CS353 Linux内核 大作业报告

517030910214 刘宏洲

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

1. 过滤页面

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：

```
unsigned long total_vm;   /* Total pages mapped */
unsigned long locked_vm;  /* Pages that have PG_mlocked set */
atomic64_t    pinned_vm;  /* Refcount permanently increased */
unsigned long data_vm;    /* VM_WRITE & ~VM_SHARED & ~VM_STACK */
unsigned long exec_vm;    /* VM_EXEC & ~VM_WRITE & ~VM_STACK */
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：

```
static int filter_page(pid_t pid)
{
    struct pid *p;
    struct task_struct *tsk;
    struct vm_area_struct *cur;
    unsigned long filtered_num, addr;

    if (pid < 0)
    {
        printk(KERN_ERR "Invalid PID\n");
        return -EFAULT;
    }

    p = find_get_pid(pid);

    if (!p)
    {
        printk(KERN_ERR "Invalid pid\n");
        return -EFAULT;
    }
```

```
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链表在短时间内是不发生改变的。这方便了后续热度信息的收集。

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）

```
./heat 10
dmesg
```

```
[20936.355876] filter PID: 15986
[20936.355879] FILTERED: 0×560ab602f000    0×560ab6030000
[20936.355880] FILTERED: 0×560ab6568000    0×560abb5c7000
[20936.355931] FILTERED: 0×7fb074000000    0×7fb075013000
[20936.355942] FILTERED: 0×7fb07c000000    0×7fb07d013000
[20936.355954] FILTERED: 0×7fb080000000    0×7fb081013000
[20936.355964] FILTERED: 0×7fb084000000    0×7fb085013000
[20936.355975] FILTERED: 0×7fb088000000    0×7fb089013000
[20936.355986] FILTERED: 0×7fb08c000000    0×7fb08d013000
[20936.355997] FILTERED: 0×7fb090000000    0×7fb091013000
[20936.356008] FILTERED: 0×7fb094000000    0×7fb097ff9000
[20936.356052] FILTERED: 0×7fb09c000000    0×7fb09fff9000
[20936.356098] FILTERED: 0×7fb0a4000000    0×7fb0a7ff9000
[20936.356141] FILTERED: 0×7fb0ac000000    0×7fb0afff9000
[20936.356187] FILTERED: 0×7fb0b4000000    0×7fb0b7ff9000
[20936.356233] FILTERED: 0×7fb0bc000000    0×7fb0bfff9000
[20936.356285] FILTERED: 0×7fb0c4000000    0×7fb0c7ff9000
[20936.356339] FILTERED: 0×7fb0cb23f000    0×7fb0cba3f000
[20936.356346] FILTERED: 0×7fb0cba40000    0×7fb0cc240000
[20936.356353] FILTERED: 0×7fb0cc241000    0×7fb0cca41000
[20936.356360] FILTERED: 0×7fb0cca42000    0×7fb0cd242000
[20936.356367] FILTERED: 0×7fb0cd243000    0×7fb0cda43000
