

# H1 CS353 Linux内核 大作业报告

517030910214 刘宏洲

## H2 0. 简介

在本作业中，我们实现了进程页面热度的收集与展示。主要实现了一个模块，用于过滤指定进程的数据页面、采集热度信息，并且实现了热度信息在内核态与用户态之间的共享。

本大作业的系统环境配置如下：

- Ubuntu 18.04.4 LTS (GNU/Linux 5.5.8 x86\_64)
- GNU Make 4.1
- gcc 7.5.0

## H2 1. 过滤页面

### H3 1.1 实现

首先，我们需要过滤掉与进程内数据无关的页面。由于我们首先获取的是进程的VMA结构体，我们可以根据VMA的特征来判断一个VMA所包含的所有页面是否为包含数据的页面。在linux/mm\_types.h中定义的mm\_struct结构体中的一些数据成员引起了我的注意：

```
1 unsigned long total_vm; /* Total pages mapped */
2 unsigned long locked_vm; /* Pages that have PG_mlocked set */
3 atomic64_t pinned_vm; /* Refcount permanently increased */
4 unsigned long data_vm; /* VM_WRITE & ~VM_SHARED & ~VM_STACK */
5 unsigned long exec_vm; /* VM_EXEC & ~VM_WRITE & ~VM_STACK */
6 unsigned long stack_vm; /* VM_STACK */
```

根据注释和变量名，我们知道，这些数据成员统计了一个内存描述符所管理的各种类型的VMA数量。而data\_vm正是包含数据的VMA的数量，其注释告诉我们，可以利用这个特征过滤出包含数据的所有VMA。因此，过滤页面的代码如下：

```
1 static int filter_page(pid_t pid)
2 {
3     struct pid *p;
4     struct task_struct *tsk;
5     struct vm_area_struct *cur;
6     unsigned long filtered_num, addr;
7
8     if (pid < 0)
9     {
10         printk(KERN_ERR "Invalid PID\n");
11         return -EFAULT;
12     }
13
14     p = find_get_pid(pid);
15
16     if (!p)
17     {
18         printk(KERN_ERR "Invalid pid\n");
19         return -EFAULT;
20     }
```

```

21
22     tsk = get_pid_task(p, PIDTYPE_PID);
23
24     if (!tsk)
25     {
26         printk(KERN_ERR "Invalid task_struct\n");
27         return -EFAULT;
28     }
29
30     if (!tsk->mm)
31     {
32         printk(KERN_ERR "task_struct->mm is NULL\n");
33         return -EFAULT;
34     }
35
36     cur = tsk->mm->mmap;
37     filtered_num = 0;
38     heat_info_len = 0;
39
40     while (cur)
41     {
42         if (cur->vm_flags & (VM_WRITE & ~VM_SHARED & ~VM_STACK))
43         {
44             printk(KERN_INFO "FILTERED: 0x%lx\t0x%lx\n", cur->vm_start, cur-
>vm_end);
45             for (addr = cur->vm_start; addr < cur->vm_end; addr += PAGE_SIZE)
46             {
47                 heat_info[heat_info_len + 1].addr = addr;
48                 heat_info[heat_info_len + 1].access_time = 0;
49                 heat_info_len++;
50             }
51             filtered_num += vma_pages(cur);
52         }
53         cur = cur->vm_next;
54     }
55     heat_info[0].addr = 0;
56     heat_info[0].access_time = filtered_num;
57     if (is_show_malloc)
58         printk(KERN_INFO "total: %ld, filtered: %ld, malloc: %ld\n", tsk->mm-
>total_vm, filtered_num, malloc_num);
59     else
60         printk(KERN_INFO "total: %ld, filtered: %ld\n", tsk->mm->total_vm,
filtered_num);
61
62     return 0;
63 }

```

首先调用 `find_get_pid` 以及 `get_pid_task`，从进程PID获得进程的 `task_struct`，从而找到进程的 `vm_area_struct`。随后，遍历 `vm_area_struct` 链表，根据条件过滤页面，并将页面的信息存入 `heat_info` 数组中，置访问次数为0。为了方便后续与用户态的信息传递，将过滤所得数据页面总数存入数组的第一个元素中。在这里，我们假定一个进程的VMA链表在短时间是不发生改变的。这方便了后续热度信息的收集。

