way to extract information of the memory behaviors since other processors could not critically effect the memory; However, there is the experiment which is performed under multicores. Below code locks a process onto core 0.

```
#define _GNU_SOURCE
#include <sched.h>

void lock_process(int coreid) {
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(coreid, &mask);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &mask) == -1)
    {
        assert(false);
    }
}
..
lock_process(0);
```

`sched_setaffinity()` decides the set of CPUs which are able to run. This function accepts `cpu_set_t` variable that contains which cores will be allowed to perform the process. `CPU_ZERO()` or `CPU_SET` are macros to handle CPU sets.

## Input File Creation by fallocate

`mmap()` supports two types of mappings: Anonymous based and file-backed based mappings. To perform experiments with the file-backed based `mmap()`, I needed to generate a 1G size input file. **fallocate** was a great option to create a zero-filled file of specified size.

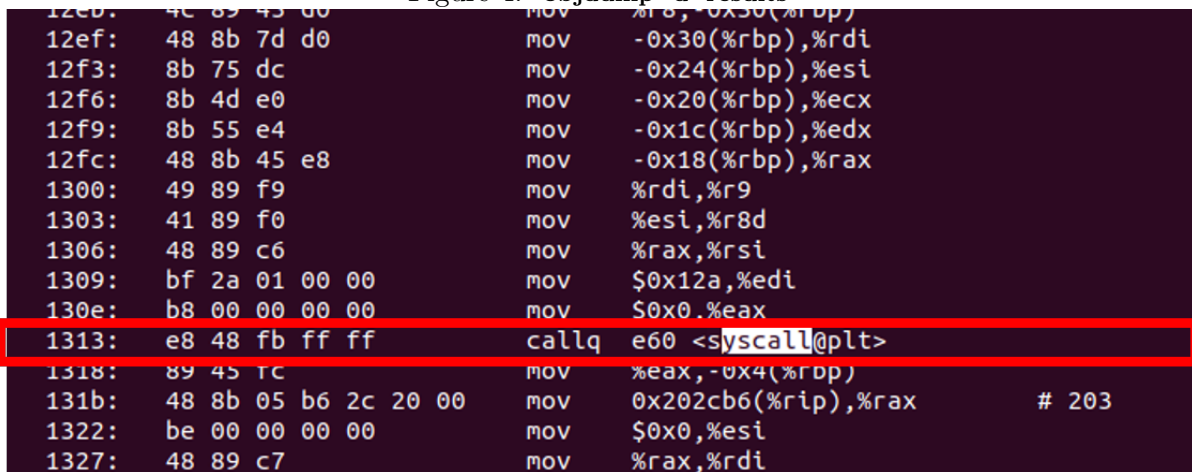
```
# fallocate -l 1G test1.input
```

Above is a command for the 1G file creation. The last parameter is the file name to be created and 'l' option accepts a size of the file, 1G.

## 5.3 Used System Call

The `perf_event_open()` calls syscall named `__NR_perf_event_open`. Below figure is the result of 'objdump -d [binary].' You can see the syscall is called at 0x1313 in the red box. This syscall opcode is 'e8 48 fb ff ff.'

Figure 4: 'objdump -d' results



```
12e0: 4c 89 43 00      mov     %r8, -0x30(%rbp)
12ef: 48 8b 7d d0      mov     -0x24(%rbp),%esi
12f3: 8b 75 dc        mov     -0x20(%rbp),%ecx
12f6: 8b 4d e0        mov     -0x1c(%rbp),%edx
12f9: 8b 55 e4        mov     -0x18(%rbp),%rax
12fc: 48 8b 45 e8      mov     %rdi,%r9
1300: 49 89 f9        mov     %esi,%r8d
1303: 41 89 f0        mov     %rax,%rsi
1306: 48 89 c6        mov     $0x12a,%edi
1309: bf 2a 01 00 00   mov     $0x0,%eax
130e: b8 00 00 00 00   mov     %eax,%eax
1313: e8 48 fb ff ff   callq   e60 <syscall@plt>
1318: 89 45 7c        mov     %eax, -0x4(%rbp)
131b: 48 8b 05 b6 2c 20 00 mov     0x202cb6(%rip),%rax # 203
1322: be 00 00 00 00   mov     $0x0,%esi
1327: 48 89 c7        mov     %rax,%rdi
```

## 5.4 Simple practice: Analysis of do\_mem\_access()

### do\_mem\_access(): Introduction

Now, in this section, we performed simple experiment with `do_mem_access()`. Below is the source code of it.

```
// p points to a region that is 1GB (ideally)
void do_mem_access(char* p, int size) {
    int i, j, count, outer, locality;
    int ws_base = 0;
    int max_base = ((size / CACHE_LINE_SIZE) - 512);
    for(outer = 0; outer < (1<<20); ++outer) {
        long r = simplerand() % max_base;
        // Pick a starting offset
        if( opt_random_access ) {
            ws_base = r;
        } else {
            ws_base += 512;
            if( ws_base >= max_base ) {
                ws_base = 0;
            }
        }
    }
}
```



```

    }
}
for(locality = 0; locality < 16; locality++) {
    volatile char *a;
    char c;
    for(i = 0; i < 512; i++) {
        // Working set of 512 cache lines, 32KB
        a = p + ws_base + i * CACHE_LINE_SIZE;
        if((i%8) == 0) {
            *a = 1;
        } else {
            c = *a;
        }
    }
}
}
}
}