[20936.356374] FILTERED: 0×7fb0cda44000    0×7fb0ce244000
[20936.356380] FILTERED: 0×7fb0ce245000    0×7fb0cea45000
[20936.356389] FILTERED: 0×7fb0cec4c000    0×7fb0cec4d000
[20936.356391] FILTERED: 0×7fb0cee60000    0×7fb0cee61000
[20936.356391] FILTERED: 0×7fb0cee61000    0×7fb0cee62000
[20936.356393] FILTERED: 0×7fb0cf067000    0×7fb0cf068000
[20936.356394] FILTERED: 0×7fb0cf26b000    0×7fb0cf26c000
[20936.356395] FILTERED: 0×7fb0cf493000    0×7fb0cf494000
[20936.356396] FILTERED: 0×7fb0cf7c8000    0×7fb0cf7cc000
[20936.356398] FILTERED: 0×7fb0cfa61000    0×7fb0cfa62000
[20936.356398] FILTERED: 0×7fb0cfa62000    0×7fb0cfa82000
[20936.356400] FILTERED: 0×7fb0cfc92000    0×7fb0cfc93000
[20936.356401] FILTERED: 0×7fb0cfc93000    0×7fb0cfcb3000
[20936.356402] FILTERED: 0×7fb0cfeb6000    0×7fb0cfeb7000
[20936.356403] FILTERED: 0×7fb0d00ce000    0×7fb0d00cf000
[20936.356404] FILTERED: 0×7fb0d0359000    0×7fb0d035a000
[20936.356404] FILTERED: 0×7fb0d035a000    0×7fb0d035b000
[20936.356406] FILTERED: 0×7fb0d0746000    0×7fb0d0748000
[20936.356407] FILTERED: 0×7fb0d0748000    0×7fb0d074c000
[20936.356408] FILTERED: 0×7fb0d0966000    0×7fb0d0967000
[20936.356408] FILTERED: 0×7fb0d0967000    0×7fb0d096b000
[20936.356409] FILTERED: 0×7fb0d0b99000    0×7fb0d0b9a000
[20936.356410] FILTERED: 0×7fb0d0f37000    0×7fb0d0f38000
[20936.356411] FILTERED: 0×7fb0d12bb000    0×7fb0d12bd000
[20936.356411] FILTERED: 0×7fb0d12bd000    0×7fb0d12c1000
[20936.356412] FILTERED: 0×7fb0d1550000    0×7fb0d1551000
[20936.356413] FILTERED: 0×7fb0d1551000    0×7fb0d1556000
[20936.356414] FILTERED: 0×7fb0d1743000    0×7fb0d174d000
[20936.356414] FILTERED: 0×7fb0d177b000    0×7fb0d177d000
[20936.356415] FILTERED: 0×7fb0d177e000    0×7fb0d177f000
[20936.356416] FILTERED: 0×7fb0d177f000    0×7fb0d1780000
[20936.356417] FILTERED: 0×7ffde4a46000    0×7ffde4a67000
[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: 0×560ab65a3f80 0×560ab65e3f80
after: 0×560ab65abf90 0×560ab65ebf90
before: 0×7fb0c4000b20 0×7fb0c4040b20
after: 0×7fb0c4008b30 0×7fb0c4048b30
before: 0×560ab65b4fb0 0×560ab65f4fb0
after: 0×560ab65bcfc0 0×560ab65fcfc0
before: 0×7fb0c4010b40 0×7fb0c4050b40
after: 0×7fb0c4018b50 0×7fb0c4058b50
before: 0×7fb0bc000b20 0×7fb0bc040b20
after: 0×7fb0bc008b30 0×7fb0bc048b30
```