### H3 1.2 结果

我们对提供的 `heat.cpp` 与 `heat_rand.cpp` 进行一定的修改，并利用这两个程序来展示这个部分的实验结果。

使用以下命令，便可观察输出（只展示 `heat.cpp` 的结果）

```
1 ./heat 10
2 dmesg
```

```
[20936.355876] filter PID: 15986
[20936.355879] FILTERED: 0x560ab602f000    0x560ab6030000
[20936.355880] FILTERED: 0x560ab6568000    0x560abb5c7000
[20936.355931] FILTERED: 0x7fb074000000    0x7fb075013000
[20936.355942] FILTERED: 0x7fb07c000000    0x7fb07d013000
[20936.355954] FILTERED: 0x7fb080000000    0x7fb081013000
[20936.355964] FILTERED: 0x7fb084000000    0x7fb085013000
[20936.355975] FILTERED: 0x7fb088000000    0x7fb089013000
[20936.355986] FILTERED: 0x7fb08c000000    0x7fb08d013000
[20936.355997] FILTERED: 0x7fb090000000    0x7fb091013000
[20936.356008] FILTERED: 0x7fb094000000    0x7fb097ff9000
[20936.356052] FILTERED: 0x7fb09c000000    0x7fb09fff9000
[20936.356098] FILTERED: 0x7fb0a4000000    0x7fb0affff9000
[20936.356141] FILTERED: 0x7fb0ac000000    0x7fb0affff9000
[20936.356187] FILTERED: 0x7fb0b4000000    0x7fb0b7ff9000
[20936.356233] FILTERED: 0x7fb0bc000000    0x7fb0bfff9000
[20936.356285] FILTERED: 0x7fb0c4000000    0x7fb0c7ff9000
[20936.356339] FILTERED: 0x7fb0cb23f000    0x7fb0cba3f000
[20936.356346] FILTERED: 0x7fb0cba40000    0x7fb0cc240000
[20936.356353] FILTERED: 0x7fb0cc241000    0x7fb0cca41000
[20936.356360] FILTERED: 0x7fb0cca42000    0x7fb0cd242000
[20936.356367] FILTERED: 0x7fb0cd243000    0x7fb0cda43000
[20936.356374] FILTERED: 0x7fb0cda44000    0x7fb0ce244000
[20936.356380] FILTERED: 0x7fb0ce245000    0x7fb0cea45000
[20936.356389] FILTERED: 0x7fb0cec4c000    0x7fb0cec4d000
[20936.356391] FILTERED: 0x7fb0cee60000    0x7fb0cee61000
[20936.356391] FILTERED: 0x7fb0cee61000    0x7fb0cee62000
[20936.356393] FILTERED: 0x7fb0cf067000    0x7fb0cf068000
[20936.356394] FILTERED: 0x7fb0cf26b000    0x7fb0cf26c000
[20936.356395] FILTERED: 0x7fb0cf493000    0x7fb0cf494000
[20936.356396] FILTERED: 0x7fb0cf7c8000    0x7fb0cf7cc000
[20936.356398] FILTERED: 0x7fb0cfa61000    0x7fb0cfa62000
[20936.356398] FILTERED: 0x7fb0cfa62000    0x7fb0cfa82000
[20936.356400] FILTERED: 0x7fb0cfc92000    0x7fb0cfc93000
[20936.356401] FILTERED: 0x7fb0cfc93000    0x7fb0cfc9b3000
[20936.356402] FILTERED: 0x7fb0cfcb6000    0x7fb0cfcb7000
[20936.356403] FILTERED: 0x7fb0d00ce000    0x7fb0d00cf000
[20936.356404] FILTERED: 0x7fb0d0359000    0x7fb0d035a000
[20936.356404] FILTERED: 0x7fb0d035a000    0x7fb0d035b000
[20936.356406] FILTERED: 0x7fb0d0746000    0x7fb0d0748000
[20936.356407] FILTERED: 0x7fb0d0748000    0x7fb0d074c000
[20936.356408] FILTERED: 0x7fb0d0966000    0x7fb0d0967000
[20936.356408] FILTERED: 0x7fb0d0967000    0x7fb0d096b000
[20936.356409] FILTERED: 0x7fb0d0b99000    0x7fb0d0b9a000
[20936.356410] FILTERED: 0x7fb0d0f37000    0x7fb0d0f38000
[20936.356411] FILTERED: 0x7fb0d12bb000    0x7fb0d12bd000
[20936.356411] FILTERED: 0x7fb0d12bd000    0x7fb0d12c1000
[20936.356412] FILTERED: 0x7fb0d1550000    0x7fb0d1551000
[20936.356413] FILTERED: 0x7fb0d1551000    0x7fb0d1556000
[20936.356414] FILTERED: 0x7fb0d1743000    0x7fb0d174d000
[20936.356414] FILTERED: 0x7fb0d177b000    0x7fb0d177d000
[20936.356415] FILTERED: 0x7fb0d177e000    0x7fb0d177f000
[20936.356416] FILTERED: 0x7fb0d177f000    0x7fb0d1780000
[20936.356417] FILTERED: 0x7ffde4a46000    0x7ffde4a67000
[20936.356418] total: 275415, filtered: 178508, malloc: 163840
```

