CS380L: Advanced Operating Systems Lab #1

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

We use a Linux server for all the experiments. The server has 4 Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz processors and 16GB of memory, and runs Ubuntu 16.04.2 LTS (kernel version 4.11.0). The CPU has 32KB of L1 data cache per core (8-way set associative) (found through `getconf -a | grep CACHE`). In addition, it has two-level TLBs. The first level (data TLB) has 64 entries (4-way set associative), and the second level has 1536 entries for both instructions and data (6-way set associative) (found through `cpuid | grep -i tlb`).

2 Memory map

`/proc/[pid]/maps` file contains process `[pid]`'s mapped memory regions and their access permissions [1]. We use the following code to read content of `/proc/self/maps` file [2]:

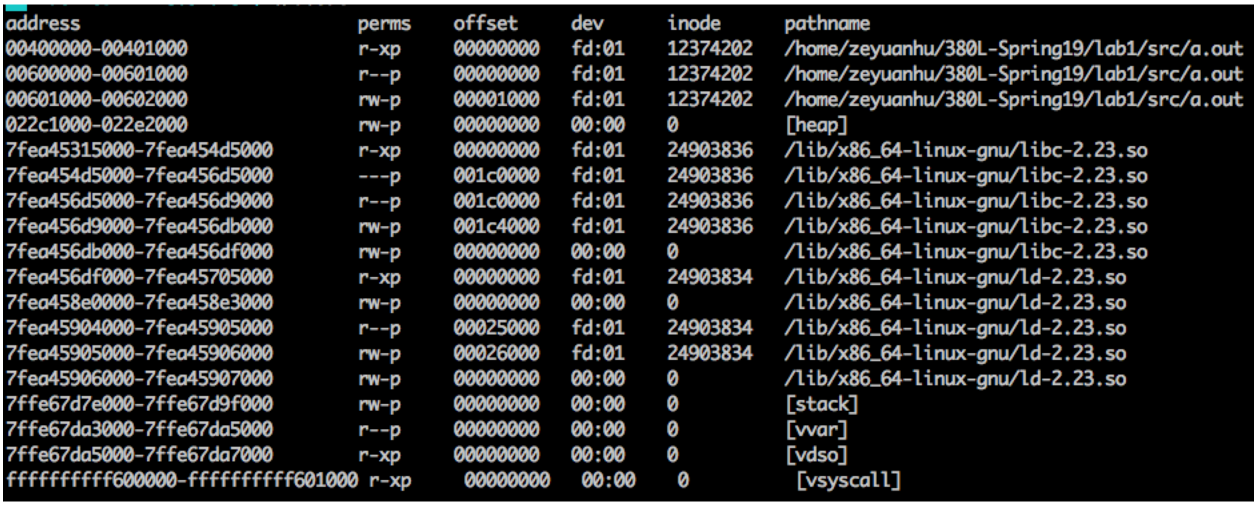
```
sprintf(filepath, "/proc/%u/maps", (unsigned)getpid());
FILE *f = fopen(filepath, "r");

printf("%-32s %-8s %-10s %-8s %-10s %s\n", "address", "perms", "offset", "dev", "
    inode", "pathname");
while (fgets(line, sizeof(line), f) != NULL) {
     sscanf(line, "%s%s%s%s%s%s", address, perms, offset, dev, inode, pathname);
     printf("%-32s %-8s %-10s %-8s %-10s %s\n", address, perms, offset, dev, inode,
          pathname);
}
fclose(f);
```

In the file, each line corresponds to a mapped memory region. There are six columns of each line, which represent six properties of the mapped memory region: `address`, `perms`, `offset`, `dev`, `inode`, and `pathname`. The result of running `memory_map.c` is below:

The `address` field gives range of virtual memory address of the mapped memory region. Access permission of each memory region is indicated by `perms` field. There are four bits in the field: `rwx`

[1] 30 hours spent on this lab.
[2] see `memory_map.c` for complete code



address	perms	offset	dev	inode	pathname
00400000-00401000	r-xp	00000000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
00600000-00601000	r--p	00000000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
00601000-00602000	rw-p	00001000	fd:01	12374202	/home/zeyuanhu/380L-Spring19/lab1/src/a.out
022c1000-022e2000	rw-p	00000000	00:00	0	[heap]
7fea45315000-7fea454d5000	r-xp	00000000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea454d5000-7fea456d5000	---p	001c0000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456d5000-7fea456d9000	r--p	001c0000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456d9000-7fea456db000	rw-p	001c4000	fd:01	24903836	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456db000-7fea456df000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/libc-2.23.so
7fea456df000-7fea45705000	r-xp	00000000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea458e0000-7fea458e3000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45904000-7fea45905000	r--p	00025000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45905000-7fea45906000	rw-p	00026000	fd:01	24903834	/lib/x86_64-linux-gnu/ld-2.23.so
7fea45906000-7fea45907000	rw-p	00000000	00:00	0	/lib/x86_64-linux-gnu/ld-2.23.so
7ffe67d7e000-7ffe67d9f000	rw-p	00000000	00:00	0	[stack]
7ffe67da3000-7ffe67da5000	r--p	00000000	00:00	0	[vvar]
7ffe67da5000-7ffe67da7000	r-xp	00000000	00:00	0	[vdso]
ffffffffff600000-ffffffffff601000	r-xp	00000000	00:00	0	[vsyscall]

Figure 1: Output of memory_map.c

represents read, write, and executable respectively; the last bit (`p` or `s`) represents whether the region is private or shared. `offset` field represents the offset in the mapped file. `dev` field indicates the device (represented with format of `major:minor`) that the mapped file resides . There are two kinds of value in this column for our case: `fd:01` and `00:00`. The former one is the device id (in hex) of `/` (checked with `mountpoint -d /`) and the latter one represents no device associated with the file. `inode` field represents the inode number of the file on the device. `0` means no file is associated with the mapped memory region. `pathname` field gives the absolute path to the file associated with the mapped memory region. It can be some special values like `[heap]`, `[stack]`, `[vdso]`, etc.

