CS380L: Advanced Operating Systems Lab #1

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

Unless otherwise noted, 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]:

```
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 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`).

According to man page [5], there is no glibc wrapper for `perf_event_open` system call. However, we can wrap it on our own as the following:

```
static long perf_event_open(struct perf_event_attr *hw_event, pid_t pid, int cpu,
    int group_fd, unsigned long flags) {
  return syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);
}
```

Using the `syscall` does not mean there is a syscall opcode in the program. Figure 3 shows `perf_event_open` section of `objdump -d`. Using `syscall` function will have `perf_event_open` system call number compiled in the program. In this case, the system call number of `perf_event_open` is 298 (can be checked in `arch/x86/entry/syscalls/syscall_64.tbl` of the Linux kernel source tree),

```
0000000000400d36 <perf_event_open>:
  400d36:       55                      push   %rbp
  400d37:       48 89 e5                mov    %rsp,%rbp
  400d3a:       48 83 ec 20             sub    $0x20,%rsp
  400d3e:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
  400d42:       89 75 f4                mov    %esi,-0xc(%rbp)
  400d45:       89 55 f0                mov    %edx,-0x10(%rbp)
  400d48:       89 4d ec                mov    %ecx,-0x14(%rbp)
  400d4b:       4c 89 45 e0             mov    %r8,-0x20(%rbp)
  400d4f:       48 8b 7d e0             mov    -0x20(%rbp),%rdi
  400d53:       8b 75 ec                mov    -0x14(%rbp),%esi
  400d56:       8b 4d f0                mov    -0x10(%rbp),%ecx
  400d59:       8b 55 f4                mov    -0xc(%rbp),%edx
  400d5c:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  400d60:       49 89 f9                mov    %rdi,%r9
  400d63:       41 89 f0                mov    %esi,%r8d
  400d66:       48 89 c6                mov    %rax,%rsi
  400d69:       bf 2a 01 00 00          mov    $0x12a,%edi
  400d6e:       b8 00 00 00 00          mov    $0x0,%eax
  400d73:       e8 c8 fd ff ff          callq  400b40 <syscall@plt>
  400d78:       c9                      leaveq
  400d79:       c3                      retq
```

Figure 3: `perf_event_open` section of `objdump -d` output

which is `$0x12a` in hex.

We want to monitor the following events: read, write, read miss for level 1 data cache and read miss, write miss for data TLB. A call to `perf_event_open` gives a file descriptor that corresponds to one event being measured. We can use `ioctl` interface to control the counters and read file descriptors to obtain counter vales. The following code highlights the usage of `perf_event_open` interface to measure all five events for trivial `printf` [4].

```
int hw_cache_perf_event_open(int group_fd, int cache_id, int cache_op_id,
int cache_op_result_id) {
  struct perf_event_attr pe;
  memset(&pe, 0, sizeof(struct perf_event_attr));
  pe.type = PERF_TYPE_HW_CACHE;
  pe.size = sizeof(struct perf_event_attr);
  pe.config = cache_id | (cache_op_id << 8) | (cache_op_result_id << 16);
  pe.disabled = 0;
  if (group_fd == -1) {
    pe.disabled = 1;
  }
  pe.exclude_kernel = 1;
  pe.exclude_hv = 1;
  int fd = perf_event_open(&pe, 0, cpu_id, group_fd, 0);
  if (fd == -1) {
```

[4] complete code can be seen in `perf_event_open_usage.c`

```c
    perror("perf_event_open");
    exit(EXIT_FAILURE);
  }
  return fd;
}

int main(int argc, char **argv) {

  int l1_read_access_fd = hw_cache_perf_event_open(
      -1, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_ACCESS);
  int leader_fd = l1_read_access_fd;

  int l1_read_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_MISS);
  int l1_write_access_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_L1D, PERF_COUNT_HW_CACHE_OP_WRITE,
      PERF_COUNT_HW_CACHE_RESULT_ACCESS);
  int tlb_read_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_DTLB, PERF_COUNT_HW_CACHE_OP_READ,
      PERF_COUNT_HW_CACHE_RESULT_MISS);
  int tlb_write_miss_fd = hw_cache_perf_event_open(
      leader_fd, PERF_COUNT_HW_CACHE_DTLB, PERF_COUNT_HW_CACHE_OP_WRITE,
      PERF_COUNT_HW_CACHE_RESULT_MISS);

  ioctl(leader_fd, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
  ioctl(leader_fd, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);