图1. 筛选页面

由于终端显示的行数有限，我截取了 `/var/log/kern.log` 中的内容，可见，筛选出的页面总数为申请页面总数的109%，说明这样的筛选策略是有效的。

```
before: 0x560ab65a3f80 0x560ab65e3f80
after: 0x560ab65abf90 0x560ab65ebf90
before: 0x7fb0c4000b20 0x7fb0c4040b20
after: 0x7fb0c4008b30 0x7fb0c4048b30
before: 0x560ab65b4fb0 0x560ab65f4fb0
after: 0x560ab65bcfc0 0x560ab65fcfc0
before: 0x7fb0c4010b40 0x7fb0c4050b40
after: 0x7fb0c4018b50 0x7fb0c4058b50
before: 0x7fb0bc000b20 0x7fb0bc040b20
after: 0x7fb0bc008b30 0x7fb0bc048b30
```

图2. 申请页面

除此之外，我们观察用户态申请的二维数组中，部分一维数组的起始地址与结束地址，如图2。我们会发现，这些地址都包含在筛选出的VMA中，并且，每一个一维数组的起始地址与页面起始地址都不相同。例如图中的第一个数组的起始地址为0x560ab65a3f80，而它所在的页面起始地址应当为0x560ab65a3000。

## H2 2. 收集页面热度

### H3 2.1 实现

得到了过滤后的页面，就可以开始统计热度信息了。统计热度信息的实现如下：

```
1 static int collect_heat(pid_t pid)
2 {
3     struct pid *p;
4     struct task_struct *tsk;
5     pte_t *pte;
6     int i;
7     ktime_t calltime, delta, retime;
8     unsigned long long duration;
9
10    if (pid < 0)
11    {
12        printk(KERN_ERR "Invalid PID\n");
13        return -EFAULT;
14    }
15
16    p = find_get_pid(pid);
17
18    if (!p)
19    {
20        printk(KERN_ERR "Invalid pid\n");
21        return -EFAULT;
22    }
23
24    tsk = get_pid_task(p, PIDTYPE_PID);
25
```

```

26     if (!tsk)
27     {
28         printk(KERN_ERR "Invalid task_struct\n");
29         return -EFAULT;
30     }
31
32     if (!tsk->mm)
33     {
34         printk(KERN_ERR "task_struct->mm is NULL\n");
35         return -EFAULT;
36     }
37
38     calltime = ktime_get();
39     for (i = 0; i < heat_info_len; ++i)
40     {
41         pte = _find_pte(tsk->mm, heat_info[i + 1].addr);
42         if (!pte)
43         {
44             continue;
45         }
46         if (pte_young(*pte))
47         {
48             heat_info[i + 1].access_time += 1;
49             *pte = pte_mkold(*pte);
50         }
51     }
52     rettime = ktime_get();
53     delta = ktime_sub(rettime, calltime);
54     duration = (unsigned long long)ktime_to_us(delta);
55     printk(KERN_INFO "Heat collection elapsed after %lld us\n", duration);
56     return 0;
57 }

```

可见，同第一部分一样，要调用 `find_get_pid` 以及 `get_pid_task` 获得 `task_struct`。随后遍历之前过滤得到的所有数据页面，并将页面的虚拟地址转化为对应的页表项 `pte`。随后调用 `pte_young` 判断在上一个时间段中，这个页面是否被访问。若被访问过，则需要统计累计访问信息，并调用 `pte_mkold` 清空所有标志位。值得注意的是，`pte_mkold` 并不会改变原有表项的值，而是返回一个新的 `pte`，因此，我们需要将返回值写入这个 `pte` 中，完成更新。注意，有的页面虚拟地址对应的 `pte` 并不存在（没有数据存在），因此我们需要判断并跳过这些虚拟地址。转换虚拟地址到 `pte` 的函数如下：

```

1  static inline pte_t* _find_pte(struct mm_struct *mm, unsigned long addr)
2  {
3      pgd_t *pgd = NULL;
4      p4d_t *p4d = NULL;
5      pud_t *pud = NULL;
6      pmd_t *pmd = NULL;
7      pte_t *pte = NULL;
8
9      pgd = pgd_offset(mm, addr);
10     if (pgd_none(*pgd) || pgd_bad(*pgd))
11         return NULL;