```

`do_mem_access()` accepts a dummy buffer of 1GB and, randomly or sequentially access the buffer. In particular, this function consists of two phases. First, find a base address for a working set. Second, from the base address, it iterates over the buffer as much the size of 512 cache lines from the base address; therefore, accessed data would be a member of the working set. The base address can be determined randomly or sequentially based on users' choice. Last, those two phases are iterated 1M time. Through this function, we can compare performances between sequential and random access; of course, there is a possibility that randomly selected working set overlaps with the previously selected working set. In that case, locality would be improved. Experiments related to this comparison is introduced in the next section. Another goal of the function is to check effects of locality. The working set composed by 512 item is traversed 16 times. Considering the cache size is enough to accommodate the working set, the number of cache miss would be kept decreasing. Scalability test of the number of iterations for locality is also introduced in the later section; It is a reasonable assumption that cache miss rates are proportional to the number of iterations.

### **do\_mem\_access(): HW Event and Resource Usage Analysis**

By using `getrusage()` and `perf_event_open()`, I aggregated consumed resources and HW events counts related to TLB and level 1 data cache. There are two objectives on this experiment: **1) Understanding meaning of the each field printed by those function calls** and **2) comparing overall performance results between random and sequential access.**

The table 2 shows some statistics of hardware event counts and resource usages during `do_mem_access()`. I performed the experiment third times and calculated an average. 'Data Cache Miss Rate' and 'Data TLB Miss Rate' are calculated by 'Data Cache Miss / Data Cache Access \* 100' and 'Data TLB Miss / Data TLB Access \* 100' respectively. 'Software Page Faults' is the number of page faults serviced without any I/O activity by reclaiming a page frame from the list of pages awaiting reallocation. Simply say, a soft page fault occurs when the page exists elsewhere in memory: i.e. transitional or prefetched states. 'User CPU Time (s)' is consumed time in user space. 'Involuntary Context Switch' implies how much the process was swapped in order to get CPU. The more time requires to execute the process, the more number of involuntary context switch occurs.

Interestingly, the performance of the sequential access outperformed the random access cases as I expected before. Data cache miss rate of sequential access is improved about 44% from



HW Events/Res Usage	Random Access	Sequential Access
Data Cache Read Access	76314595821	76316093364
Data Cache Write Access	25818038284	25818038316
Data Cache Miss	970218600	417020977
<b>Data Cache Miss Rate (%)</b>	<b>1.27</b>	<b>0.55</b>
Data Cache Prefetch Miss	624559447	23722791
Data TLB Access	76314595821	76316093364
Data TLB Miss	13908489	21793
<b>Data TLB Miss Rate (%)</b>	<b>0.018</b>	<b>2E-05</b>
Software Page Faults	7314	4189
User CPU Time (s)	34	31
Involuntary Context Switch	324	339

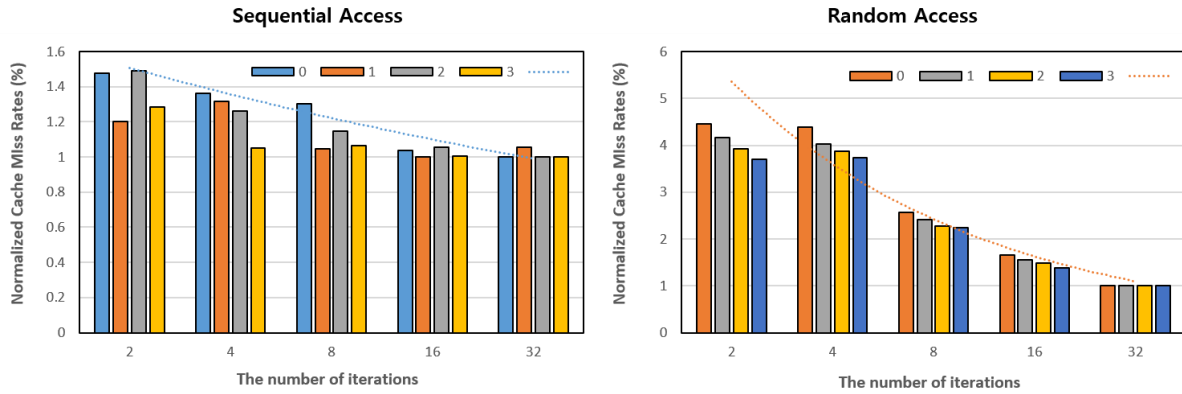
Table 1: `do_mem_access()` resource and event counter statistics

the random access. Data TLB miss rate is also improved extremely. It seems that those improvements had impact on execution time. The sequential access took shorter time than the random access about 3 seconds. I could expect those differences of miss rates and performance are came from **prefetching** data or address, and **out-of-order** execution. Both cases access data cache or TLB evenly for read and write operations, and therefore, upper bound of the number of possible prefetching for data or address should be comparable too. (my laptop does not support prefetch access counter, so I could not strongly argue this statement.) However, the sequential access has lower prefetch miss count than the random access. As a result, it is reasonable enough to think like that prefetching definitely affected TLB/cache miss rates, and consecutively performance, too even though ntel does not support a page prefetching measurement function. We can also apply simple fact in order to give more weight: The sequential access is predictable enough. Expecting next target address is easy when sequentially access, making easy to apply prefetching technique. In the same way, out-of-order execution might be applied to the sequential access ‘better’ than the random access.