  // Do the work that we want to analyze
  printf("Do some work that we want to measure here\n");

  ioctl(leader_fd, PERF_EVENT_IOC_DISABLE, PERF_IOC_FLAG_GROUP);

  uint64_t l1_read_miss = 0;
  uint64_t l1_read_access = 0;
```

```
  uint64_t l1_write_access = 0;
  uint64_t tlb_read_miss = 0;
  uint64_t tlb_write_miss = 0;

  read(l1_read_access_fd, &l1_read_access, sizeof(uint64_t));
  read(l1_read_miss_fd, &l1_read_miss, sizeof(uint64_t));
  read(l1_write_access_fd, &l1_write_access, sizeof(uint64_t));
  read(tlb_read_miss_fd, &tlb_read_miss, sizeof(uint64_t));
  read(tlb_write_miss_fd, &tlb_write_miss, sizeof(uint64_t));

  close(l1_read_access_fd);
  close(l1_read_miss_fd);
  close(l1_write_access_fd);
  close(tlb_read_miss_fd);
  close(tlb_write_miss_fd);

  printf("[Performance counters]\n");
  printf("Data L1 read access: %" PRIu64 "\n", l1_read_access);
  printf("Data L1 write access: %" PRIu64 "\n", l1_write_access);
  printf("Data L1 read miss: %" PRIu64 "\n", l1_read_miss);
  printf("Data L1 read miss rate: %.5f\n",
        (double)l1_read_miss / l1_read_access);
  printf("Data TLB read miss: %" PRIu64 "\n", tlb_read_miss);
  printf("Data TLB write miss: %" PRIu64 "\n", tlb_write_miss);

  fflush(stdout);

  return 0;
}
```

File descriptors returned from calling `perf_event_open` can be grouped together so that we can measure corresponding events simultaneously. There are five events we want to measure and as shown in the code above, we group them together and measure them at the same time. However, measuring all five events at the same time may not possible for some machine as CPU only has limited amount of machine specific registers (MSRs) for low-level performance counting. In our environment, this number is 8 [5]. For the following experiment, we also want to lock our process onto a single processor. To achieve it, we use the following code:

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(cpu_id, &set);
if (sched_setaffinity(0, sizeof(cpu_set_t), &set) == -1) {
        perror("sched_setaffinity");
        exit(EXIT_FAILURE);
}
```

Here, all counters we setup are associated with CPU specified by `cpu_id`. We use `sched_setaffinity` system call to set the CPU affinity of current process and ensure that it runs on CPU of `cpu_id` only.

5 Measuring memory access behavior

`mmap` system call maps files or devices into memory. The created mapping can be file-backed or anonymous. File-backed mapping maps an area of the process's virtual memory to files (i.e., an area of the process's virtual memory is mapped to file-backed memory); reading those areas of memory causes the file read. Anonymous mapping is the opposite of file-backed mapping (i.e., not backed by file; an area of the process's virtual memory is mapped to anonymous memory). In this section, we experiment with both types of mappings. Since file-backed mapping can be shared or private, we have three memory mappings setup in our experiment: anonymous, file-based (private) and file-based (share). In this experiment, we use `MAP_POPULATE` flag. `MAP_POPULATE` populates page tables for a mapping. For file-backed mapping, this means read-ahead on file, which reduce blocking on page faults later. In addition, we use `memset` to initialize the mapped region and `msync` to flush the change made to the file-backed memory back to file system whenever possible.

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.

[5] check via `cpuid|grep 'number of counters per logical processor'`

[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.