```

```

12
13     p4d = p4d_offset(pgd, addr);
14     if(p4d_none(*p4d) || p4d_bad(*p4d))
15         return NULL;
16
17     pud = pud_offset(p4d, addr);
18     if(pud_none(*pud) || pud_bad(*pud))
19         return NULL;
20
21     pmd = pmd_offset(pud, addr);
22     if(pmd_none(*pmd) || pmd_bad(*pmd))
23         return NULL;
24
25     pte = pte_offset_map(pmd, addr);
26     if(pte_none(*pte) || !pte_present(*pte))
27         return NULL;
28     return pte;
29 }

```

### H3 2.2 结果

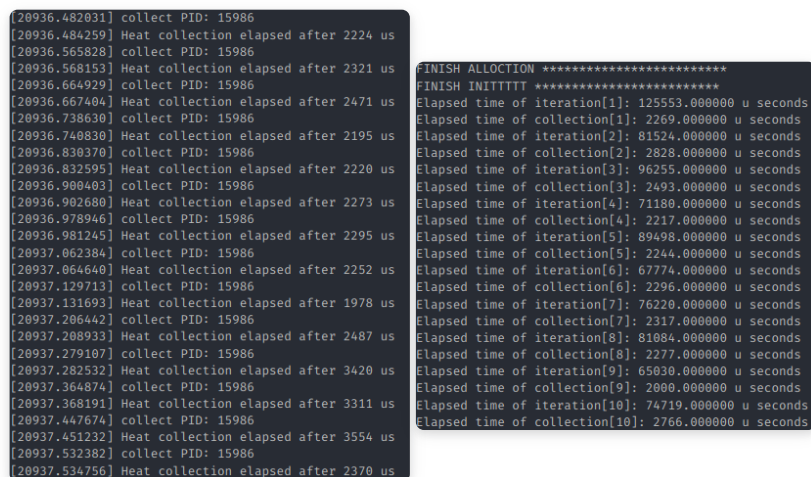
为了更好地测算收集热度所耗费的时间，我们将收集和信息打印分为两个函数实现，通过用户态写入 `/proc/heat` 文件的信息调用这两个函数。打印信息的函数实现非常简单

```

1 static void print_info(void)
2 {
3     int i;
4     for(i = 0; i < heat_info_len; ++i)
5     {
6         printk(KERN_INFO "PAGE: 0x%x\theat: %d\n", heat_info[i + 1].addr,
7             heat_info[i + 1].access_time);
8     }
9 }

