# CS380L: Advanced Operating Systems

## Lab #1

### Program measurement and mmap

The goal of this assignment is to learn about how to measure your program's behavior, and a bit about mmap.

We are interested in doing experimental computer science. We will follow the scientific method, which wikipedia tells me has been around for a long time. But knowing about science in the abstract is relatively easy; actually doing good science is difficult both to learn and to execute.

Let's start with reproducibility. You will write a report for this lab, and in your report you will include details about your system. Think about what it would take to recreate your results. I won't spell out exactly what information you should include, but please include everything relevant while not bogging down your reader. You should report things like the kernel version for your host and guest system.

ON HOST:

tb@tb-os:~$ uname -a

Linux tb-os 4.4.0-59-generic #80-Ubuntu SMP Fri Jan 6 17:47:47 UTC 2017 x86\_64 x86\_64 x86\_64 GNU/Linux

ON GUEST:

tb@tb-os:~$ uname -a

Linux tb-os 4.4.0-59-generic #80-Ubuntu SMP Fri Jan 6 17:47:47 UTC 2017 x86\_64 x86\_64 x86\_64 GNU/Linux

Your report should answer every question in this lab and should do so in a way that is clearly labeled.

This lab requires you to report data. But how you report it is up to you. Please think carefully. I do not want a huge table of numbers for every experiment. You can provide a table, discuss what is and is not relevant, then give me a graph, or a small table, or bold the entries that I should pay attention to (and that you explain in your text).

I have a major pet peeve with excessive digits of precision. Your measurements are usually counts. If you average three counts, don't give me six decimal places of precision even if six digits is the default output format for floats in the language you are using. Decide how many digits are meaningful and then report that many. Also, make your decimal points line up when that makes sense. For example, if you report a mean and a standard deviation, make the decimal places always align so you can see easily if the standard deviation is less than a tenth of the mean (which is a good sign for reproducibility).

I would use C, but you can use whatever programming tools you want. One thing I want you to do both for this class and for real life is always check the return code of every single sytem call you ever make. I know it sounds a bit pedantic, but start the habit now and you will have a happier programming life. For almost every system call all that means is checking if the return code less than zero and if so call perror. When system calls don't work, you really want to know about it early, trust me on this point.

## Tasks

Please include text about each experiment called for in this section, and include any graphs, tables or other data to make your point clearly and concisely. As always, report your experimental platform.

ASUS M4A88TD-M motherboard, a AMD

Athlon II processor, a SAPPHIRE Radeon HD 5770 video card, four Kingston

KVR13N9S8 4GB random access memory (RAM) sticks, and a one terabyte

Hitachi HUA72201 hard drive.

AND MORE...

You should find out what kind of CPU you have

AMD Athlon II X3 445

, its level 1 data cache capacity and associativity

|  |  |
| --- | --- |
|  | 3 x 64 KB 2-way set associative instruction caches 3 x 64 KB 2-way set associative data caches |

3 cores, each with dedicated L1 and L2 caches (no L3 cache)

, and its data TLB capacity and associativity.

1024 4K pages

Does your TLB have a second level? If so, describe it. Put all this information about your hardware in your report.

### Memory map

Your first task will be to write a program that opens, reads, and prints the /proc/self/maps file.

#include <stdio.h>

int main()

{

char c;

FILE \*fptr;

fptr = fopen("/proc/self/maps", "r");

if (fptr) {

while ((c = getc(fptr)) != EOF)

putchar(c);

fclose(fptr);

}

}

tb@tb-os:~/Projects/AdvOSLab1$ ./print\_map

00400000-00401000 r-xp 00000000 08:05 5279519 /home/tb/Projects/AdvOSLab1/print\_map

00600000-00601000 r--p 00000000 08:05 5279519 /home/tb/Projects/AdvOSLab1/print\_map

00601000-00602000 rw-p 00001000 08:05 5279519 /home/tb/Projects/AdvOSLab1/print\_map

00a7b000-00a9c000 rw-p 00000000 00:00 0 [heap]

7f179b477000-7f179b636000 r-xp 00000000 08:05 5775180 /lib/x86\_64-linux-gnu/libc-2.23.so