### `do_mem_access()`: Scalability Test

This section introduces simple scalability check tests. In the `do_mem_access()`, locality loop iterates 16 times. This tests show what happened when the number of iterations is changed. The reason why I did this experiment is that I wanted to make sure there are no side effects on locality. According to our knowledge, locality must be improved when we access the expected and same address. However, is it true? A computer system might be more complex and there might exist a lot of obstacles. The locality is not a main factor on these experiments, but the secrets of the inner system. From 2 to 32, I increased the number of iterations by 2 and checked the number of cache miss and TLB miss. Below is the graph for it. Note that the upper bounds of y-axis on the two graphs are different since I wanted to show the trends of the cache miss rates. I just scaled based on its own data. Each bars are tagged from ‘0’ to ‘3’. Those are assigned number to each mapping methods on the **section 6.5**. To simplify the graphs and focus on ‘**locality**’ information, and also to consider the scenario, I introduced those graphs in this section. Lastly, I have six combinations of flags but I just used four combinations; I thought that was enough.

Figure 5: Scalability test



## 6 Measuring Memory Access Behavior

### 6.1 Implementation

For convenience reason, I utilized `getopt()` during implementation. Users can perform experiments mentioned on this report by specifying options. Detailed information could be found on the source code help command like below.

```
# ./lab1 -h
```

### 6.2 Analysis using strace

Before exploring memory behaviors, I checked a system call by utilizing **strace**. The **strace** is mainly used to monitor communications between process and Linux kernel including system call. Therefore, by using this tool, I can trace system calls called by the process. I highlighted two important parts. On the first box, the process tries to access `/etc/ld.so.preload` and failed it: it returns -1 and warns that “File does not exist.” `/etc/ld.so.preload` pre-loads other specified shared library including `libc.so`. However, since it is possible that a system does not have this file, a process just tries to access `/etc/ld.so.preload` and if it does not exist on a file system, it just gives up to preload other library.

During the process execution, a system call named **arch\_prctl** was called. `arch_prctl()` sets FS and GS segment registers, while considering architecture. Both FS and GS registers are used by a special purpose on x86\_64 architecture: It means that they don't have a specific obligation. In some cases, kernel uses those registers to points to kernel space stack or thread local area. Therefore, in this case, we can assume that `arch_prctl` is used to prepare calling a system call.

### 6.3 Used flags for mmap()

`mmap()` supports various ways to create a new mapping in virtual address space. This section briefly introduces for each combinations, what characteristics exist. In this lab, I used six combinations of flags. Below list is the explanation about it:

1. **MAP\_SHARED**: Create a mapping which is visible to other processes mapping the same region. Therefore, update this mapping affects other processors too.
2. **MAP\_PRIVATE**: Create a private mapping which is not visible to other processes. Changes to this area are not guaranteed to be reflected to the file.

Figure 6: strace result

```

[hgc@hgc-vm:~/Practice/lab1/final$ strace ./lab1 -p 2 -f 0
execve("/lab1", ["/lab1", "-p", "2", "-f", "0"], 0x7fffd8627a510 /* 22 vars */) = 0
brk(NULL) = 0x564f02139000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=77714, ...}) = 0
mmap(NULL, 77714, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff7f08002000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0\0\0\1\0\0\0\260\34\2\0\0\0\0\0...", 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff7f08000000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff7f079fd000
mprotect(0x7ff7f07be4000, 2097152, PROT_NONE) = 0
mmap(0x7ff7f07de4000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7ff7f07de4000
mmap(0x7ff7f07dea000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff7f07dea000
close(3) = 0
arch_prctl(ARCH_SET_FS, 0x7ff7f08001500) = 0
mprotect(0x7ff7f07de4000, 16384, PROT_READ) = 0
mprotect(0x564f00e2b000, 4096, PROT_READ) = 0
mprotect(0x7ff7f08015000, 4096, PROT_READ) = 0
munmap(0x7ff7f08002000, 77714) = 0
sysinfo({uptime=77439, loads=[0, 1824, 2144], totalram=4122923008, freeram=2721378304, sharedram=15056896, buffer
procs=355, totalhigh=0, freehigh=0, mem_unit=1}) = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 64), ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
brk(NULL) = 0x564f02139000
brk(0x564f0215a000) = 0x564f0215a000
write(1, "Selected problem type: 2\n", 25Selected problem type: 2
) = 25
write(1, "Selected flag type: 0\n", 22Selected flag type: 0
) = 22
sched_setaffinity(0, 128, [0]) = 0

```

3. `MAP_PRIVATE` and `MAP_POPULATE`: Based on `MAP_PRIVATE` flag, prefetch (pre-fault) page tables for a mapping.
4. `MAP_SHARED` and `MAP_ANONYMOUS`: Create a mapping with a zero-filled virtual file and apply `MAP_SHARED`
5. `MAP_PRIVATE` and `MAP_ANONYMOUS`: Create a mapping with a zero-filled virtual file and apply `MAP_PRIVATE`
6. `MAP_PRIVATE` and `MAP_POPULATE` and `MAP_ANONYMOUS`: Create a mapping with a zero-filled and private virtual file and prefetch page tables for a mapping.