```

再此基础上，再次运行 `heat.cpp` 与 `heat_rand.cpp`（输入参数与第一部分一致），我们可以在内核日志中看到获得的热度信息结果。由于一次打印将会输出约17万行，我们只在用户态程序的最后一个循环结束后打印热度信息。在 `/var/log/kern.log` 中的输出以及用户态的输出如下：



```

[20936.482031] collect PID: 15986
[20936.484259] Heat collection elapsed after 2224 us
[20936.565828] collect PID: 15986
[20936.568153] Heat collection elapsed after 2321 us
[20936.664929] collect PID: 15986
[20936.667404] Heat collection elapsed after 2471 us
[20936.738630] collect PID: 15986
[20936.740830] Heat collection elapsed after 2195 us
[20936.830370] collect PID: 15986
[20936.832595] Heat collection elapsed after 2220 us
[20936.900403] collect PID: 15986
[20936.902680] Heat collection elapsed after 2273 us
[20936.978946] collect PID: 15986
[20936.981245] Heat collection elapsed after 2295 us
[20937.062384] collect PID: 15986
[20937.064640] Heat collection elapsed after 2252 us
[20937.129713] collect PID: 15986
[20937.131693] Heat collection elapsed after 1978 us
[20937.206442] collect PID: 15986
[20937.208933] Heat collection elapsed after 2487 us
[20937.279107] collect PID: 15986
[20937.282532] Heat collection elapsed after 3420 us
[20937.364874] collect PID: 15986
[20937.368191] Heat collection elapsed after 3311 us
[20937.447674] collect PID: 15986
[20937.451232] Heat collection elapsed after 3554 us
[20937.532382] collect PID: 15986
[20937.534756] Heat collection elapsed after 2370 us
FINISH ALLOCATION *****
FINISH INITITIT *****
Elapsed time of iteration[1]: 125553.000000 u seconds
Elapsed time of collection[1]: 2269.000000 u seconds
Elapsed time of iteration[2]: 81524.000000 u seconds
Elapsed time of collection[2]: 2828.000000 u seconds
Elapsed time of iteration[3]: 96255.000000 u seconds
Elapsed time of collection[3]: 2493.000000 u seconds
Elapsed time of iteration[4]: 71180.000000 u seconds
Elapsed time of collection[4]: 2217.000000 u seconds
Elapsed time of iteration[5]: 89498.000000 u seconds
Elapsed time of collection[5]: 2244.000000 u seconds
Elapsed time of iteration[6]: 67774.000000 u seconds
Elapsed time of collection[6]: 2296.000000 u seconds
Elapsed time of iteration[7]: 76220.000000 u seconds
Elapsed time of collection[7]: 2317.000000 u seconds
Elapsed time of iteration[8]: 81084.000000 u seconds
Elapsed time of collection[8]: 2277.000000 u seconds
Elapsed time of iteration[9]: 65030.000000 u seconds
Elapsed time of collection[9]: 2000.000000 u seconds
Elapsed time of iteration[10]: 74719.000000 u seconds
Elapsed time of collection[10]: 2766.000000 u seconds

```

图3. 收集热度（部分输出）



可见，收集约170000个页面热度信息所需的时间约为3ms，相对于访问密集型程序的计算时间来说还是比较小的。最终，经历了100个循环后的部分热度信息输出如下：

```
[20944.188669] PAGE: 0x560ab602f000    heat: 100
[20944.188670] PAGE: 0x560ab6568000    heat: 1
[20944.188670] PAGE: 0x560ab6569000    heat: 0
[20944.188670] PAGE: 0x560ab656a000    heat: 0
[20944.188671] PAGE: 0x560ab656b000    heat: 0
[20944.188671] PAGE: 0x560ab656c000    heat: 0
[20944.188672] PAGE: 0x560ab656d000    heat: 0
[20944.188672] PAGE: 0x560ab656e000    heat: 0
[20944.188673] PAGE: 0x560ab656f000    heat: 0
[20944.188673] PAGE: 0x560ab6570000    heat: 1
[20944.188674] PAGE: 0x560ab6571000    heat: 0
[20944.188674] PAGE: 0x560ab6572000    heat: 0
[20944.188675] PAGE: 0x560ab6573000    heat: 0
[20944.188675] PAGE: 0x560ab6574000    heat: 0
[20944.188675] PAGE: 0x560ab6575000    heat: 0
[20944.188676] PAGE: 0x560ab6576000    heat: 0
[20944.188676] PAGE: 0x560ab6577000    heat: 0
[20944.188677] PAGE: 0x560ab6578000    heat: 0
[20944.188677] PAGE: 0x560ab6579000    heat: 0
[20944.188678] PAGE: 0x560ab657a000    heat: 100
[20944.188678] PAGE: 0x560ab657b000    heat: 100
[20944.188679] PAGE: 0x560ab657c000    heat: 100
[20944.188679] PAGE: 0x560ab657d000    heat: 100
[20944.188679] PAGE: 0x560ab657e000    heat: 100
[20944.188680] PAGE: 0x560ab657f000    heat: 100
[20944.188680] PAGE: 0x560ab6580000    heat: 100
[20944.188681] PAGE: 0x560ab6581000    heat: 100
[20944.188681] PAGE: 0x560ab6582000    heat: 100
[20944.188682] PAGE: 0x560ab6583000    heat: 100
[20944.188682] PAGE: 0x560ab6584000    heat: 100
[20944.188683] PAGE: 0x560ab6585000    heat: 100
[20944.188683] PAGE: 0x560ab6586000    heat: 100
[20944.188684] PAGE: 0x560ab6587000    heat: 100
[20944.188684] PAGE: 0x560ab6588000    heat: 100
[20944.188685] PAGE: 0x560ab6589000    heat: 100
[20944.188685] PAGE: 0x560ab658a000    heat: 100
```