7f179b636000-7f179b836000 ---p 001bf000 08:05 5775180 /lib/x86\_64-linux-gnu/libc-2.23.so

7f179b836000-7f179b83a000 r--p 001bf000 08:05 5775180 /lib/x86\_64-linux-gnu/libc-2.23.so

7f179b83a000-7f179b83c000 rw-p 001c3000 08:05 5775180 /lib/x86\_64-linux-gnu/libc-2.23.so

7f179b83c000-7f179b840000 rw-p 00000000 00:00 0

7f179b840000-7f179b866000 r-xp 00000000 08:05 5771709 /lib/x86\_64-linux-gnu/ld-2.23.so

7f179ba46000-7f179ba49000 rw-p 00000000 00:00 0

7f179ba63000-7f179ba65000 rw-p 00000000 00:00 0

7f179ba65000-7f179ba66000 r--p 00025000 08:05 5771709 /lib/x86\_64-linux-gnu/ld-2.23.so

7f179ba66000-7f179ba67000 rw-p 00026000 08:05 5771709 /lib/x86\_64-linux-gnu/ld-2.23.so

7f179ba67000-7f179ba68000 rw-p 00000000 00:00 0

7ffcb282c000-7ffcb284d000 rw-p 00000000 00:00 0 [stack]

7ffcb29b1000-7ffcb29b3000 r--p 00000000 00:00 0 [vvar]

7ffcb29b3000-7ffcb29b5000 r-xp 00000000 00:00 0 [vdso]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

Put the result of your program in your report and write a few sentences about what it means. I want you to look at the output, read the documentation to understand its fields,

*/proc/[pid]/maps*

A file containing the currently mapped memory regions and

their access permissions. See [mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html) for some further

information about memory mappings.

*address* *perms offset* *dev* *inode* *pathname*

00400000-00452000 r-xp 00000000 08:02 173521 /usr/bin/dbus-daemon

<http://stackoverflow.com/questions/19938324/what-are-vdso-and-vsyscall>

“

The vsyscall and vDSO segments are two mechanisms used to accelerate certain system calls in Linux. For instance, gettimeofday is usually invoked through this mechanism. The first mechanism introduced was vsyscall, which was added as a way to execute specific system calls which do not need any real level of privilege to run in order to reduce the system call overhead. Following the previous example, all gettimeofday needs to do is to read the kernel's the current time. There are applications that call gettimeofday frequently (e.g to generate timestamps), to the point that they care about even a little bit of overhead. To address this concern, the kernel maps into user space a page containing the current time and a fast gettimeofday implementation (i.e. just a function which reads the time saved into vsyscall). Using this virtual system call, the C library can provide a fast gettimeofday which does not have the overhead introduced by the context switch between kernel space and user space usually introduced by the classic system call model INT 0x80 or SYSCALL.

However, this vsyscall mechanism has some limitations: the memory allocated is small and allows only 4 system calls, and, more important and serious, the vsyscall page is statically allocated to the same address in each process, since the location of the vsyscall page is nailed down in the kernel ABI. This static allocation of the vsyscall compromises the benefit introduced by the memory space randomisation commonly used by Linux. An attacker, after compromising an application by exploiting a stack-overflow, can invoke a system call from the vsyscall page with arbitrary parameters. All he needs is the address of the system call, which is easily predicable as it is statically allocated (if you try to run again your command even with different applications, you'll notice that the address of the vsyscall does not change). It would be nice to remove or at least randomize the location of the vsyscall page to thwart this type of attack. Unfortunately, applications depend on the existence and exact address of that page, so nothing can be done.

This security issue has been addressed by replacing all system call instructions at fixed addresses by a special trap instruction. An application trying to call into the vsyscall page will trap into the kernel, which will then emulate the desired virtual system call in kernel space. The result is a kernel system call emulating a virtual system call which was put there to avoid the kernel system call in the first place. The result is a vsyscall which takes longer to execute but, crucially, does not break the existing ABI. In any case, the slowdown will only be seen if the application is trying to use the vsyscall page instead of the vDSO.

