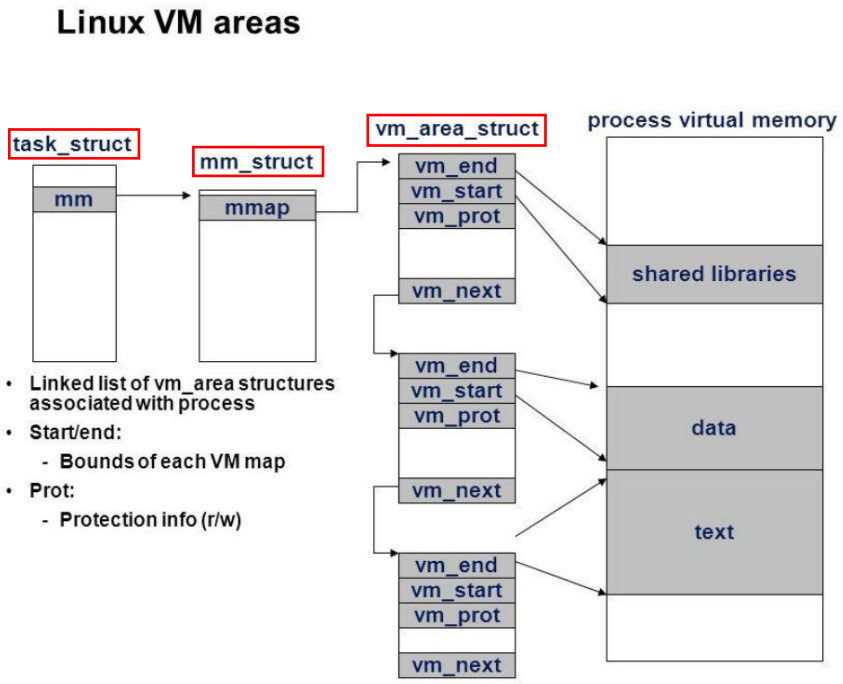
Term Project 1

PROCRANK kernel module

2017313008 김태은

1. Kernel data structure study



- kernel은 process들을 task_list라고 부르는 환형 양방향 링크드리스트를 이용하여 관리, 저장한다. task_list의 각 항목을 process descriptor이라고 하며 <linux/sched.h>에 struct task_struct 형식으로 구현되어 있다. task_struct 구조체 내에는 사용 중인 파일, 프로세스의 주소 공간, 프로세스 상태 등 프로세스를 설명할 수 있는 많은 정보가 담겨있다.
- 각 task는 자신만의 고유한 가상 메모리를 가진다. task_struct 구조체 내에서 task가 사용하는 모든 memory 정보를 담고 있는 자료구조로 struct mm_struct *mm (mm은 Memory Management의 약자)이 선언되어 있다. 이 mm_struct를 memory descriptor라고 하며 <linux/mm_types.h>에 구현되어 있다. 각 task의 메모리 정보에 접근하고 싶으면 struct task_struct *p; p->mm으로 접근할 수 있다.
- 리눅스 커널에서 실행되고 있는 프로세스는 크게 2가지로 분류할 수 있는데, user 공간에서 생성된 user process와 kernel 공간에서 생성된 kernel process(kernel thread)이다. kernel thread는 task struct 구조체 안의 mm이 NULL이다.
- mm_struct 구조체 내에 가상 메모리 영역을 관리하기 위해 사용하는 자료구조로 struct vm_area_struct *mmap이 선언되어 있다. 이 구조체 또한 <linux/mm_types.h>에 구현되어 있다. vm_area_struct들은 리스트로 연결되어 있고, mmap 변수는 리스트의 시작이다. vm_area_struct 구조체 내에는 vm_start (영역의 시작 주소), vm_end (영역의 끝 주소), vm_next (다음 가상 메모리 블록을 가리킴, 마지막 블록은 NULL을 가리킴) 등이 존재한다. 각 task의 vm_area_struct에 접근하고 싶으면 struct task_struct *p; struct vm_area_struct * vma=p->mm->mmap 한 뒤 vma=vma->vm_next를 통해 vm_area_struct를 traverse할 수 있다.
- 리눅스 커널은 위의 그림과 같은 virtual memory management structure를 사용하여 memory를 관리한다.