图4. 热度信息

## H2 3. 与用户态交互

### H3 3.1 实现

与用户态程序的交互分为两部分，一部分是用户态将命令写入`/proc/heat`文件，触发相应功能，另一部分是用户态读取共享在`/proc/heat`中的热度信息并输出。

proc文件系统中文件定义的各种操作不再叙述。写函数的定义也很简单，并且已经在之前两个部分实现了：

```
1 static ssize_t getHeat_proc_write(struct file *file, const char __user *buffer,
2 size_t count, loff_t *data)
3 {
4     int pid;
5     int ret;
6     if (copy_from_user(proc_buf, buffer, count))
7     {
8         printk(KERN_ERR "Copy from user unfinished\n");
9         return -EFAULT;
10    }
11    proc_buf[count] = '\0';
12    if (strncmp(proc_buf, "filter", 6) == 0)
13    {
14        sscanf(proc_buf + 7, "%d", &pid);
15        printk(KERN_INFO "filter PID: %d\n", pid);
16        ret = filter_page(pid);
17    }
18    else if (strncmp(proc_buf, "collect", 7) == 0)
19    {
20        sscanf(proc_buf + 8, "%d", &pid);
21        printk(KERN_INFO "collect PID: %d\n", pid);
22        ret = collect_heat(pid);
23    }
24    else if (strncmp(proc_buf, "malloc", 6) == 0)
25    {
26        sscanf(proc_buf + 7, "%d %d", &is_show_malloc, &malloc_num);
27    }
28    else if (strncmp(proc_buf, "print", 5) == 0)
29    {
30        print_info();
31    }
32    else
33    {
34        printk(KERN_ERR "Invalid input!\n");
35    }
36
37    return count;
38 }
```

这里的命令分为以下四种

1. filter PID：过滤给定PID对应进程的数据页面
2. collect PID：在过滤的基础上，收集给定PID对应进程的页面热度信息
3. malloc 0/1 size：是否（0/1）打印用户态程序所分配的页面数
4. print：内核态打印热度信息

由于需要在用户态的程序中输出热度信息，我们需要通过proc文件系统共享heat\_info数组，并通过自定义读函数来实现用户态的热度信息输出。这部分内核态的实现也很简单



```

1 static ssize_t getHeat_proc_read(struct file *filp, char *usr_buf, size_t count,
  loff_t *offp)
2 {
3     if (copy_to_user(usr_buf, heat_info, sizeof(heat_info)))
4     {
5         printk(KERN_ERR "Copy to user unfinished\n");
6         return -EFAULT;
7     }
8     return sizeof(heat_info);
9 }

```

可以直接将 `heat_info` 复制到用户空间的缓冲区去，然后在用户态程序中声明同样的结构体以及数组，使用读文件函数即可获得这些信息。

我们将 `heat.cpp` 以及 `heat_rand.cpp` 进行修改。首先，将获取、打印热度信息与原有计算函数 `heat()` 分为两个线程。即

```

1 int main(int argc, char *argv[]){
2     pthread_t id;
3     int ret;
4     .....
5     ret = pthread_create(&id, NULL, &collect, NULL);
6     if (ret == 0)
7     {
8         printf("-----Start heat collection-----\n");
9     }
10    else
11    {
12        printf("Start collection failed!\n");
13        return 0;
14    }
15
16    heat();
17
18    return 0;
19 }

```

在 `heat()` 函数中，使之每计算5个循环休眠一定时间。`collect` 线程的实现如下：

```

1 void* collect(void* args)
2 {
3     int f, i, c, actual_len;
4     char write_info[100];
5     // char read_info[5000000];
6
7     sleep(1);
8
9     f = open("/proc/heat", O_RDWR | O_TRUNC);
10
11    ftruncate(f, 0);
12    lseek(f, 0, SEEK_SET);

```

```

13  sprintf(write_info, "malloc 0 0");
14  write(f, write_info, strlen(write_info));
15
16  ftruncate(f, 0);
17  lseek(f, 0, SEEK_SET);
18  sprintf(write_info, "filter %d", getpid());
19  write(f, write_info, strlen(write_info));
20
21  for (c = 0; c < 50; ++c)
22  {
23      ftruncate(f, 0);
24      lseek(f, 0, SEEK_SET);
25      sprintf(write_info, "collect %d", getpid());
26      write(f, write_info, strlen(write_info));
27      read(f, read_info, sizeof(read_info));
28      actual_len = read_info[0].access_time;
29      printf("-----HEAT AT %d-----\n", c);
30      for (i = 1; i <= actual_len; ++i)
31      {
32          if(read_info[i].access_time > 0)
33              printf("PAGE: 0x%lx\tHEAT: %d\n", read_info[i].addr,
34                  read_info[i].access_time);
35      }
36      printf("-----HEAT AT %d-----\n", c);
37      usleep(50000);
38  }
39  close(f);
40  }