The vDSO offers the same functionality as the vsyscall, while overcoming its limitations. The vDSO (Virtual Dynamically linked Shared Objects) is a memory area allocated in user space which exposes some kernel functionalities at user space in a safe manner. This has been introduced to solve the security threats caused by the vsyscall. The vDSO is dynamically allocated which solves security concerns and can have more than 4 system calls. The vDSO links are provided via the glibc library. The linker will link in the glibc vDSO functionality, provided that such a routine has an accompanying vDSO version, such as gettimeofday. When your program executes, if your kernel does not have vDSO support, a traditional syscall will be made.

“

*/\**

*[2](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L2) \* vvar.h: Shared vDSO/kernel variable declarations*

*[3](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L3) \* Copyright (c) 2011 Andy Lutomirski*

*[4](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L4) \* Subject to the GNU General Public License, version 2*

*[5](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L5) \**

*[6](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L6) \* A handful of variables are accessible (read-only) from userspace*

*[7](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L7) \* code in the vsyscall page and the vdso. They are declared here.*

*[8](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L8) \* Some other file must define them with DEFINE\_VVAR.*

*[9](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L9) \**

*[10](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L10) \* In normal kernel code, they are used like any other variable.*

*[11](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L11) \* In user code, they are accessed through the VVAR macro.*

*[12](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L12) \**

*[13](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L13) \* These variables live in a page of kernel data that has an extra RO*

*[14](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L14) \* mapping for userspace. Each variable needs a unique offset within*

*[15](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L15) \* that page; specify that offset with the DECLARE\_VVAR macro. (If*

*[16](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L16) \* you mess up, the linker will catch it.)*

*[17](http://lxr.free-electrons.com/source/arch/x86/include/asm/vvar.h?v=3.14" \l "L17) \*/*

It looks like we have a few different types of mappings. There are things that are the property of the process. There’s also the stuff that has to do with supporting libc. It looks like libc and ld are both c library resources. There’re pointers to the heap, the stack, and process memory. There’s an address for system calls. Vdso = vDSO (**virtual dynamically linked shared object**) is a Linux kernel mechanism for exporting a carefully selected set of kernel space routines to user space applications so that applications can call these kernel space routines in-process, without incurring the performance penalty of a context switch that is inherent when ...

then think about what is the most interesting thing you found in your output and write a few sentences about what that is.

If the *pathname* field is blank, this is an anonymous mapping

as obtained via [mmap(2)](http://man7.org/linux/man-pages/man2/mmap.2.html). There is no easy way to coordinate

this back to a process's source, short of running it through

[gdb(1)](http://man7.org/linux/man-pages/man1/gdb.1.html), [strace(1)](http://man7.org/linux/man-pages/man1/strace.1.html), or similar.

Also answer this question: Report the base address of your executable (the start of the text section) and the start address of libc. Why are these numbers so different?

Base address of process: 00400000

Base of libc: 7f179b477000

### Getrusage

Next, call getrusage at the end of your program and print out the fields. Pay particular attention to utime, stime, maxrss, minflt, majflt, inblock, oublock, voluntary and involuntary context switches.   
User CPU Seconds: 0

User CPU uSeconds: 0

System CPU Seconds: 0

System CPU uSeconds: 0

Max Resident Set Size: 3012

Integral Shared Mem Size: 0

Integral Unshared Mem Size: 0

Integral Unshared Stack Size: 0

Page Reclaims (Soft Page Faults): 76

Hard Page Faults: 0

Swaps: 0

Block Input Ops: 0

Block Output Ops: 0

IPC Msgs Sent: 0

IPC Msgs Rcvd: 0

Signals Rcvd: 0

Voluntary Context Switches: 0

Involuntary Context Switches: 3

Time measurement in jiffies...

### perf\_event\_open

You will use the perf\_event\_open interface to measure your code. You can do this in a VM or not, but please specify in your report what you have done. Note that you have to configure your kernel to support perf and you might need to build the perf tools yourself if you are booting the latest kernel (as I requested in Lab 0, but which is not 100% necessary for this lab). Read the documentation. It is pretty wild how you call it, right? Does using the syscall routine mean there is a syscall opcode in your program? Check using objdump -d and explain what you find in your report.

