CS380L: Advanced Operating Systems Lab #2

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

We have two Linux machines for experiments: `erode-lom` and `thoothukudi-lom`. Both of them run Ubuntu 16.04 LTS. [2]

2 System Tools Exercise

2.1 strace

`script` command allows user to record terminal printout into a file [1]. Per the lab instruction, we use `strace` to trace the syscalls and signals of a target process [2]. In our case, we trace the process involving `cat`. One thing I notice is that `script` contains some unicode as shown in Figure **??**. Thus, we use the following code to clean up the output:

```
cat $FILE | perl -pe 's/\e([^\[\]]|\[.*?[a-zA-Z]|\].*?\a)//g' | col -b > $FILE-
    processed
```

The result is shown in Figure 1 [3].

2.2 lsof

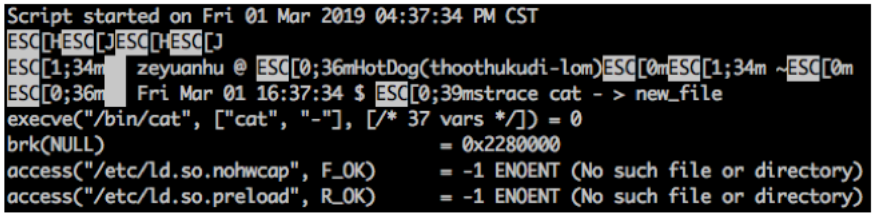
`lsof` lists all open files [3]. `lsof | grep /dev` shows all the open devices used by user-space programs. On our machine, we have the following opened devices:

- `/dev/null`: null device
- `/dev/pts/*` and `/dev/tty`: terminal devices
- `/dev/urandom`: kernel random number source device
- `/dev/ptmx`: a character file to create a pseudoterminal master

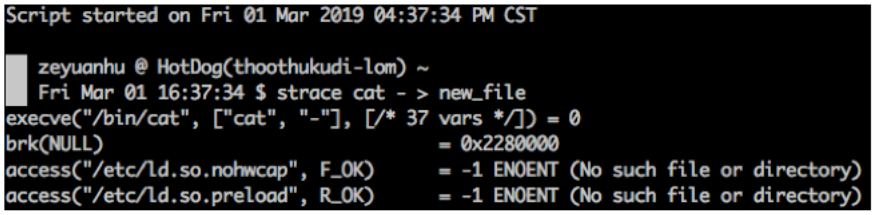
[1] 30 hours spent on this lab.

[2] `erode-lom` with ip address: `192.168.1.120` and `thoothukudi-lom` with ip address: `192.168.1.121`. Due to the network setup, those two machines are not properly registered in the department DNS server. We use ip address instead host names to logon these two machines.

[3] raw output and cleanup output comes with the report as `session_record` and `session_record-processed` respectively



Raw output of script, which contains Unicode character.



Script output after cleanup

Figure 1: `script` output before and after cleanup

3 Network Tools

`ifconfig` command lists all the network interfaces the machine is using to communicate externally. On our machine, interface for Ethernet is `eno1`. We can find IP address, gateway address, and subnet mask from the output.

`tcpdump` command can dump traffic on a network interface. We use the `tcpdump` output provided by the lab instruction to answer the questions below.

a. Are DHCP messages sent over UDP or TCP?

We use `tcpdump -nn -r tcpdump.out.1 | grep -i dhcp` to filter out the DHCP messages from the dump. `-nn` ensures that we can see the actual port number instead of the port name. One line of the output is:

```
10:19:24.525962 IP 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from
    a8:20:66:3b:66:51, length 300
```

The first field shows the time that the packet was traveling. The second field shows the source host address and port, followed by the destination host address and port. The third field shows the protocl the packet was using. From DHCP [4], we know DHCP messages sent over UDP. As shown by the printout, the messages are sent between port 68 (client) and port 67 (server).

b. What is the link-layer (e.g., Ethernet) address of your host? (Feel free to obscure the last couple bytes for privacy's sake)