Among the above flags, I expected that choosing between anonymous and file backed, and, between populate and no-populate would make a performance or miss rate differences. First, both anonymous and file backed should reserve specified size of memory. However, in case of file backed `mmap()`, it requires to read a file. Second, populating page in advance should affect on TLB miss rates. Experiments related to this assumption were carried away and are introduced in the later section.

## 6.4 msync()

**msync()** is typically used to flush data resided in the memory. In other words, **msync()** moves data to the disk and clears the buffer. I performed a simple experiment to make sure the effect of **msync()** by using file-backed **mmap()**. To do this, I used **tmux** and **gdb** in order to track file changes one line by one line. The left side is about **gdb** and the right side is file data printed by 'cat.' I made breakpoints **mmap():line 527**, **memset():line 540**, and **msync():line 542**. To analyze it more easily, I re-compiled the source code with '-g' flag which generates debugging

information. The initial file consists of only '-' character, but `memset()` tries to set the file as '\*' character. As you can see the second breakpoint, after calling `mmap()`, the file was not changed yet. However, after performing `msync()`, the file contents were changed to '\*' immediately.

Figure 7: GDB for `msync()`

### Set Breakpoints

```
Reading symbols from lab1_gdb...done.
(gdb) b 527
Breakpoint 1 at 0x1be4: file lab1.c, line 527.
(gdb) b 540
Breakpoint 2 at 0x1c89: file lab1.c, line 540.
(gdb) b 542
Breakpoint 3 at 0x1ca6: file lab1.c, line 542.
```

### break: mmap()

```
(gdb) r -p 2 -f 0
Starting program: /home/lhc/aos/kernel/Practice/lab1/part3/lab1_gdb -p 2 -f 0
Selected problem type: 2
Selected flag type: 0
Breakpoint 1, mmap_test_start (n_type=1, mo_type=0, a_type=-1) at lab1.c:527
527     buf = mmap(NULL, (length + offset - pa_offset), \
```

### break: memset()

```
(gdb) c
Continuing.
Breakpoint 2, mmap_test_start (n_type=1, mo_type=0, a_type=-1) at lab1.c:540
540     memset(buf, '-', length);
```

### break: msync()

```
Breakpoint 3, mmap_test_start (n_type=1, mo_type=0, a_type=-1) at lab1.c:542
542     msync(buf, length, MS_SYNC);
```

Next, I did experiments in order to understand impacts on performance of `msync()`. According to my expectation, if we perform `msync()`, soft page fault in file-backed mapping would be decreased since related address translation information would reside in memory before executing `do_mem_access()`. In experiments, cache miss rates percentage with `msync()` was decreased about 0.5% compared to the case without `msync()`.

## 6.5 mmap() Performance and Resource Usage Analysis

### Considerations

This section introduces evaluations and analysis of performance and resource usages, while executing `mmap()` and `do_mem_access()`. There are a lot of options including mapping methods, protection level on the mapping, and memory access methods. However, I do not take all the combinations. Instead, only results that I thought meaningful are introduced. In addition, whenever I needed compare, I tried to distinguish controlled variables to be fixed and manipulated variables to be measured.

First, I used `PROT_READ` and `PROT_WRITE` flags for the protection. Second, only six combinations mentioned on the section 6.3 were used for the experiments. This is because using those options were enough to experience and feel diverse memory behaviors. Third, using either random or sequential access was a good chance to understand the memory behavior. Comparing them is treated as an important part in this section. All the experiments were carried away **third times** and each result numbers is average of them. All the results had **standard deviations lower than 3% of average**. Therefore, I determined that using averages is reasonable. For convenience, I map number from 0 to 5 to each mapping methods, which are used by graphs and tables.

### Comparisons among private, shared and populate flags

First of all, I compared TLB and cache miss rates among shared-, private- and (private- + populate-) results for each file-backed and anonymous mapping methods. In this experiment,

Number	Mapping Methods
0	MAP_SHARED
1	MAP_PRIVATE
2	MAP_PRIVATE and MAP_POPULATE
3	MAP_SHARED and MAP_ANONYMOUS
4	MAP_PRIVATE and MAP_ANONYMOUS
5	MAP_PRIVATE, MAP_POPULATE and MAP_ANONYMOUS

Table 2: Numbering the mapping methods

target memory to be allocated is controlled variable, but mapping method is a manipulated variable. Therefore, we can check performance differences among those three options. To understand trends of cache and TLB miss better, I normalized the data by the lowest value.