400b40: e8 0b fd ff ff callq 400850 <[syscall@plt](mailto:syscall@plt)>

This seems to be a regular function call into a function referenced in the physical lookup table. Syscall man page seems to suggest this function makes the system call, but it’s not technically in my program.

We want you to monitor the following events: level 1 data cache accesses and misses, and data TLB misses. Figuring out how to encode those events into perf\_event\_open is not trivial, so use the force (and the Internet). Note the division of events into read, write and prefetch. According to my experiments, the level 1 data cache experiences many misses due to prefetch, but they do not count prefetch accesses or prefetch data TLB misses (the latter likely because a prefetch that would cause a TLB miss is dropped).

(perf\_hw\_cache\_id) | (perf\_hw\_cache\_op\_id << 8) |

(perf\_hw\_cache\_op\_result\_id << 16)

where *perf\_hw\_cache\_id* is one of:

**PERF\_COUNT\_HW\_CACHE\_L1D**

**PERF\_COUNT\_HW\_CACHE\_DTLB  
and *perf\_hw\_cache\_op\_id* is one of:  
 PERF\_COUNT\_HW\_CACHE\_OP\_READ  
 PERF\_COUNT\_HW\_CACHE\_OP\_WRITE  
 PERF\_COUNT\_HW\_CACHE\_OP\_PREFETCH  
and *perf\_hw\_cache\_op\_result\_id* is one of:  
 PERF\_COUNT\_HW\_CACHE\_RESULT\_ACCESS  
 PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS**

L1 Data Cache Accesses -

3 Codes for 3 types of accesses  
PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_READ << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_ACCESS << 16)

PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_WRITE << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_ACCESS << 16)

PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_PREFETCH << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_ACCESS << 16)

L1 Data Cache Misses

3 Codes for 3 types of accesses  
PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_READ << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16)

PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_WRITE << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16) UNSUPPORTED

PERF\_COUNT\_HW\_CACHE\_L1D | (PERF\_COUNT\_HW\_CACHE\_OP\_PREFETCH << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16)

Data TLB Misses

3 Codes for 3 types of accesses  
PERF\_COUNT\_HW\_CACHE\_DTLB | (PERF\_COUNT\_HW\_CACHE\_OP\_READ << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16)

PERF\_COUNT\_HW\_CACHE\_DTLB | (PERF\_COUNT\_HW\_CACHE\_OP\_WRITE << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16) UNSUPPORTED

PERF\_COUNT\_HW\_CACHE\_DTLB | (PERF\_COUNT\_HW\_CACHE\_OP\_PREFETCH << 8) | (PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS << 16) UNSUPPORTED

Allocate a 1GB buffer, and pass the pointer to this routine (I'm assuming your host and VM have at least 2GB of memory). This routine assumes a global variable called opt\_random\_access. It also assumes your code defines the constant CACHE\_LINE\_SIZE, which is 64 on x86 platforms (and you are allowed to hard code this constant). This code represents the behavior of a program that we wish to study. Briefly summarize its memory access behavior in your report.

// 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;

}

}

}

}

}

This simple random number generator comes from wikipedia. We give it to you so you can observe it using a simple profiler. Note that our code calls this function whether or not it is doing a random traversal. Explain why in your report.

///////////////////////////////////////////////////////////////

// Simple, fast random number generator, here so we can observe it using profiler

long x = 1, y = 4, z = 7, w = 13;

long simplerand(void) {

long t = x;

t ^= t << 11;

t ^= t >> 8;

x = y;

y = z;

z = w;

w ^= w >> 19;

w ^= t;

return w;

}

Enable your tracing events right before calling do\_mem\_access, then disable the events, read the results and report them along with the getrusage results. You should compute and report the cache miss rate and the data TLB miss rate as a percentage (100.0 \* cache misses / cache\_accesses and 100.0 \* tlb misses / cache\_accesses). Also look at the counts themselves.