```

首先需要等待 `heat()` 函数初始化完毕，然后打开 `/proc/heat` 文件，并进行页面过滤。随后执行50次热度收集与打印。每一次打印完毕后，需要睡眠一定时间，让 `heat()` 进行一定的计算。而这个睡眠时间是人为设定的，必须要恰当地设置，使得 `heat()` 函数返回后 `collect()` 函数才返回。这样就完成了内核态和用户态之间热度信息的传递。

### H3 3.2 结果

直接运行benchmark中的两个程序（输入参数为10）即可观察热度信息的打印。由于收集一次热度信息输出的字符太多（约300万个），如果在终端使用printf输出，会花费大量时间，需要调整heat线程的睡眠时间，于是我将输出重定向至res文件中，以下是输出的结果：

```

-----HEAT AT 0-----
PAGE: 0x55e60ad7f000  HEAT: 1
PAGE: 0x55e60ad7f000  HEAT: 1
PAGE: 0x55e60ad87f00  HEAT: 1
PAGE: 0x55e60ad87f00  HEAT: 1
PAGE: 0x55e60ad891000  HEAT: 1
PAGE: 0x55e60ad892000  HEAT: 1
PAGE: 0x55e60ad893000  HEAT: 1
PAGE: 0x55e60ad894000  HEAT: 1
PAGE: 0x55e60ad895000  HEAT: 1
PAGE: 0x55e60ad896000  HEAT: 1
PAGE: 0x55e60ad897000  HEAT: 1
PAGE: 0x55e60ad898000  HEAT: 1
PAGE: 0x55e60ad899000  HEAT: 1
PAGE: 0x55e60ad89a000  HEAT: 1
PAGE: 0x55e60ad89b000  HEAT: 1
PAGE: 0x55e60ad89c000  HEAT: 1
PAGE: 0x55e60ad89d000  HEAT: 1
PAGE: 0x55e60ad89e000  HEAT: 1
PAGE: 0x55e60ad89f000  HEAT: 1
PAGE: 0x55e60da00000  HEAT: 1
PAGE: 0x55e60da1000  HEAT: 1
PAGE: 0x55e60da2000  HEAT: 1
PAGE: 0x55e60da3000  HEAT: 1
PAGE: 0x55e60da4000  HEAT: 1
PAGE: 0x55e60da5000  HEAT: 1
PAGE: 0x55e60da6000  HEAT: 1
PAGE: 0x55e60da7000  HEAT: 1

-----HEAT AT 09-----
PAGE: 0x7f433d800000  HEAT: 50
PAGE: 0x7f433d800000  HEAT: 50
PAGE: 0x7f433d800000  HEAT: 1
PAGE: 0x7f433d8005000  HEAT: 1
PAGE: 0x7f433d8006000  HEAT: 1
PAGE: 0x7f433d8007000  HEAT: 1
PAGE: 0x7f433d8008000  HEAT: 1
PAGE: 0x7f433d8009000  HEAT: 1
PAGE: 0x7f433d8224000  HEAT: 1
PAGE: 0x7f433d8228000  HEAT: 50
PAGE: 0x7f433d84b8000  HEAT: 1
PAGE: 0x7f433d86ab000  HEAT: 1
PAGE: 0x7f433d86ac000  HEAT: 1
PAGE: 0x7f433d86ad000  HEAT: 50
PAGE: 0x7f433d86ae000  HEAT: 1
PAGE: 0x7f433d86af000  HEAT: 1
PAGE: 0x7f433d86b0000  HEAT: 1
PAGE: 0x7f433d86b1000  HEAT: 1
PAGE: 0x7f433d86b2000  HEAT: 1
PAGE: 0x7f433d86b3000  HEAT: 1
PAGE: 0x7f433d86b4000  HEAT: 1
PAGE: 0x7f433d86e3000  HEAT: 1
PAGE: 0x7f433d86e4000  HEAT: 1
PAGE: 0x7f433d86e6000  HEAT: 1
PAGE: 0x7f433d86e7000  HEAT: 1
PAGE: 0x7ffc7776f000  HEAT: 1
PAGE: 0x7ffc77770000  HEAT: 50
PAGE: 0x7ffc77771000  HEAT: 1
PAGE: 0x7ffc77772000  HEAT: 1