Figure 8: Cache miss rates per private/shared/populate

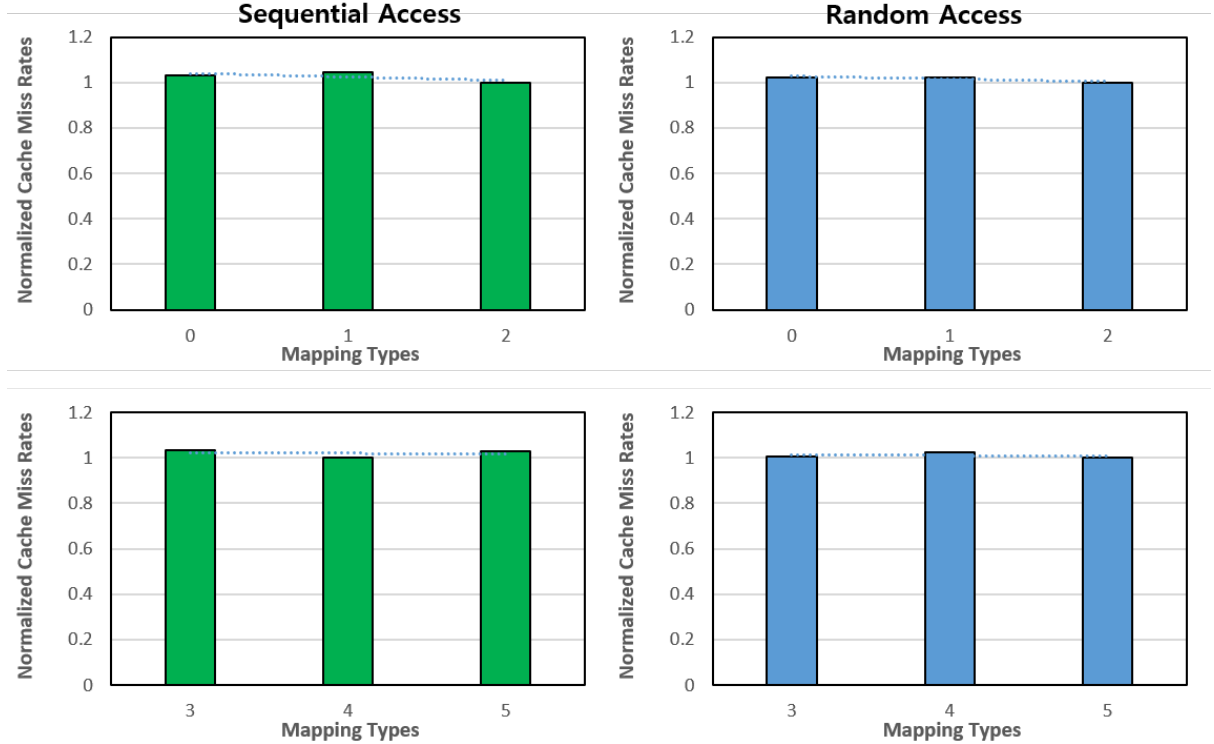
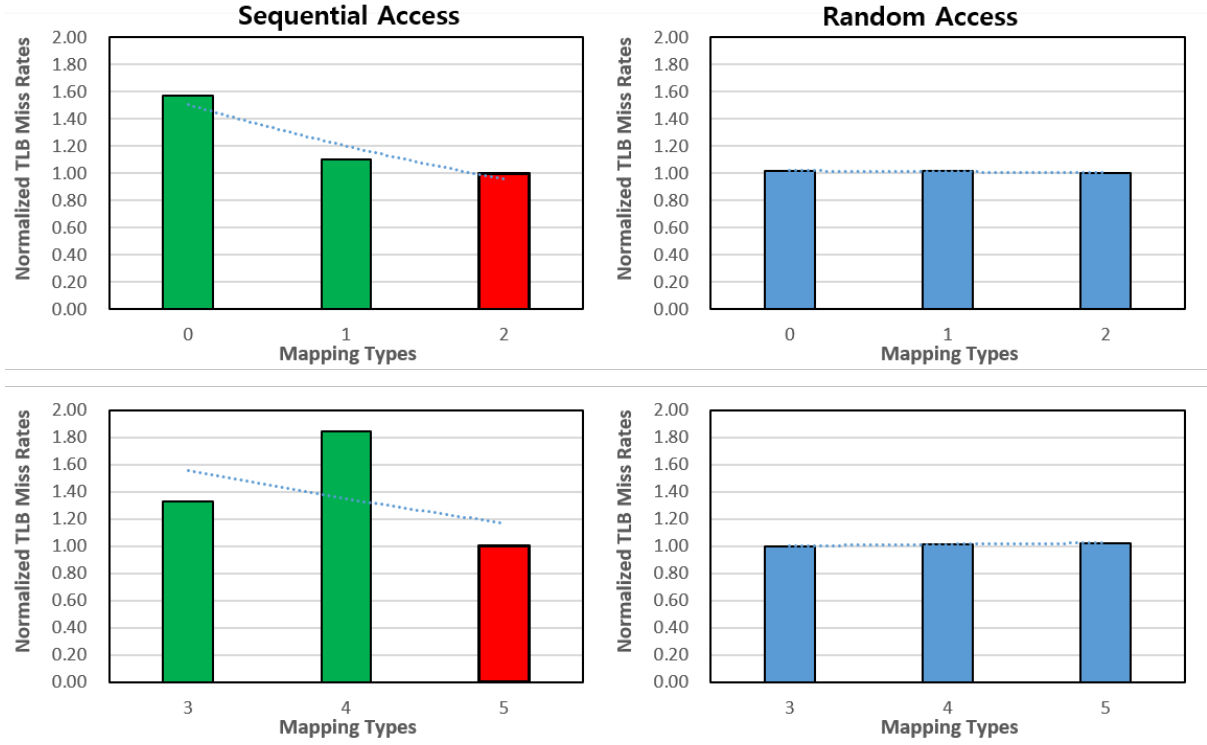