SEQUENTIAL ACCESS:

Clearing cache

Setting cpu affinitystarting instrumented stuff

ending instrumented stuff

L1 Data Cache Read Accesses: 44172599114

L1 Data Cache Write Accesses: 268514101

L1 Data Cache Prefetch Accesses: 0

L1 Data Cache Read Misses: 12620477

L1 Data Cache Prefetch Misses: 0

DTLB Read Misses: 14110

Cache Miss Rate = 12620477./44172599114 \* 100% = 0.02857082728464598 %

TLB Miss Rate = 14110.0/44172599114 \* 100% = 3.1942879257761394e-05 %

RANDOM ACCESS:

Clearing cache

Setting cpu affinity

starting instrumented stuff

ending instrumented stuff

L1 Data Cache Read Accesses: 44009901750

L1 Data Cache Write Accesses: 270055746

L1 Data Cache Prefetch Accesses: 0

L1 Data Cache Read Misses: 14909907

L1 Data Cache Prefetch Misses: 0

DTLB Read Misses: 411520

Cache Miss Rate = 14909907./44009901750 \* 100% = 0.03387852825642811 %

TLB Miss Rate = 411520./44009901750 \* 100% = 0.0009350623010650097%

To be even more careful about generating repeatable results you should flush the level 1 data cache before enabling the performance counters. You can do this by reading (or writing, if your cache does write allocate) a memory buffer that is larger than your cache size. Also, lock your process onto a single processor and describe the system calls you need to do this in your report.   
  
 printf("Clearing cache\n");

char\* wasted\_space;

wasted\_space = (char \*) malloc(64\*1000\*3);

char x;

for(int i=0; i<64\*1000\*3; i++){

x = wasted\_space[i];

}

// set cpu affinity

printf("Setting cpu affinity\n");

cpu\_set\_t mask;

CPU\_ZERO(&mask); // no cpus

CPU\_SET(0, &mask); // add cpu zero

int result = sched\_setaffinity(0, sizeof(mask), &mask); // apply mask

if (result){

printf("Error setting affinity code %d\n", result);

}

### Measuring memory access behavior

This is the main part of your lab. Perform the following experiments. Allocate your buffer using mmap, both mapping anonymous memory and file-backed memory. Consider sequential and random access (these are properties of the do\_mem\_access code we supply). For file-based mmap consider MAP\_PRIVATE and MAP\_SHARED. Also consider MAP\_POPULATE. Consider the case when you memset the entire buffer after allocating it. Call msync after the memset. What does that do?

To generate file backed:

tb@tb-os:~/Projects/AdvOSLab1$ dd if=/dev/zero of=one\_gb.txt count=1048576 bs=1024

1048576+0 records in

1048576+0 records out

1073741824 bytes (1.1 GB, 1.0 GiB) copied, 2.00902 s, 534 MB/s

Msync flushes back to memory

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Total execution | CPU Utilization | L1 Data Cache Miss rate | DTLB Miss Rate | Soft Page faults |  |
| Anonymous – SEQUENTIAL, MAP\_PRIVATE |  |  |  |  |  |  |
| File backed –  SEQUENTIAL,  MAP\_PRIVATE |  |  |  |  |  |  |
| Anonymous – RANDOM, MAP\_PRIVATE |  |  |  |  |  |  |
| File backed –  RANDOM,  MAP\_PRIVATE |  |  |  |  |  |  |
| File backed – SEQUENTIAL, MAP\_SHARED |  |  |  |  |  |  |
| File backed – RANDOM, MAP\_SHARED |  |  |  |  |  |  |
| File backed – SEQUENTIAL, MAP\_PRIVATE, MAP\_POPULATE |  |  |  |  |  |  |
| File backed – RANDOM, MAP\_PRIVATE, MAP\_POPULATE |  |  |  |  |  |  |
| File backed – SEQUENTIAL, MAP\_PRIVATE, memset |  |  |  |  |  |  |
| File backed – RANDOM, MAP\_PRIVATE, memset |  |  |  |  |  |  |
| File backed – SEQUENTIAL, MAP\_PRIVATE, memset & memsync |  |  |  |  |  |  |
| File backed – RANDOM, MAP\_PRIVATE, memset & memsync |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Notice reliably worse results for Random versus sequential

Test 1 – Anonymous Map Private

Faster because its not file backed

Test 2 – File Backed Map Private

Slower than shared because it’s copy on write?