We use the same `tcpdump` command as above with extra `-e` option to show linke-layer header. The following printout contains DHCP messages for acquiring IP address:

```
10:19:24.525962 a8:20:66:3b:66:51 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800
    ), length 342: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from
    a8:20:66:3b:66:51, length 300
10:19:24.566258 00:21:9b:fb:61:0c > a8:20:66:3b:66:51, ethertype IPv4 (0x0800
    ), length 342: 128.83.158.2.67 > 128.83.158.160.68: BOOTP/DHCP, Reply,
    length 30
```

From the printout we can see that the link-layer address (MAC address) of the host is `a8:20:66:3b:66:51`.

c. What is the IP address of your DHCP server?

From the printout above, we can see the IP address of DHCP server is `128.83.158.2` and the new IP address acquired from DHCP server is `128.83.158.160`.

d. What is the purpose of the DHCP release message?

DHCP release message is used to release IP address.

e. Does the DHCP server issue an acknowledgment of receipt of the client's DHCP request?

DHCP server does not issue an acknowledgment of receipt of the client?s release message.

f. What would happen if the client's DHCP release message is lost?

If DHCP release message is lost, the DHCP server has to wait for the lease to timeout before assigning it to other clients.

4 User-level File System

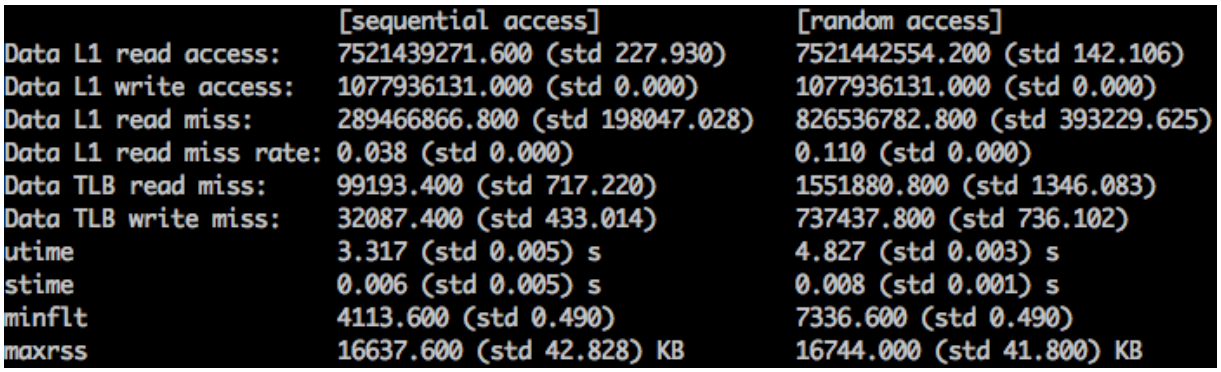
5 Measuring memory access behavior

`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.

We study the following code, which is used to access the mapped region:

```
#define CACHE_LINE_SIZE 64
int opt_random_access;
void do_mem_access(char *p, int size) {
  int outer, locality, i;
  int ws_base = 0;
  int max_base = size / CACHE_LINE_SIZE - 512;
  for (outer = 0; outer < (1 << 20); outer++) {
    long r = simplerand() % max_base;
    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++) {
        a = p + ws_base + i * CACHE_LINE_SIZE;
        if (i % 8 == 0) {
          *a = 1;
        } else {
          c = *a;
        }
      }
    }
  }
}
```



	[sequential access]	[random access]
Data L1 read access:	7521439271.600 (std 227.930)	7521442554.200 (std 142.106)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	289466866.800 (std 198047.028)	826536782.800 (std 393229.625)
Data L1 read miss rate:	0.038 (std 0.000)	0.110 (std 0.000)
Data TLB read miss:	99193.400 (std 717.220)	1551880.800 (std 1346.083)
Data TLB write miss:	32087.400 (std 433.014)	737437.800 (std 736.102)
utime	3.317 (std 0.005) s	4.827 (std 0.003) s
stime	0.006 (std 0.005) s	0.008 (std 0.001) s
minflt	4113.600 (std 0.490)	7336.600 (std 0.490)
maxrss	16637.600 (std 42.828) KB	16744.000 (std 41.800) KB