2. Detailed explanation of your implementations

development environment: ubuntu 16.04 (VM)

```
struct task_struct *p;
struct mem_size_stats mss;
struct vm_area_struct *vma;

unsigned long sharedMemory=0, vss=0, rss=0, uss=0;
unsigned long long pss=0;
```

```
for_each_process(p) // Traverse all processes
{

        memset(&mss, 0, sizeof(mss));
        vss=0, rss=0, uss=0, pss=0;

        if(p->mm!=NULL){
                for (vma=p->mm->mmap; vma; vma= vma->vm_next) {
                        smap_gather_stats(vma, &mss);
                        vss+=(vma->vm_end-vma->vm_start);
                }
                sharedMemory=mss.shared_clean+mss.shared_dirty;
                rss=mss.resident;
                uss=rss-sharedMemory;
                pss=mss.pss;


        }
        insert_node(p, vss, rss, uss, pss);
}
```

- kernel macro “for_each_process”를 사용하여 모든 process들을 traverse한다.
- 새로운 process가 들어올 때마다 mss, vss, rss, uss, pss를 0으로 초기화한다.
- kernel thread의 경우 mm_struct가 null(p->mm==NULL)이므로 vm_area_struct 또한 존재하지 않는다. 따라서 커널 쓰레드가 아닌 경우(p->mm!=NULL)에만 vm_area_struct를 traverse할 수 있도록 한다. 또한 kernel thread의 경우 pid, process name만 출력하고 나머지 vss, rss, uss, pss는 모두 0으로 출력될 수 있도록 한다.
- kernel thread가 아닌 경우에는 vm_area_struct를 traverse해서 vss, rss, uss, pss를 구할 수 있다.
- for문의 (vma=vma->vm_next)를 통해 vm_area_struct를 traverse하며 smap_gather_stats(vma, &mss)를 해서 mem_size_stats mss를 update한다.
- VSS(Virtual Set Size): for문의 (vma=vma->vm_next)를 통해 vm_area_struct를 traverse하며 각 vma마다 vm_end와 vm_start의 차이를 더해주면서 구할 수 있다.
- RSS(Resident Set Size): vm_area_struct를 모두 traverse한 뒤의 mss.resident로 구할 수 있다.
- USS(Unique Set Size): RSS에서 shared memory를 뺀 값과 동일하다. shared memory는 mss.shared_clean+mss.shared_dirty로 구할 수 있다.
- PSS(Proportional Set Size): vm_area_struct를 모두 traverse한 뒤의 mss.pss로 구할 수 있다.
- insert_node함수: 프로세스 별 정보를 linked list에 삽입한다. 이는 process 정보를 PSS 값에 의해 내림차순 정렬해서 출력하기 위한 것이다.

```c
typedef struct node{
        struct task_struct *p;
        unsigned long vss, rss, uss;
        unsigned long long pss;
        struct node *next;
}NODE;
```

- 프로세스 별 정보(task struct pointer, vss, rss, uss, pss)를 저장하고 있는 NODE

```c
void insert_node(struct task_struct *p, unsigned long vss, unsigned long rss, unsigned long uss, unsigned long long pss){

        NODE *cur=start;
        NODE *newNode=(NODE *)kmalloc(sizeof(NODE), GFP_KERNEL);
        newNode->p=p;
        newNode->vss=vss;
        newNode->rss=rss;
        newNode->uss=uss;
        newNode->pss=pss;

        newNode->next=NULL;

        if(start==NULL){
                start=newNode;
                newNode->next=NULL;
                return;
        }
        if(newNode->pss > start->pss){
                newNode->next=start;
                start=newNode;
                return;
        }

        while(1){
                if(cur->next==NULL){
                        cur->next=newNode;
                        newNode->next=NULL;
                        return;
                }
                else if(cur->next->pss < newNode->pss){
                        newNode->next=cur->next;
                        cur->next=newNode;
                        return;
                }
                else cur=cur->next;
        }
}
```