```

图5. 用户态输出 (部分)

除此之外，我还绘制了热度信息的大致分布。

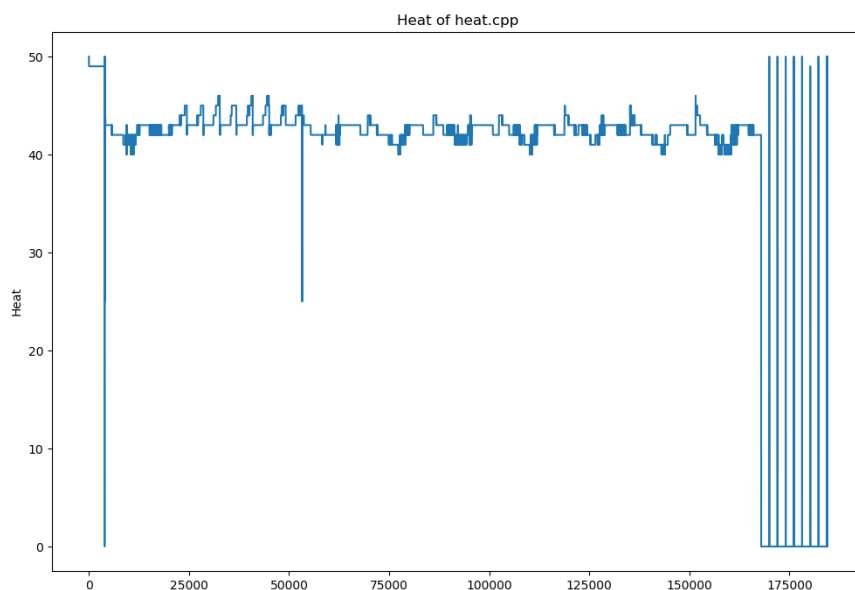


图6. heat.cpp 热度分布

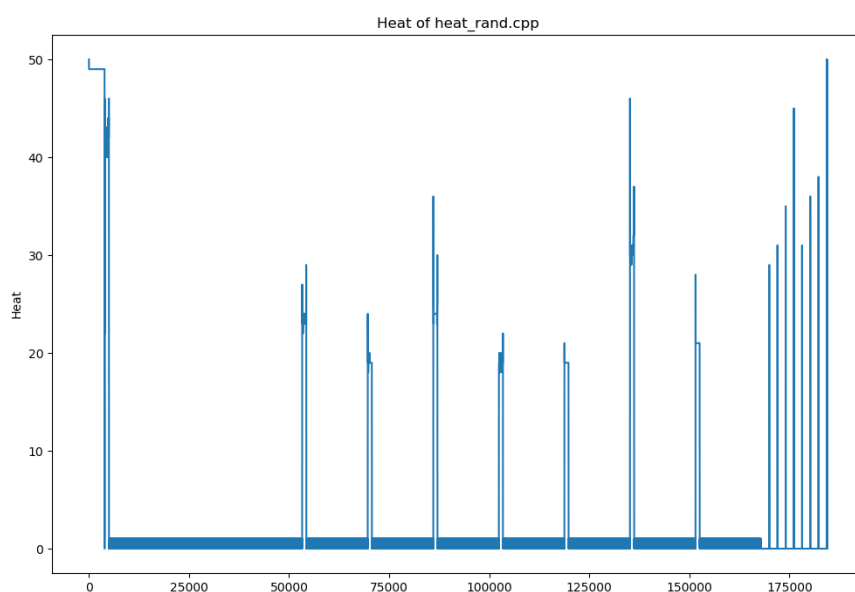


图7. heat\_rand.cpp 热度分布

由此可见，heat.cpp对于数据的访问频次比较均匀，而heat\_rand.cpp对数据的访问主要集中在某几个地址。至于高地址部分，两个程序都出现了密集的波峰，这可能是因为数据页面在这一部分的分布比较分散。

## H2 4. 总结

在本大作业中，我实现了进程数据页面的过滤、热度信息的收集以及为用户态程序的交互。这次的大作业加深了我对于Linux内核内存管理相关知识的理解，并且进一步提升了我的Linux内核编程能力，我受益匪浅。