Figure 2: The statistics of sequential and random memory accesses with anonymous mapped memory regions

`simplerand()` is a function generates random number. A global variable `opt_random_access` is used to specify the memory access to be sequential or random. Specifically, `db_mem_access()` routine will access a working set consists of 512 consecutive cache lines. Among those 512 cache lines, $\frac{1}{8}$ will be write access and $\frac{7}{8}$ will be read access. The same working set is accessed 16 times. The whole routine will work on 2^{20} working sets. Working sets are selected either sequentially or in random (specified by `opt_random_access`). In sequential access case, offset between working sets is 512 (e.g., since there are 512 cache lines of a set). In random access case, the start(i.e., base) of working set is selected randomly based on the number generated by `simplerand()`. `simplerand()` is called no matter the sequential memory access or random access because we want to keep the experiment results difference only due to the memory access pattern difference. In other words, we want to have both memory access patterns have the same function call and context switch overheads (caused by calling `simplerand()`).

We study `db_mem_access()` behavior on a VM with 2GB of memory and 4 virtual CPU cores. The VM runs Ubuntu 18.10 with kernel version 4.19.6. `-cpu host` is added as part of QEMU option to ensure our VM can access performance counters. We allocate 1GB buffer via `mmap` as input for `do_mem_access`. To ensure the repeatable result, we flush the level 1 data cache by accessing a 16MB buffer before calling `do_mem_access`. We invoke `getrusage` just before and after `do_mem_access` to collect resource usage statistics of initialization code and `do_mem_access`. We run the program 5 times and calculate mean and stand deviation of statistics [4]. We also submit the raw results along with this report.

We first compare sequential access and random access (indicated by `opt_random_access`). We use anonymous mapped memory regions. Figure 2 shows statistics for both cases [5]. As shown in the figure, the ratio between read and write access is 7:1 for data L1 cache for both cases, which is as expected. The read miss rate in random access case is 3x higher than sequential access. For

[4] We compile our program with `-O2` to prevent any unwanted memory access from the generated executable

[5] `majflt`, `inblock`, `oublock` are omitted because they are all 0

	[private, seq]	[private,rand]	[share, seq]	[share, rand]
maxrss (init)	17540.800 (std 17.046) KB	17545.600 (std 18.693) KB	17550.400 (std 23.269) KB	17516.800 (std 15.052) KB
inblock (init)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (init)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
maxrss (fun)	16658.400 (std 45.789) KB	16816.000 (std 52.642) KB	16630.400 (std 45.438) KB	16839.200 (std 19.823) KB
inblock (fun)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (fun)	0.000 (std 0.000)	0.000 (std 0.000)	6566.400 (std 13132.800)	6566.400 (std 13132.800)

Figure 3: The statistics of sequential and random memory accesses with file-backed mapping with `MAP_PRIVATE` and `MAP_SHARED`. `fun` is a shorthand for `do_mem_access`.

data TLB, read miss and write miss of random access is 15.6X and 23x higher than sequential access respectively. Random access runs 1.46x slower than sequential access. From those statistics, we can see that there is no significant difference in terms of runtime and miss rate of data L1 cache between random access and sequential access (e.g., miss rate for random access is around 0.1). In the case of sequential access, the number of data TLB read miss is about 99193, which is 396MB for 4KB page size. 396MB is smaller than the size of our buffer. If we take a look at the code, we notice `max_base` in `do_mem_access` restricts the maximum base address of the working set. If we take a look at how `max_base` is initialized and the line we initialize `a`, we can see that the working set is essentially the first $\frac{1}{64}$ of the given buffer, which is 16MB, which matches with the value of `maxrss` we observed and is much lower than 1GB. Realizing this fact also explains why TLB miss count is lower than the total number of data pages [6].