- insert_node: 프로세스 별 정보를 linked list에 삽입하는 함수. PSS값을 비교해서 PSS값이 클수록 linked list의 front에, PSS값이 작을수록 linked list의 tail에 가깝게 삽입된다. 이는 process 정보를 PSS값에 의해 내림차순 정렬되게 출력하기 위한 것이다.

```c
while(cur!=NULL){
        p=cur->p;

        seq_printf(m, "%5u ", p->pid);  // print PID
        seq_printf(m, "%17s ", p->comm); // print Process_NAME
        seq_printf(m, "%9luK ", cur->vss>>10); // print VSS
        seq_printf(m, "%9luK ", cur->rss>>10); // print RSS
        seq_printf(m, "%9luK ", cur->uss>>10); // print USS
        seq_printf(m, "%9lluK\n", cur->pss>>(10+PSS_SHIFT)); // print PSS

        cur=cur->next;
}

return 0;
```

- linked list를 순회하며 프로세스 별 정보를 출력한다. 기존의 procrank는 vss, rss, uss, pss 출력단위가 KB이기 때문에 이와 맞춰주었다. kernel thread의 경우에는 pid, process name 외에 vss, rss, uss, pss는 모두 0K라고 출력될 것이다.

3. Experiment 1 (VSS and RSS)

Test 1> $./mem 1

PID	Process_NAME	VSS	RSS	USS	PSS
2632	mem	5376K	2376K	1112K	1125K

Test 2> $./mem 5

PID	Process_NAME	VSS	RSS	USS	PSS
2616	mem	9472K	6524K	5208K	5223K

Test 3> $./mem2 1

PID	Process_NAME	VSS	RSS	USS	PSS
2751	mem2	5376K	1900K	596K	610K

Test 4> $./mem2 5

PID	Process_NAME	VSS	RSS	USS	PSS
2763	mem2	9472K	3920K	2648K	2662K

RSS=VSS-(memory that has been allocated but never touched)

./mem 1과 ./mem2 1는 **VSS**가 5376K로 **동일하다.**

./mem 5와 ./mem2 5는 **VSS**가 9472K로 **동일하다.**

많은 메모리를 할당할수록 VSS, RSS가 크다.

메모리 접근

```
while (1) {
    x[i++] += 1; // main work of loop done here.

    // if we've gone through the whole loop, reset a bunch of stuff
    // and then (perhaps) print out some statistics.
    if (i == num_ints) {
        double delta_time = Time_GetSeconds() - t;
        time_since_last_print += delta_time;
        if (time_since_last_print >= 0.2) { // only print every .2 seconds
            printf("loop %d in %.2f ms (bandwidth: %.2f MB/s)\n",
                   loop_count, 1000 * delta_time,
                   size_in_bytes / (1024.0*1024.0*delta_time));
            time_since_last_print = 0;
        }

        i = 0;
        t = Time_GetSeconds();
        loop_count++;
    }
}
```

mem.c

메모리 접근

```
while (1) {
    x[i++] += 1; // main work of loop done here.

    // if we've gone through the whole loop, reset a bunch of stuff
    // and then (perhaps) print out some statistics.
    if (i == num_ints/2) {
        double delta_time = Time_GetSeconds() - t;
        time_since_last_print += delta_time;
        if (time_since_last_print >= 0.2) { // only print every .2 seconds
            printf("loop %d in %.2f ms (bandwidth: %.2f MB/s)\n",
                   loop_count, 1000 * delta_time,
                   size_in_bytes / (1024.0*1024.0*delta_time));
            time_since_last_print = 0;
        }

        i = 0;
        t = Time_GetSeconds();
        loop_count++;
    }
}
```

mem2.c

mem.c의 경우 i를 증가시키며 메모리에 접근하다가 i==num_ints일 때 다시 i를 0으로 초기화한다. mem2.c의 경우 i를 증가시키며 메모리에 접근하다가 i==num_ints/2일 때 다시 i를 0으로 초기화한다. 따라서 mem2는 mem보다 할당되었지만 접근되지 않은 메모리(memory that has been allocated but never touched)가 크다. ∴ **RSS : mem2 < mem**