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。

2. 收集页面热度

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, rettime;
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
```

```
    pgd = pgd_offset(mm, addr);
    if(pgd_none(*pgd) || pgd_bad(*pgd))
        return NULL;

    p4d = p4d_offset(pgd, addr);
    if(p4d_none(*p4d) || p4d_bad(*p4d))
        return NULL;

    pud = pud_offset(p4d, addr);
    if(pud_none(*pud) || pud_bad(*pud))
        return NULL;

    pmd = pmd_offset(pud, addr);
    if(pmd_none(*pmd) || pmd_bad(*pmd))
        return NULL;

    pte = pte_offset_map(pmd, addr);
    if(pte_none(*pte) || !pte_present(*pte))
        return NULL;
    return pte;
}
```

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

```
static void print_info(void)
{
    int i;
    for (i = 0; i < heat_info_len; ++i)
    {
        printk(KERN_INFO "PAGE: 0x%lx\theat: %d\n", heat_info[i + 1].addr, heat_info[i + 1].access_time);
    }
}
```

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 ALLOCTION ************************
FINISH INITTTTT ************************
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：

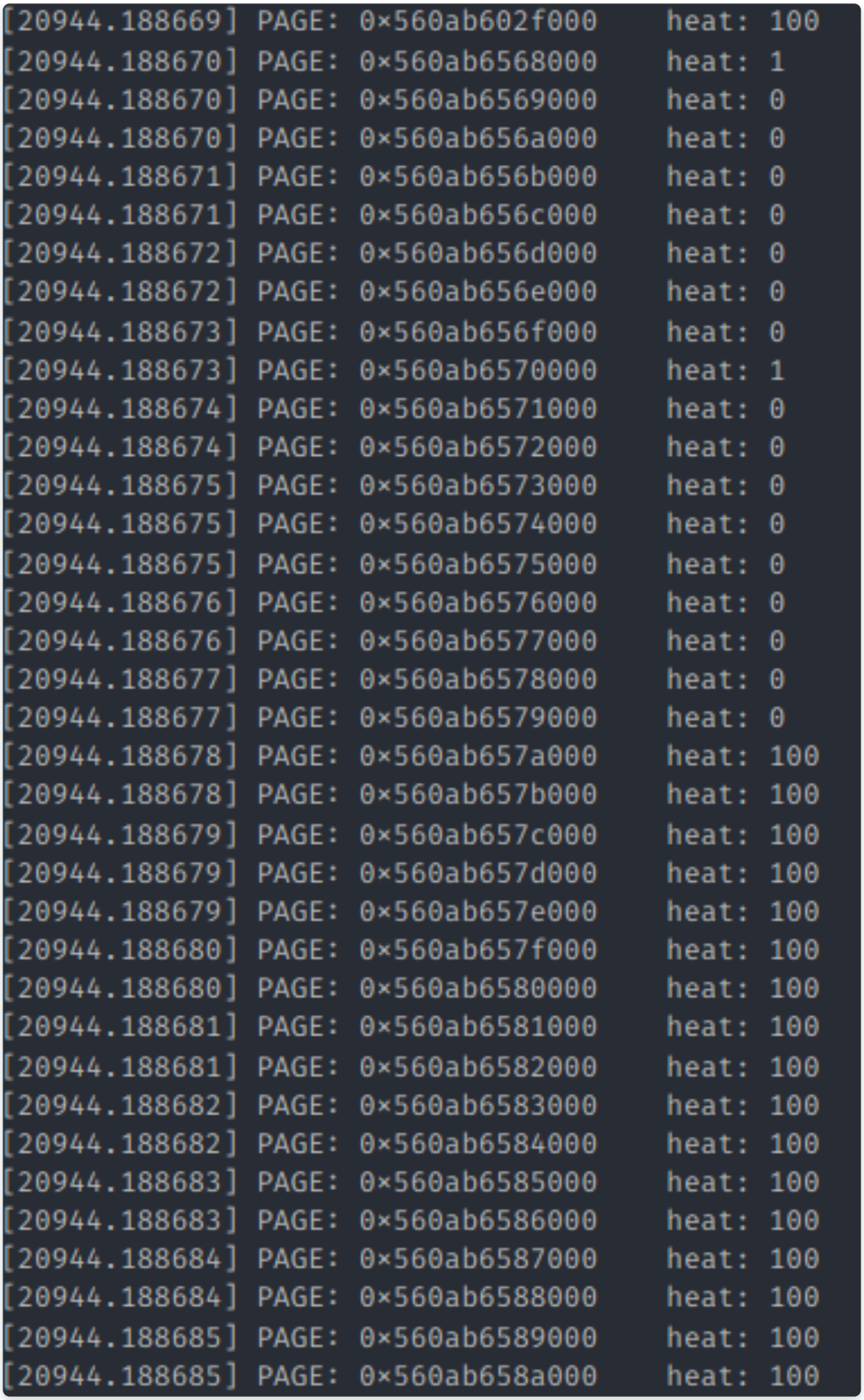

图4. 热度信息

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：

```c
static ssize_t getHeat_proc_write(struct file *file, const char __user *buffer,
size_t count, loff_t *data)
{
    int pid;
    int ret;
    if (copy_from_user(proc_buf, buffer, count))
    {
        printk(KERN_ERR "Copy from user unfinished\n");
        return -EFAULT;
    }
    proc_buf[count] = '\0';

    if (strncmp(proc_buf, "filter", 6) == 0)
    {
        sscanf(proc_buf + 7, "%d", &pid);
        printk(KERN_INFO "filter PID: %d\n", pid);
        ret = filter_page(pid);
    }
    else if (strncmp(proc_buf, "collect", 7) == 0)
    {
        sscanf(proc_buf + 8, "%d", &pid);
        printk(KERN_INFO "collect PID: %d\n", pid);
        ret = collect_heat(pid);
    }
    else if (strncmp(proc_buf, "malloc", 6) == 0)
    {
        sscanf(proc_buf + 7, "%d %ld", &is_show_malloc, &malloc_num);
    }
    else if (strncmp(proc_buf, "print", 5) == 0)
    {
        print_info();
    }
    else
    {
        printk(KERN_ERR "Invalid input!\n");
    }

    return count;
}
```

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

```
static ssize_t getHeat_proc_read(struct file *filp, char *usr_buf, size_t count,
loff_t *offp)
{
  if (copy_to_user(usr_buf, heat_info, sizeof(heat_info)))
  {
    printk(KERN_ERR "Copy to user unfinished\n");
    return -EFAULT;
  }
  return sizeof(heat_info);
}
```

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