In the next step, we experiment with file-based memory mappings. In this part, we only concern `maxrss`, `inblock`, and `oublock` because the statistics related to data L1 cache and data TLB are generated with the same setup as anonymous mappings. In addition, statistics related to memory cache is not affected by the file-backed mapping. The statistics we have is shown in Figure 3. Compare to previous result, we notice there are large amount of `oublock` during `do_mem_access` for file-backed mapping with `MAP_SHARED`. This is intuitive as the changes to the file have to be saved in shared mode. We notice that there is `inblock` is zero, which is due to the internal cache in Linux kernel for reading file. `oublock` for both random access and sequential access under shared mode are exactly the same. This may due to the fact that access pattern is not significant for the shared mode as synchronization for internal file cache is needed no matter what memory access pattern we may encounter.

In this experiment, we further examine `MAP_POPULATE` flag and `memset` the mapped memory for initialization. Figure 4 shows the statistics. Note that all the columns are from the sequential memory access. With `MAP_POPULATE` flag, the mapped memory will be populate page tables at the very beginning and perform read-ahead on the file. Thus we see the `maxrss` will match the allocated buffer size, which is around 1GB during the initialization. If the mapped memory is initialized with `memset` and then perform synchronization to the file system via `msync`, then there will be large

[6] If we want to access all the allocated buffer (1GB), we should replace line 21 (`a` initialization) with `a = p + ( ws_base + i)* CACHE_LINE_SIZE;`

	[private, pop]	[share, pop]	[share, memset]
maxrss (init)	1066060.000 (std 106.883) KB	1064109.600 (std 274.140) KB	1066129.600 (std 31.455) KB
inblock (init)	509718.400 (std 224734.764)	80596.800 (std 161193.600)	0.000 (std 0.000)
oublock (init)	0.000 (std 0.000)	0.000 (std 0.000)	2064320.000 (std 0.000)
maxrss (fun)	0.000 (std 0.000)	2334.400 (std 259.200)	320.800 (std 34.260)
inblock (fun)	99.200 (std 198.400)	0.000 (std 0.000)	0.000 (std 0.000)
oublock (fun)	0.000 (std 0.000)	6566.400 (std 13132.800)	32832.000 (std 0.000)

Figure 4: The statistics of file-backed mapping for `MAP_POPULATE` and memset after buffer allocation. `fun` is a shorthand for `do_mem_access`.

amount of `oublock` during the initialization.

We also using `strace` to trace our program [7]. We find a system call `arch_prctl(ARCH_SET_FS , 0x7f83a006d540)` in the printout. The system call means setting FS register to `0x7f83a006d540` [6]. FS register is a segment register with no specific processor-defined purpose and is given purpose by the OS. In Linux, FS register is for thread local storage. We can also see the system call `access ("/etc/ld.so.preload", R_OK)`, which tries to access `/etc/ld.so.preload` file. This file contains ELF shared objects, which are loaded before the execution of the program [7]. From the printout, we can see that this file is accessed before `libc.so` is loaded.

6 Measuring memory access behavior with background activity

In this section, we study the memory access behavior when there is a background activity. We use another process to allocate and access a large amount of memory to simulate background activity. We use `compete_for_memory` function for this purpose.

```
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, 0);
      if (p == MAP_FAILED) {
        perror(``Failed anon MMAP competition'');
        fflush(stdout);
        exit(EXIT_FAILURE);
```

[7] we trace our program as `strace ./memory_access 0 0 0 0 0 0`, where `memory_access` is the executable from `memory_access.c`

```
13      }
14
15      int i = 0;
16      while (1) {
17        volatile char *a;
18        long r = simplerand() % (mem_size / page_sz);
19        char c;
20        if (i >= mem_size / page_sz) {
21          i = 0;
22        }
23        a = p + r * page_sz;
24        c += *a;
25        if (i % 8 == 0) {
26          *a = 1;
27        }
28        i++;
29      }
30      return 0;
31  }
```