Test 3 – File Backed Map Shared

Test 4 – File Backed Map Private Map Populate

Read ahead on file

Test 5 – File Backed Map Private Memset

Test 6 – File Backed Map Private Memset Memsync

Remember to examine overall execution time and CPU utilization (often reported by the shell).

To record total execution time: time -p ./print\_map &

To watch cpu utilization: top -p <PID>  
\*Note that you can also get virtual memory and ram utilization of proc from top table  
 \* Soft page faults much higher for file backed than anonymous

Perform all of the above experiments and study the data generated by getrusage and perf\_event\_open. Describe the results in your report using tables, graphs or whatever you think is appropriate. Explain the differences between experiments and why they happen. Determine if your results are stable by running your experiment several times and measuring the standard deviation of your results. Include the results of your analysis in your report.

Running 7 times gives you a standard deviation that you may be 95% confident is within 50% of the true value

Here is an example of the sort of question you should be asking yourself. Why can your data TLB miss count be lower than the total number of data pages your application accesses? Answer this question in your report.

One reason is because of prefetching I think.

Run your program under strace. Enter the output for arch\_prctl in your report and explain what this system call does. Put the output of the system call that involves /etc/ld.so.preload in your report and explain what is going on.

tb@tb-os:~/Projects/AdvOSLab1$ strace ./print\_map 1 0 0 0 0 0

execve("./print\_map", ["./print\_map", "1", "0", "0", "0", "0", "0"], [/\* 62 vars \*/]) = 0

brk(NULL) = 0xd97000

access("/etc/ld.so.nohwcap", F\_OK) = -1 ENOENT (No such file or directory)

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d16c2000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

open("/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=111640, ...}) = 0

mmap(NULL, 111640, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7fa2d16a6000

close(3) = 0

access("/etc/ld.so.nohwcap", F\_OK) = -1 ENOENT (No such file or directory)

open("/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\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1864888, ...}) = 0

mmap(NULL, 3967392, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fa2d10d6000

mprotect(0x7fa2d1295000, 2097152, PROT\_NONE) = 0

mmap(0x7fa2d1495000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1bf000) = 0x7fa2d1495000

mmap(0x7fa2d149b000, 14752, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d149b000

close(3) = 0

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d16a5000

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d16a4000

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d16a3000

arch\_prctl(ARCH\_SET\_FS, 0x7fa2d16a4700) = 0

mprotect(0x7fa2d1495000, 16384, PROT\_READ) = 0

mprotect(0x602000, 4096, PROT\_READ) = 0

mprotect(0x7fa2d16c4000, 4096, PROT\_READ) = 0

munmap(0x7fa2d16a6000, 111640) = 0

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 1), ...}) = 0

brk(NULL) = 0xd97000

brk(0xdb8000) = 0xdb8000

write(1, "--------------------------------"..., 45--------------------------------------------

) = 45

getrusage(RUSAGE\_SELF, {ru\_utime={0, 0}, ru\_stime={0, 0}, ...}) = 0

arch\_prctl(ARCH\_GET\_FS, [0x7fa2d16a4700]) = 0

arch\_prctl(ARCH\_GET\_GS, [0]) = 0

open("/proc/self/maps", O\_RDONLY) = 3

fstat(3, {st\_mode=S\_IFREG|0444, st\_size=0, ...}) = 0

read(3, "00400000-00402000 r-xp 00000000 "..., 1024) = 1024

close(3) = 0

mmap(NULL, 1073741824, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2910d6000