To locate the start of the text section of the executable, we invoke `objdump -h` on the binary and get `0000000000400600`. Output of `/proc/self/maps` shows that the start address of `libc` is `7fea45315000`. The reason for these two addresses are different is `libc` is dynamic loaded library, which is loaded during the runtime of executable, which is not compiled and linked as part of executable. The code segment contains the executable instruction, not the dynamic loaded library.

One interesting thing happens between runs of the executable: the content of `/proc/self/maps` is different. Addresses of all mapped memory regions are different except for the regions mapped to the executable and `[vsyscall]`. The root cause behind this phenomenon is Address Space Layout Randomization (ASLR) [2] for programs in user space. This feature is enabled by default and can be seen via the content of `/proc/sys/kernel/randomize_va_space` file. In our case, the value is 2, which means the positions of stack itself, virtual dynamic shared object (VDSO) page, shared memory regions, and data segments are randomized [3].

3 Getrusage

To get resource usage of the current process, we use `getrusage` [4]. The result is stored in `rusage` struct. Not all fields of the struct are completed: unmaintained fields are set to zero by the kernel. Those fields exist for compatibility with other systems purpose. The following code instantiates `rusage` struct and print all the maintained fields [3]:

```c
struct rusage usage;
if (getrusage(RUSAGE_SELF, &usage) != 0) {
    perror("getrusage");
    return 0;
}

// user CPU time used
printf("utime = %ld.%06ld s\n", usage.ru_utime.tv_sec,
usage.ru_utime.tv_usec);
// system CPU time used
printf("stime = %ld.%06ld s\n", usage.ru_stime.tv_sec,
usage.ru_stime.tv_usec);
// maximum resident set size
printf("maxrss = %ld KB\n", usage.ru_maxrss);
// page reclaims (soft page faults)
printf("minflt = %ld\n", usage.ru_minflt);
// page faults (hard page faults)
printf("majflt = %ld\n", usage.ru_majflt);
// block input operations
printf("inblock = %ld\n", usage.ru_inblock);
// block output operations
printf("oublock = %ld\n", usage.ru_oublock);
// voluntary context switches
printf("nvcsw = %ld\n", usage.ru_nvcsw);
// involuntary context switches
printf("nivcsw = %ld\n", usage.ru_nivcsw);
```

man page of `getrusage` explains the meaning of each field [4] in details. `utime` and `stime` are about CPU time usage; `minflt` and `majflt` are related to page faults; `maxrss` represents the maximum size of working set; `inblock` and `oublock` are about file system I/O.

[3]complete code can be seen in `getrusage.c`

```
L1-dcache-load-misses                              [Hardware cache event]
L1-dcache-loads                                    [Hardware cache event]
L1-dcache-stores                                   [Hardware cache event]
L1-icache-load-misses                              [Hardware cache event]
LLC-load-misses                                    [Hardware cache event]
LLC-loads                                          [Hardware cache event]
LLC-store-misses                                   [Hardware cache event]
LLC-stores                                         [Hardware cache event]
branch-load-misses                                 [Hardware cache event]
branch-loads                                       [Hardware cache event]
dTLB-load-misses                                   [Hardware cache event]
dTLB-loads                                         [Hardware cache event]
dTLB-store-misses                                  [Hardware cache event]
dTLB-stores                                        [Hardware cache event]
iTLB-load-misses                                   [Hardware cache event]
iTLB-loads                                         [Hardware cache event]
node-load-misses                                   [Hardware cache event]
node-loads                                         [Hardware cache event]
node-store-misses                                  [Hardware cache event]
node-stores                                        [Hardware cache event]
```

Figure 2: Part of output of `perf list` related to cache

4 perf_event_open

We use the machine specified in the 1, which is a physical server (i.e., not VM). We first check the support of `perf_event_open` interface by checking the existence of `/proc/sys/kernel/perf_event_paranoid` , which is true in our case (value is set to `2`). In Linux, `perf_event_open` interface [5] is used to setup performance monitoring. Specifically, it provides an interface that allows user to access various events (i.e., events counted by performance counters [6]). `perf list` gives available events on current machine. Counters related to cache in our machine is shown in Figure 2. We are interested in counters related to L1 data cache and data TLB. As shown in Figure 2, we have counters for number of times the L1 cache was accessed for data (`L1-dcache-loads`), the number of those access that resulted in a cache miss (`L1-dcache-load-misses`), and write access of L1 data cache (`L1-dcache-stores`). Similarly, for data TLB, we have read access (`dTLB-loads`), read miss (`dTLB-load-misses`), write access (`dTLB-stores`), and write miss (`dTLB-store-misses`).

References

[1] "proc(5) - linux man page." `http://man7.org/linux/man-pages/man5/proc.5.html`.

[2] "Address space layout randomization (aslr)." `https://en.wikipedia.org/wiki/Address_space_layout_randomization`, 2018.

[3] "Linux and aslr: kernel/randomize_va_space." `https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting`, 2016.

[4] "getrusage(2) - linux man page." http://man7.org/linux/man-pages/man2/getrusage.2.html.

[5] "perf_event_open(2) - linux man page." http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[6] "Hardware performance counter." https://en.wikipedia.org/wiki/Hardware_performance_counter, 2018.