`get_mem_size` is used to obtain the total amount of RAM of current system in bytes by reading from `/proc/meminfo` file. In the function above, `fflush(stdout)` is invoked after printing total memory. This is necessary because otherwise we cannot see the total memory message. However, flush to stderr is not necessary because each print call to stderr will be automatically followed by a flush. Since `compete_for_memory` function will try to allocate a memory region equals to the size of RAM, we need to setup swap file for the system because otherwise, the process will be killed due to oversubscribe of the memory. In the experiment, we setup a swap file of 2GB in the VM. `compete_for_memory` is called by a separate child process via `fork`. The child process will be killed when the main process exits. After forking the child process, the main process will sleep for 5 seconds to wait for child process to allocate memory. The statistics is shown in Figure 5. Note that the statistics is for anonymous mapping with random memory access. For other mapping and access behaviors, we submit the raw data along with the report. With memory pressure in the system, we expect the program takes more time to finish and encounters more page faults. From the statistics, we see that the program with memory pressure takes more time to finish and page faults are significantly higher than the no-compete case (shown both from `minflt` and `majflt`). However, read miss for both L1 cache and data TLB in compete case is slightly higher than the

no-compete case. We also see there are non-zero values of `inblock` in the case of memory pressure. This is another indicator that more page faults happened. Other mappings (file-backed in shared and private mode) has similar statistics with slight higher page faults.

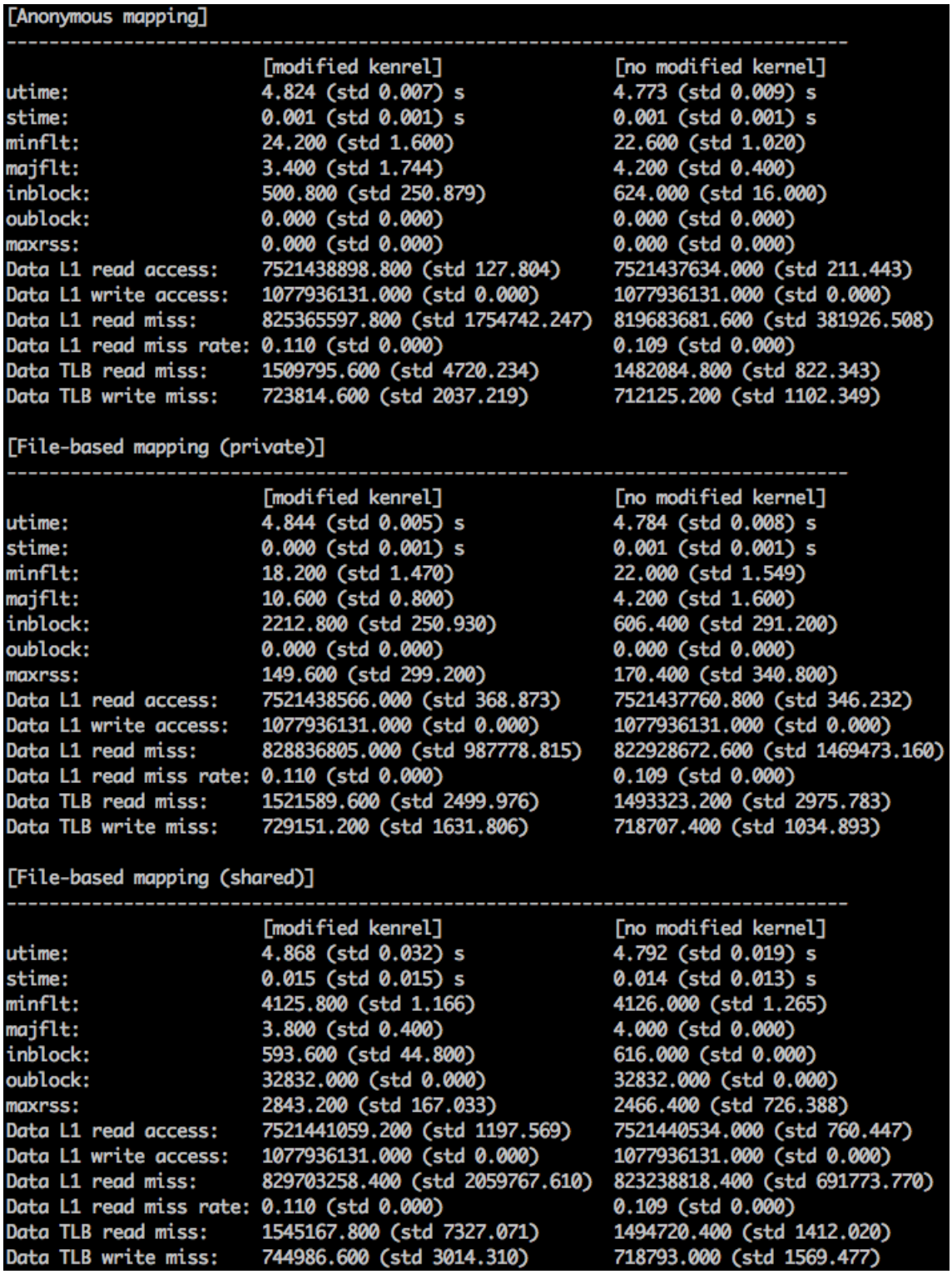
Next, we modify the page replacement policy of Linux kernel. Linux kernel uses LRU to reclaim pages. We modify `shrink_page_list` function in `mm/vmscan.c` file. The target code block looks like below:

```
switch (references) {
case PAGEREF_ACTIVATE:
                goto activate_locked;
case PAGEREF_KEEP:
                nr_ref_keep++;
                goto keep_locked;
case PAGEREF_RECLAIM:
case PAGEREF_RECLAIM_CLEAN:
                 ; /* try to reclaim the page below */
}
```