```
int main(int argc, char *argv[]){
  pthread_t id;
  int ret;
  ......
  ret = pthread_create(&id, NULL, &collect, NULL);
  if (ret == 0)
  {
    printf("----------------------------Start heat collection----------------------------
\n");
  }
  else
  {
    printf("Start collection failed!\n");
    return 0;
  }

  heat();

  return 0;
}
```

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：

```
void* collect(void* args)
{
  int f, i, c, actual_len;
  char write_info[100];
  // char read_info[5000000];

  sleep(1);

  f = open("/proc/heat", O_RDWR | O_TRUNC);

  ftruncate(f, 0);
  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,
    read_info[i].access_time);
34        }
35        printf("----------------------------HEAT AT %d----------------------------\n", c);
36        usleep(50000);
37      }
38      close(f);
39  }
```

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。

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：

```
----------------------------Start heat collection----------------------------
FINISH ALLOCTION ************************
FINISH INITTTTT ************************
----------------------------HEAT AT 0----------------------------
PAGE: 0×55e60adf7000    HEAT: 1
PAGE: 0×55e60d87f000    HEAT: 1
PAGE: 0×55e60d887000    HEAT: 1
PAGE: 0×55e60d891000    HEAT: 1
PAGE: 0×55e60d892000    HEAT: 1
PAGE: 0×55e60d893000    HEAT: 1
PAGE: 0×55e60d894000    HEAT: 1
PAGE: 0×55e60d895000    HEAT: 1
PAGE: 0×55e60d896000    HEAT: 1
PAGE: 0×55e60d897000    HEAT: 1
PAGE: 0×55e60d898000    HEAT: 1
PAGE: 0×55e60d899000    HEAT: 1
PAGE: 0×55e60d89a000    HEAT: 1
PAGE: 0×55e60d89b000    HEAT: 1
PAGE: 0×55e60d89c000    HEAT: 1