4. Experiment 2 (USS and PSS)

Test 1> $./mem 1 & ./mem2 1

PID	Process_NAME	VSS	RSS	USS	PSS
4053	mem	5376K	2392K	1112K	1125K
4054	mem2	5376K	1828K	600K	612K

Test 2> $./mem 5 & ./mem2 5

PID	Process_NAME	VSS	RSS	USS	PSS
4104	mem	9472K	6488K	5208K	5221K
4105	mem2	9472K	3864K	2648K	2660K

A = private memory that is mapped to physical pages of RAM.
B = shared memory that is mapped and is shared by one or more other processes
n = number of processes sharing

RSS = A + B
USS = A
PSS = A + B / n

PSS는 공유되는 메모리 크기를 공유 프로세스의 수로 나누어서 좀 더 정확한 메모리 사용량을 파악할 수 있게 해준다. 따라서 프로세스가 사용하는 실제 메모리 크기에 가장 근접한 값이라고 볼 수 있다.

먼저, 많은 메모리를 할당할수록 USS, PSS가 크다.
위의 mem.c, mem2.c 코드에서 볼 수 있듯 mem.c는 i를 증가시키며 메모리에 접근하다가 i==num_ints일 때 다시 i를 0으로 초기화한다. mem2.c의 경우 i를 증가시키며 메모리에 접근하다가 i==num_ints/2일 때 다시 i를 0으로 초기화한다.
따라서 실제로 접근된 메모리 크기는 mem이 mem2의 약 2배이다.
결과에서 mem의 USS, PSS는 mem2의 USS, PSS의 대략 2배 정도 되는 것을 확인할 수 있다. (RSS의 경우, mem 결과가 mem2 결과의 약 2배가 되지 않는다. 따라서 PSS로 더 정확한 메모리 사용량을 파악할 수 있음을 확인할 수 있다.)
또한 PSS가 USS보다 약간 더 큰데 그 차이가 크지 않음을 확인할 수 있다.
RSS(A+B), PSS(A+B/n), USS(A)을 이용해 계산해보면 대략 n=100개의 프로세스가 메모리를 공유하고 있음을 알 수 있다.

5. Experiment 3 (Shared Library)

Test 1> $./mem 1 & ./mem2 1

PID	Process_NAME	VSS	RSS	USS	PSS
1769	mem	5376K	2392K	1156K	1697K
1770	mem2	5376K	1856K	620K	1161K

Test 2> $./mem 5 & ./mem2 5

PID	Process_NAME	VSS	RSS	USS	PSS
1847	mem	9472K	6464K	5344K	5827K
1848	mem2	9472K	3936K	2816K	3299K

A = private memory that is mapped to physical pages of RAM.
B = shared memory that is mapped and is shared by one or more other processes
n = number of processes sharing

RSS = A + B
USS = A
PSS = A + B / n

먼저, Experiment 2와 동일하게 많은 메모리를 할당할수록 USS, PSS가 크다.
Shared Library를 사용하는 경우에는 USS, PSS에 대해 이전과는 다른 결과를 얻을 수 있다.
PSS가 USS보다 큰데, 그 차이가 Shared Library를 사용하지 않을 때보다 훨씬 크다.
USS(RSS-shared memory)는 shared memory를 포함하지 않으므로 Shared Library를 사용하지 않을 때보다 약간 더 큰 값이 나온다.
RSS(A+B)는 Shared Library를 사용할 때와 사용하지 않을 때가 거의 똑같은 값을 가지지만, USS(A)는 약간 증가한 값을 보이므로 shared memory 크기가 조금 감소했음을 예측할 수 있다.
또한, PSS(A+B/n)는 Shared Library를 사용하지 않을 때보다 크게 증가한 것을 볼 때, 메모리를 공유하고 있는 프로세스 수(number of processes sharing)가 크게 감소했음을 예측할 수 있다.
RSS(A+B), PSS(A+B/n), USS(A)을 이용해 계산해보면 대략 n=2개의 프로세스가 메모리를 공유하고 있음을 알 수 있다. Shared Library를 사용하지 않았을 때 대략 n=100개의 프로세스가 메모리를 공유하고 있었던 것에 비해 매우 크게 감소했다.