We modify the `PAGEREF_ACTIVATE` case to behave the same as `PAGEREF_RECLAIM` case. Before the modification, `PAGEREF_ACTIVATE` represents the case where the kernel sees the page has been recently accessed. Thus, the page will not be swapped out of RAM. If we change this case to behave the same as `PAGEREF_RECLAIM`, there is no LRU in the page replacement process. We conduct our experiment for memory random access under memory pressure for all possible mappings. The statistics is shown in Figure 6. As one can see, since we disable the LRU replacement policy, the page faults increase in the modified kernel case. This is intuitive as the kernel might swap out pages that are frequently accessed. However, we should note that the page faults do not increase dramatically as in theory LRU can behave worse than the no replacement policy at all especially when the memory access

	[with compete_for_memory]	[no compete_for_memory]
utime:	6.811 (std 0.164) s	4.790 (std 0.047) s
stime:	0.101 (std 0.130) s	0.000 (std 0.001) s
minflt:	19.800 (std 3.059)	10.200 (std 0.980)
majflt:	16.200 (std 2.482)	0.000 (std 0.000)
inblock:	1737.600 (std 734.998)	0.000 (std 0.000)
oublock:	0.000 (std 0.000)	0.000 (std 0.000)
maxrss:	674.400 (std 121.342)	0.000 (std 0.000)
Data L1 read access:	7521457650.400 (std 3348.175)	7521437569.600 (std 343.389)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	849878078.000 (std 3046793.999)	816180244.600 (std 1026125.094)
Data L1 read miss rate:	0.113 (std 0.000)	0.109 (std 0.000)
Data TLB read miss:	1660888.600 (std 24995.351)	1495066.600 (std 3024.350)
Data TLB write miss:	779031.000 (std 11851.924)	715924.000 (std 1492.899)

Figure 5: The statistics of anonymous mapping for random memory access both with and without memory pressure



[Anonymous mapping]