Figure 8 and Figure 9 show cache miss rates and TLB miss rates respectively. There is negligible differences in cache miss rates among 12 cases: sequential 6 cases + random 6 cases. It means that deciding visible or invisible mapping does not have impact on locality. Standard errors of all the cases were lower than 2% of the averages. One minor case that I noticed is that file-backed mapping got higher cache miss rates than anonymous mapping because of `memset()` and `msync()`. Contrary to anonymous mapping which does not need to reflect updates to files, it should write the updated data on disk and made more cache misses.

In Figure 9, MAP\_POPULATE with sequential access shows distinctive result. Prefetching pages worked and reduced the number of TLB miss since the sequential access can be predicted easily. Compared to the worst case of TLB misse rates, it improved percentage about 50% to 70%. On the contrary, with random access, MAP\_POPULATE did not work well since memory

Figure 9: HW TLB miss rates per private/shared/populate



was accessed randomly. It was hard to predict memory address to be prefetched.

In this experiment, I also found interesting resource usage information such as relationship between user time and involuntary context switch or the number of block input operations. But, I deferred this topic to the later evaluation part.

### Comparison between random and sequential access

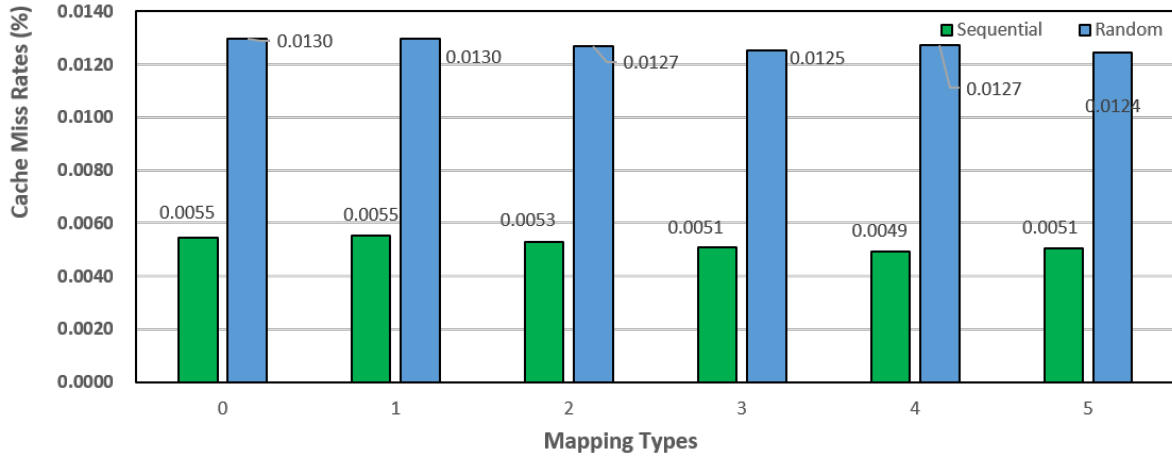
Next question is how much access methods affect on the performance: sequential- and random-memory access. I literally compared sequential and random miss rates of the six flag cases. In this experiment, the controlled variable is the mapping type and the manipulated variable is the both miss rates. All the numbers is actual measured values. Sequential memory access definitely showed better locality performance for both the cache and TLB. Sequential access showed about 61% and about 99% percentage increase on cache miss rates and TLB miss rates on average: Figure 10.

Figure 11 shows the number of L1 data cache prefetch misses of both sequential and random access methods. Random access occurred the prefetch miss from 2 times to 5 times compared to sequential access. Again, since TLB prefetch information is not supported, we cannot strongly argue, but prefetching should affect the performance improvement of both cache and TLB in the sequential access manner.