PAGE: 0×55e60d89d000    HEAT: 1
PAGE: 0×55e60d89e000    HEAT: 1
PAGE: 0×55e60d89f000    HEAT: 1
PAGE: 0×55e60d8a0000    HEAT: 1
PAGE: 0×55e60d8a1000    HEAT: 1
PAGE: 0×55e60d8a2000    HEAT: 1
PAGE: 0×55e60d8a3000    HEAT: 1
PAGE: 0×55e60d8a4000    HEAT: 1
PAGE: 0×55e60d8a5000    HEAT: 1
PAGE: 0×55e60d8a6000    HEAT: 1
PAGE: 0×55e60d8a7000    HEAT: 1
PAGE: 0×7f433d8e2000    HEAT: 50
PAGE: 0×7f433dc80000    HEAT: 50
PAGE: 0×7f433e004000    HEAT: 1
PAGE: 0×7f433e005000    HEAT: 1
PAGE: 0×7f433e006000    HEAT: 1
PAGE: 0×7f433e007000    HEAT: 1
PAGE: 0×7f433e008000    HEAT: 1
PAGE: 0×7f433e009000    HEAT: 1
PAGE: 0×7f433e224000    HEAT: 1
PAGE: 0×7f433e228000    HEAT: 50
PAGE: 0×7f433e4b8000    HEAT: 1
PAGE: 0×7f433e6ab000    HEAT: 1
PAGE: 0×7f433e6ac000    HEAT: 1
PAGE: 0×7f433e6ad000    HEAT: 50
PAGE: 0×7f433e6ae000    HEAT: 1
PAGE: 0×7f433e6af000    HEAT: 1
PAGE: 0×7f433e6b0000    HEAT: 1
PAGE: 0×7f433e6b1000    HEAT: 1
PAGE: 0×7f433e6b2000    HEAT: 1
PAGE: 0×7f433e6b3000    HEAT: 1
PAGE: 0×7f433e6b4000    HEAT: 1
PAGE: 0×7f433e6e3000    HEAT: 1
PAGE: 0×7f433e6e4000    HEAT: 1
PAGE: 0×7f433e6e6000    HEAT: 1
PAGE: 0×7f433e6e7000    HEAT: 1
PAGE: 0×7ffcb776f000    HEAT: 1
PAGE: 0×7ffcb7770000    HEAT: 50
PAGE: 0×7ffcb7771000    HEAT: 1
PAGE: 0×7ffcb7772000    HEAT: 1
----------------------------HEAT AT 49----------------------------
I've done!
```

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。

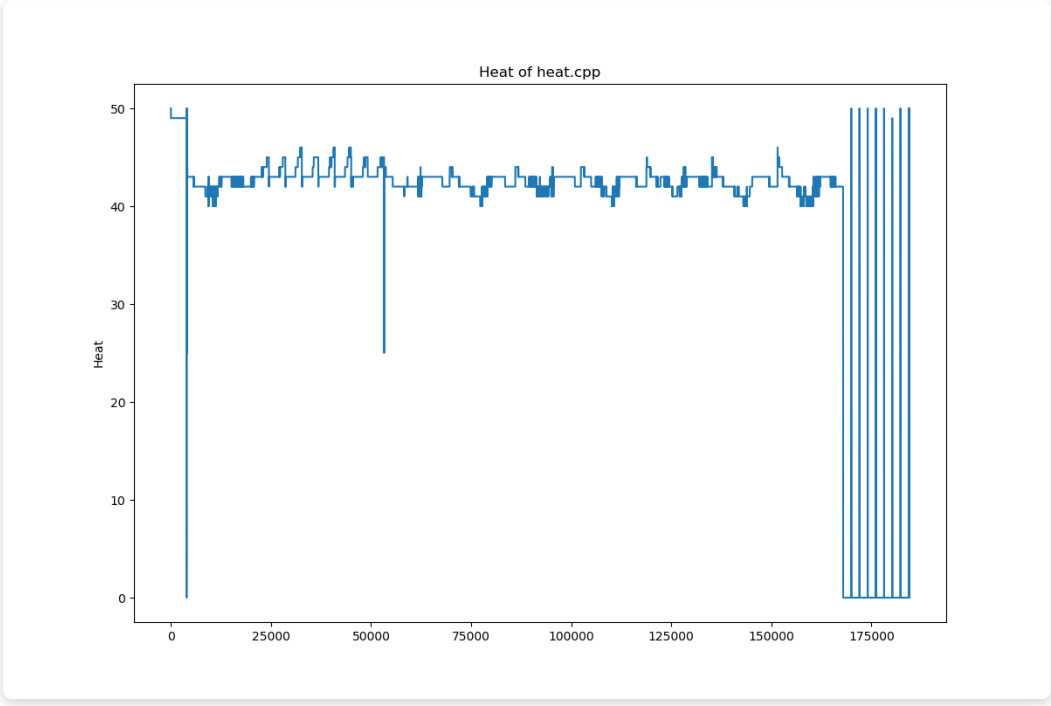

图6. heat.cpp 热度分布

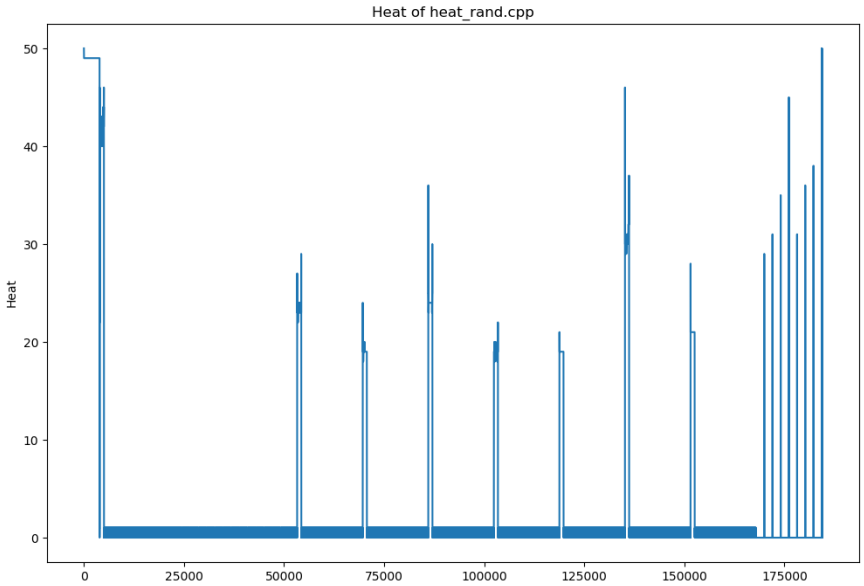

图7. heat_rand.cpp 热度分布

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与用户态程序的交互。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。