	[modified kenrel]	[no modified kernel]
utime:	4.824 (std 0.007) s	4.773 (std 0.009) s
stime:	0.001 (std 0.001) s	0.001 (std 0.001) s
minflt:	24.200 (std 1.600)	22.600 (std 1.020)
majflt:	3.400 (std 1.744)	4.200 (std 0.400)
inblock:	500.800 (std 250.879)	624.000 (std 16.000)
oublock:	0.000 (std 0.000)	0.000 (std 0.000)
maxrss:	0.000 (std 0.000)	0.000 (std 0.000)
Data L1 read access:	7521438898.800 (std 127.804)	7521437634.000 (std 211.443)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	825365597.800 (std 1754742.247)	819683681.600 (std 381926.508)
Data L1 read miss rate:	0.110 (std 0.000)	0.109 (std 0.000)
Data TLB read miss:	1509795.600 (std 4720.234)	1482084.800 (std 822.343)
Data TLB write miss:	723814.600 (std 2037.219)	712125.200 (std 1102.349)

[File-based mapping (private)]

	[modified kenrel]	[no modified kernel]
utime:	4.844 (std 0.005) s	4.784 (std 0.008) s
stime:	0.000 (std 0.001) s	0.001 (std 0.001) s
minflt:	18.200 (std 1.470)	22.000 (std 1.549)
majflt:	10.600 (std 0.800)	4.200 (std 1.600)
inblock:	2212.800 (std 250.930)	606.400 (std 291.200)
oublock:	0.000 (std 0.000)	0.000 (std 0.000)
maxrss:	149.600 (std 299.200)	170.400 (std 340.800)
Data L1 read access:	7521438566.000 (std 368.873)	7521437760.800 (std 346.232)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	828836805.000 (std 987778.815)	822928672.600 (std 1469473.160)
Data L1 read miss rate:	0.110 (std 0.000)	0.109 (std 0.000)
Data TLB read miss:	1521589.600 (std 2499.976)	1493323.200 (std 2975.783)
Data TLB write miss:	729151.200 (std 1631.806)	718707.400 (std 1034.893)

[File-based mapping (shared)]

	[modified kenrel]	[no modified kernel]
utime:	4.868 (std 0.032) s	4.792 (std 0.019) s
stime:	0.015 (std 0.015) s	0.014 (std 0.013) s
minflt:	4125.800 (std 1.166)	4126.000 (std 1.265)
majflt:	3.800 (std 0.400)	4.000 (std 0.000)
inblock:	593.600 (std 44.800)	616.000 (std 0.000)
oublock:	32832.000 (std 0.000)	32832.000 (std 0.000)
maxrss:	2843.200 (std 167.033)	2466.400 (std 726.388)
Data L1 read access:	7521441059.200 (std 1197.569)	7521440534.000 (std 760.447)
Data L1 write access:	1077936131.000 (std 0.000)	1077936131.000 (std 0.000)
Data L1 read miss:	829703258.400 (std 2059767.610)	823238818.400 (std 691773.770)
Data L1 read miss rate:	0.110 (std 0.000)	0.109 (std 0.000)
Data TLB read miss:	1545167.800 (std 7327.071)	1494720.400 (std 1412.020)
Data TLB write miss:	744986.600 (std 3014.310)	718793.000 (std 1569.477)

Figure 6: The statistics of memory random access behavior under different mappings before and after the modification of the kernel

pattern is random. This is because random memory access may create some access pattern that is in favor of no replacement policy at all case. For example, there is a clear page faults increase for file-backed mapping in private mode. However, for anonymous mapping, page faults actually decrease after the modification. In addition, since the `do_mem_access` frequently accesses around 16MB of memory, while the competing process uniformly accesses its 2GB memory, the chance that a hot page being swapped out is relative small.

References

[1] "script(1) - linux man page." `http://man7.org/linux/man-pages/man1/script.1.html`.

[2] "strace(1) - linux man page." `https://linux.die.net/man/1/strace`.

[3] "lsof(8) - linux man page." `http://man7.org/linux/man-pages/man8/lsof.8.html`.

[4] "Dynamic host configuration protocol." `https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol`, 2018.

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

[6] "arch_prctl(2) - linux man page." `http://man7.org/linux/man-pages/man2/arch_prctl.2.html`.

[7] "ld.so(8) - linux man page." `http://man7.org/linux/man-pages/man8/ld.so.8.html`.