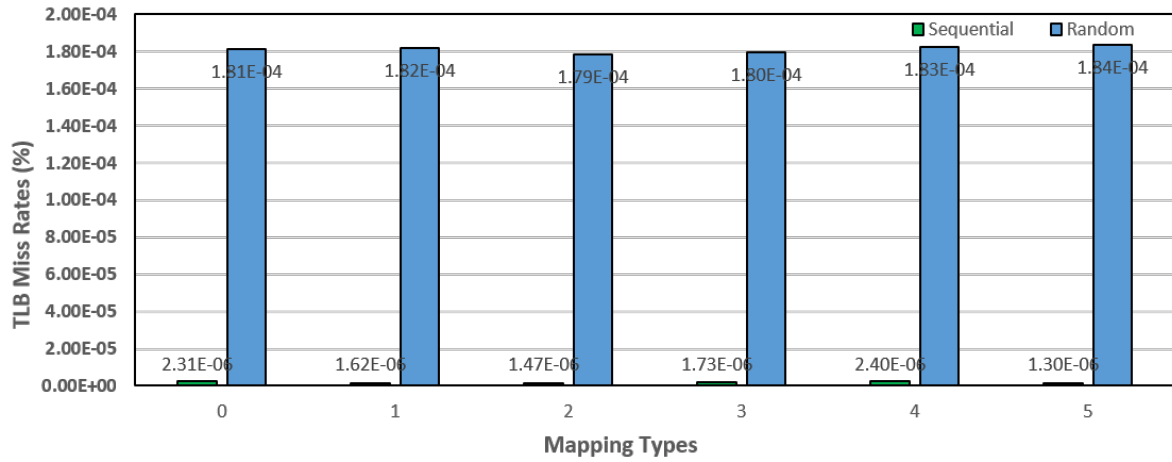
getrusage() did not show any interesting information. Most of the resource usage measurements show similar results on two access methods. Execution time for user space of sequential access is about 32 seconds, which is faster than 34 seconds of random access. I expected that there is a big differences on software page faults or involuntary context switch, but it did not. Only file-backed mapping tried to perform block I/O since anonymous mapping did not require files to create the mapping. By using 'ps' which reports a snapshot of the current processes, I measured CPU and time consumption. During the execution, CPU was consumed 84.5% and

Figure 10: Cache and TLB miss rates comparison between random and sequential access

### Cache Miss Rates



### TLB Miss Rates



memory was consumed 26% on average.

## 6.6 Memory Competition Effect: Background

In this section, I performed a background process by using below commands and compared with the results on the quiet system. For the background process, I used the `compete_for_message()`, which tries to map entire available memory

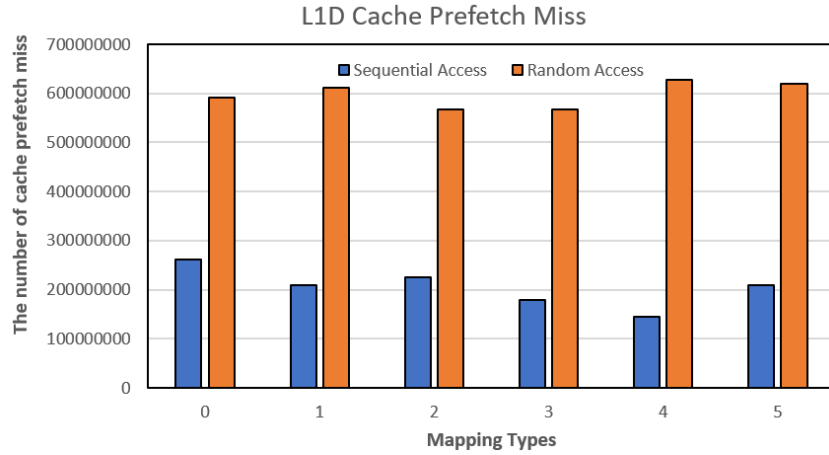
```
# ./backgroundprocess &
```

Since my background process is memory-intensive application, I expected that foreground process would be crashed frequently; it depends on the background application type, but I chose a harsh environment.

However, there were interesting results. At first, most of the tests were crashed during the experiments. Only random access applications could finish their execution successfully. This is because in case of the random access, memory to be mapped was chosen arbitrarily and therefore, there was lower tendency to grab consecutive memory area than sequential access cases; if we want to solve this problem, cache coloring could be the option. Second, TLB miss



Figure 11: L1 Data Cache Prefetch Miss



and cache miss rates were similar with the quiet system's results. However, I concluded that it is unreliable results since miss rates of the quiet system should be better than the environment with background. Therefore, further research is required.

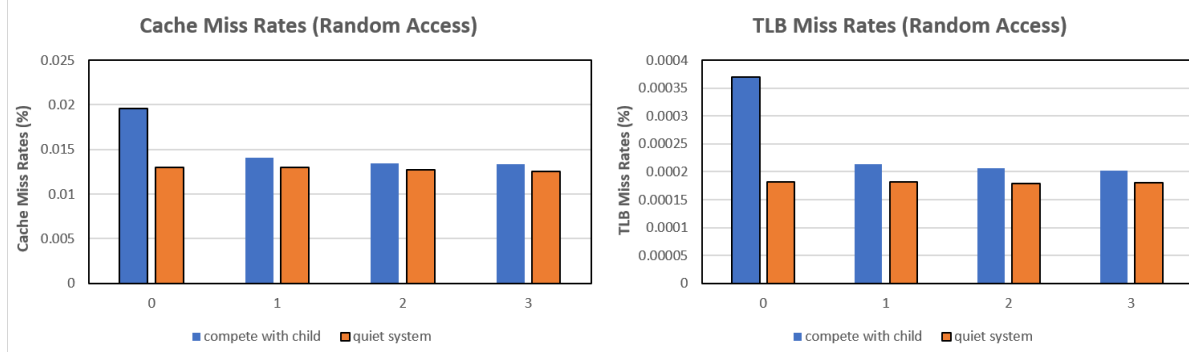
## 6.7 Memory Competition Effect: fork()