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 3

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 4

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 5

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 6

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 7

perf\_event\_open(0x7ffecb42eb90, 0, -1, -1, 0) = 8

mmap(NULL, 192512, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa2d1674000

sched\_setaffinity(0, 128, [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7ffecb42ee00, 2460d95725043f00, 4019a0, 7fa2d10f6830, 0, 7ffecb42ee08, 700000000, 400e0a, 0, a30c9fb8933db6a0, 400b20, 7ffecb42ee00, 0, 0, 5cf109bd7a1db6a0, 5c493d266fadb6a0, ...]) = 0

ioctl(3, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(4, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(5, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(6, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(7, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(8, PERF\_EVENT\_IOC\_RESET, 0) = 0

ioctl(3, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(4, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(5, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(6, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(7, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(8, PERF\_EVENT\_IOC\_ENABLE, 0) = 0

ioctl(3, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

ioctl(4, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

ioctl(5, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

ioctl(6, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

ioctl(7, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

ioctl(8, PERF\_EVENT\_IOC\_DISABLE, 0) = 0

read(3, "\272V\213F\n\0\0\0", 8) = 8

read(4, "\226\207\371\17\0\0\0\0", 8) = 8

read(5, "\0\0\0\0\0\0\0\0", 8) = 8

read(6, "\206e\227\0\0\0\0\0", 8) = 8

read(7, "\0\0\0\0\0\0\0\0", 8) = 8

read(8, "c\235\0\0\0\0\0\0", 8) = 8

write(1, "L1 Data Cache Read Accesses: 441"..., 41L1 Data Cache Read Accesses: 44133209786

) = 41

write(1, "L1 Data Cache Read Misses: 99219"..., 35L1 Data Cache Read Misses: 9921926

) = 35

write(1, "DTLB Read Misses: 40291\n\n", 25DTLB Read Misses: 40291

) = 25

getrusage(RUSAGE\_SELF, {ru\_utime={59, 440000}, ru\_stime={0, 288000}, ...}) = 0

write(1, "User CPU Seconds: 59\n", 21User CPU Seconds: 59

) = 21

write(1, "User CPU uSeconds: 440000\n", 26User CPU uSeconds: 440000

) = 26

write(1, "System CPU Seconds: 0\n", 22System CPU Seconds: 0

) = 22

write(1, "System CPU uSeconds: 288000\n", 28System CPU uSeconds: 288000

) = 28

write(1, "Page Reclaims (Soft Page Faults)"..., 39Page Reclaims (Soft Page Faults): 1123

) = 39

write(1, "Hard Page Faults: 0\n", 20Hard Page Faults: 0

) = 20

write(1, "Swaps: 0\n", 9Swaps: 0

) = 9

arch\_prctl(ARCH\_GET\_FS, [0x7fa2d16a4700]) = 0

arch\_prctl(ARCH\_GET\_GS, [0]) = 0

munmap(0x7fa2910d6000, 1073741824) = 0

write(1, "--------------------------------"..., 46---------------------------------------------

) = 46

exit\_group(0) = ?

+++ exited with 0 +++

err = arch\_prctl(ARCH\_GET\_FS, (unsigned long) &fs);

if (err){

printf("ERROR ON GET\_FS\n");

}

err = arch\_prctl(ARCH\_GET\_GS, (unsigned long) &gs);

if (err){

printf("ERROR ON GET\_GS\n");

}

printf("arch\_prctl GET\_FS: %lx\n", fs);

printf("arch\_prctl GET\_GS: %lx\n", gs);

linux apparently gives you, in FS and GS, 'thread local storage' and 'processor data area'

TLS is used in some places where ordinary, single-threaded programs would use [global variables](https://en.wikipedia.org/wiki/Global_variables) but where this would be inappropriate in multithreaded cases. An example of such situations is where functions use a global variable to set an error condition (for example the global variable [errno](https://en.wikipedia.org/wiki/Errno) used by many functions of the C library). If [errno](https://en.wikipedia.org/wiki/Errno) were a global variable, a call of a system function on one thread may overwrite the value previously set by a call of a system function on a different thread, possibly before following code on that different thread could check for the error condition.

Processor data area is apparently unused by my processor

You should do your measurements on an otherwise quiet system. However, for extra credit, you can study the effect of background activity. Explain clearly how you generated the background activity, what you expected, and what you found.

For the final section of the lab, we will add code that forks or clones a process that will compete for memory with our process under test. When your main program ends, be sure this competing process also dies. This can be accomplished in many different ways, several of which require less than or equal to three lines of code. For this part, please use the VM you build in Lab 0, configured with at least two virtual CPUs (and check that you have at least two physical CPUs in your host).