I also did experiments that compete for resources with child process. Below is the pseudo code used by this experiment. `compete_for_memory()` tries to grab all the physical DRAM area. To calculate the physical DRAM size, I used `sysconf()`.

```
long get_mem_size() {
    return sysconf(_SC_PAGE_SIZE)*sysconf(_SC_PHYS_PAGES);
}
void main(void) {
    pid = fork();
    if (pid == -1) exit(EXIT_FAILURE);
    else if (pid == 0) // child process
        compete_for_memory();
    else { // parent process
        /* mmap() and do_mem_access() */
        mmap_test();
        /* after parent finishes its job, child process is also killed */
        kill(pid, SIGKILL);
    }
}
```

As background experiments did, most of the cases were crashed during the execution. But, I could get results in some mapping cases of random access. Figure 12 shows the comparisons of TLB and data cache miss rates between the quiet system and the messed system due to the child process. X-axis means the mapping types numbered on the section 6.5. In case of cache miss rate percentage, with the quiet system, the process showed 6% to 34% increases. In case of TLB miss rate percentage, 10% to 50% were increased. Additional interesting information can be found through `getrusage()`. Average CPU user time was increased from 32 second to 58 second and about 20x involuntary context switches occurred with child process.

Figure 12: Competition with Child Process



## 6.8 Kernel Modification Effect

I carried away experiments after modifying kernel source: `mm/vmscan.c:shrink_page_list()`. This function basically attempts to reclaim pages if they are clean or not activated. If a page is detected as 'ACTIVATE', then this page will be locked. However, I modified it as active pages were treated as reclaimable pages. Figure 13 shows the modification. These codes are lines from 1263

Figure 13: Kernel modification

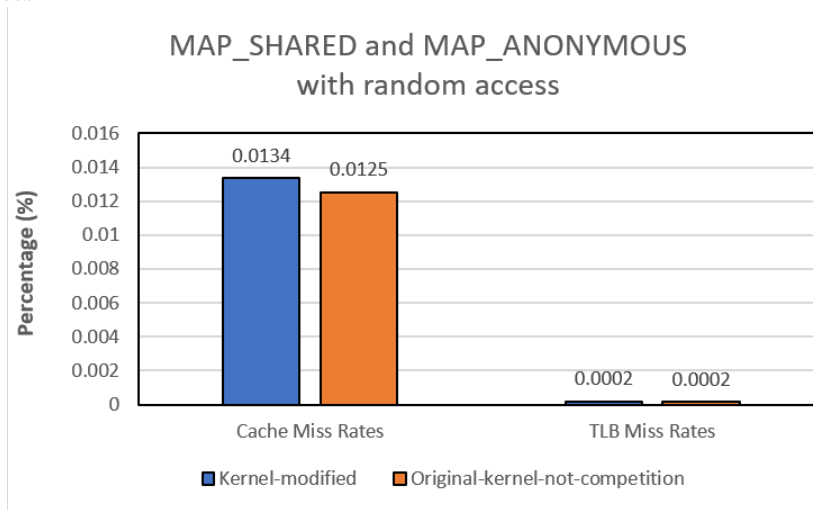
```
if (!force_reclaim)
    references = page_check_references(page, sc);

switch (references) {
//case PAGeref_ACTIVATE:
//    goto activate_locked;
case PAGeref_KEEP:
    nr_ref_keep++;
    goto keep_locked;
case PAGeref_ACTIVATE:
case PAGeref_RECLAIM:
case PAGeref_RECLAIM_CLEAN:
    /* try to reclaim the page below */
}
```

to 1274 of `linux-4.20/mm/vmscan.c:shrink_page_list()`. Instead of jumping to `activate_locked` block, the activated pages are treated as pages to be reclaimed. After compiling this new customized kernel, I carried away the `fork()` experiments mentioned above again. Interestingly, only one process could be survive during the runtime: `MAP_SHARED` and `MAP_ANONYMOUS` with random access method. Unfortunately, since original kernel version's child process experiment was crashed for that case, I could only compare with the quiet system's result.

Figure 13 shows the comparison between customized kernel with child process and quiet kernel without child process. As background experiment did, both performance did not show big differences. There were only 0.001% improvement of the cache miss rates. From this experiment, we can conclude that modified kernel actually did not help to improve both miss rates. This is reasonable since even though forcefully set activated pages to reclaimable pages, it did not mean that there were more resources. During reclaiming process and check conditions on the kernel, those pages should be caught and locked again as the activated pages. We can say the cache miss improvement was occurred because of the child process. But, further analysis and experiments are required.

Figure 14: Performance comparison with kernel-modified-competition and the quiet system without child process



## 6.9 Why fflush() after printf()?

fflush() performs a similar behavior with msync(). This function is typically used to flush output streams. To be specific, it moves the output buffered data to a specified location and clears the buffer: In case of 'stdout' option, it will be printed on the console. If a file descriptor is used, the data will be moved to disk. In our practice, compete\_for\_memory() calls fflush() after printf(). This is because the function is operated by a child process. The child process inherits resources of the parent process including the not flushed output data resided on the buffer. If the child process calls fflush(stdout) after printf(), output data will be printed on the console immediately and correctly. However, fprintf() does not need to call fflush() since its output data is not buffered.

## 7 Time Consuming

I took more than 48 hours seriously. But it was awesome and educational project.