Write some code that forks or clones and then calls compete\_for\_memory. You should fork/clone before locking your program under test to a single core. Note that compete\_for\_memory calls get\_mem\_size() which is a function you must provide that returns the size of physical memory in bytes. Because the purpose of this function is to compete for memory with the foreground process, it is ok if this number is not completely equal to the exact amount of RAM in your system, but it should be close. It is NOT acceptable to hard code this number, you must write code to measure this number at runtime.

int compete\_for\_memory(void\* unused) {

long mem\_size = get\_mem\_size();

int page\_sz = sysconf(\_SC\_PAGE\_SIZE);

printf("Total memsize is %3.2f GBs\n", (double)mem\_size/(1024\*1024\*1024));

fflush(stdout);

char\* p = mmap(NULL, mem\_size, PROT\_READ | PROT\_WRITE,

MAP\_NORESERVE|MAP\_PRIVATE|MAP\_ANONYMOUS, -1, (off\_t) 0);

if (p == MAP\_FAILED)

perror("Failed anon MMAP competition");

int i = 0;

while(1) {

volatile char \*a;

long r = simplerand() % (mem\_size/page\_sz);

char c;

if( i >= mem\_size/page\_sz ) {

i = 0;

}

// One read and write per page

//a = p + i \* page\_sz; // sequential access

a = p + r \* page\_sz;

c += \*a;

if((i%8) == 0) {

\*a = 1;

}

i++;

}

return 0;

}

For your report: Why do we call fflush after the printf? Would this fflush be necessary if we fprintf'ed to stderr?

Fflush lets you know when something happened on which process because it gives you a global ordering. Without flushes, your process stores prints into a buffer to be more efficient. Fprintfing to stderr wouldn’t have this problem because stderr is unbuffered.

Now we will mess with the kernel's head (or more precisely its LRU page replacement algorithm). These instructions are approximate since your version of the kernel might be different, but look in mm/vmscan.c in a function likely called shrink\_page\_list. In it, you will see a switch statement with a PAGEREF\_ACTIVATE case, which is the case where the kernel sees the page has been recently accessed. In this case the kernel gotos activate\_locked, but you will change the to fall through, which should be the PAGEREF\_RECLAIM case. Then evaluate what this change does to your test program when it has competition for memory. You can do this by having two identical VMs with different kernels or one kernel that you dynamically configure with its default behavior and your modified behavior, perhaps controlled via the /proc file system.

Notes

Configure your VM with at least two virtual CPUs, but first confirm that your host system has at least two CPUs.

To allocate space in a file, look at the abysmally named ftruncate and fallocate. No matter what you use, call msync after your memset and report your findings.

The perf subsystem uses the performance management unit (PMU) which is a set of hardware counters supported directly by your processor. If you look at dmesg output, it should show the PMU being initialized.

Check for perf availability in your host system before checking the guest. Most of the lab can be done on the host.

If you run perf list on the command line, it will tell you what counters are supported by your combination of hardware, OS and perf tools. You need at least level 1 data cache accesses and misses and data TLB misses. Some of the prefetch varieties may be unsupported.

Historically, VirtualBox has done a poor job of virtualizing the performance counters.

Only so many counters can be active at once (the exactly limit depends on the hardware/OS/perf tools). If you try to add too many you might get errors.

I'm not sure it is necessary, but if you get a lot of variation in your results for the experiments that follow, you might want to disable CPU frequency scaling on your system. I would do this in the BIOS, but you can also try user-level tools like this one that allow you to set the frequency directly (or perhaps the "-g performance" option would work, I'm not sure). Here is a tool. <http://manpages.ubuntu.com/manpages/hardy/man1/cpufreq-selector.1.html>

Your code will have to run with many different configurations. Consider using getopt, or maybe you would prefer a configuration file, but I find command line options superior for this sort of task as they are more explicit and more easily scripted.

Please report how much time you spent on the lab.

#### Your code is subject to visual inspection and should be clean and readable. Points will be deducted if your code is too hard to follow. Your code should follow best practices, for example avoiding arbitrary constants and checking error codes. Seriously, check the return value of every system call you ever make.