CS 251/EE 255: Real-Time Embedded Systems
University of California, Riverside

HW Project #3: Task Allocation and Virtual Memory Management

Due: 3/12/2023, Sunday, 11:59 p.m. (Pacific time)

Read the entire handout carefully before starting work. Good luck and happy kernel hacking!

1. Introduction

This homework project is designed to exercise the following topics:

- Task allocation heuristics
- Memory locking in user-space applications
- Memory descriptor and memory areas

Before you start working:

- Read the handout carefully: background, writeup questions and the tips section aim to give you hints about design and implementation.
- Whenever the handout asks you to do something without explaining how to accomplish it, please consult references on kernel development. That's by design.
- To be awarded full credit, your kernel must ***not*** crash or freeze under any circumstances.

2. Background Information

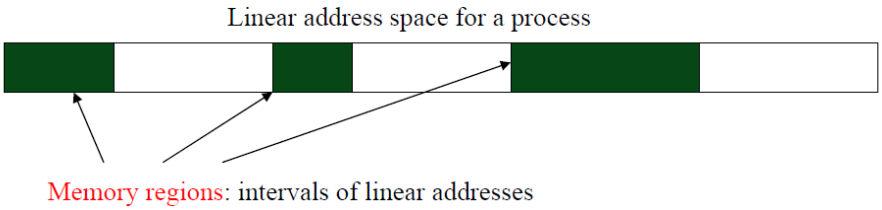
In the Linux kernel, each virtual address space of a process is managed by a memory descriptor, `struct mm_struct`, defined in `include/linux/mm_types.h`. Each `struct task_struct` has a pointer to one active memory descriptor (`struct mm_struct *mm` field), and one memory descriptor may be shared among tasks that belong to the same virtual address space (e.g., threads of the same process). Note that all normal user-level processes/threads have a non-NULL `mm` field in their TCBs, but kernel threads have `mm == NULL` since they do not own a separate virtual address space.

A memory descriptor contains several items related to the virtual address space management. See below:

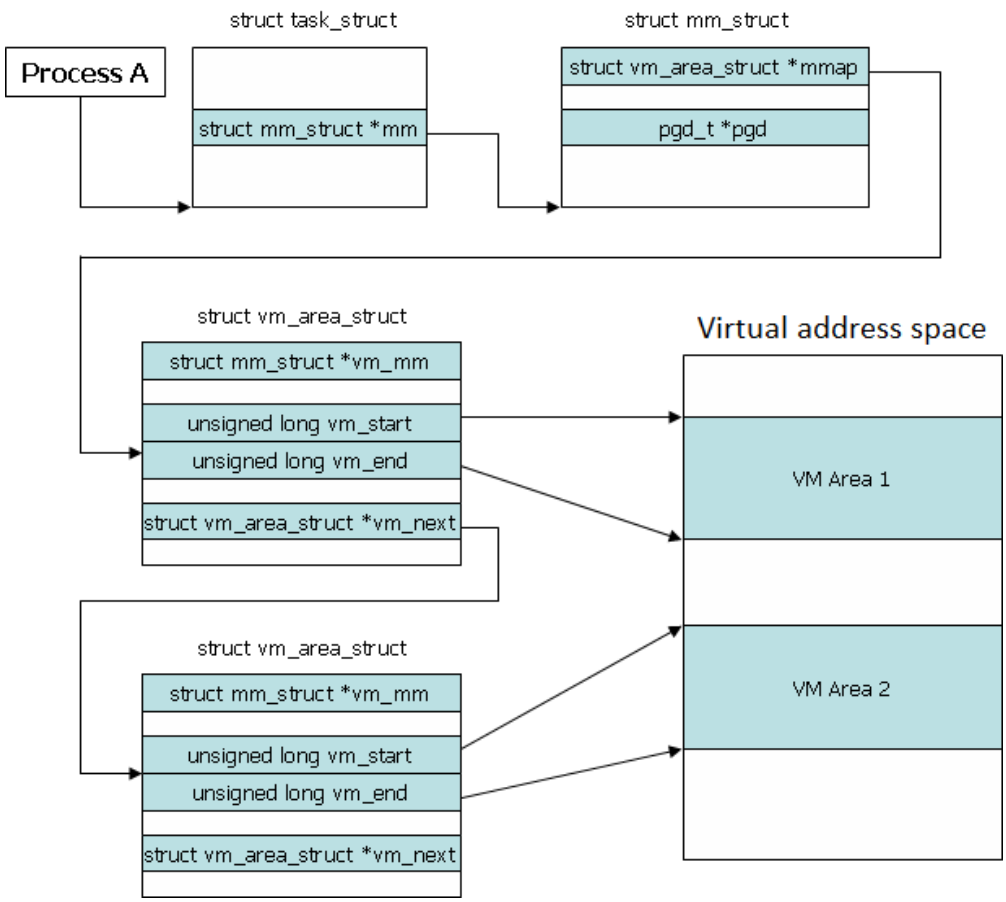
```
struct mm_struct {
    struct vm_area_struct *mmap;      /* list of virtual memory areas (VMAs) */
    ...
    pgd_t * pgd;                      /* start address of the page directory table */
    atomic_t mm_users;                /* How many users with user space? */
    atomic_t mm_count;                /* How many references to "struct mm_struct" */
    ...
    int map_count;                    /* number of VMAs */

    spinlock_t page_table_lock;       /* Protects page tables and some counters */
    struct rw_semaphore mmap_sem;

    ...
    unsigned long total_vm;         /* Total pages mapped */
    unsigned long locked_vm;        /* Pages that have PG_mlocked set */
    ...
```

```
    unsigned long start_code, end_code, start_data, end_data; /* VM segment info */
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    ...
```

One of the main item is **`pgd_t *pgd`**, which stores the pointer to the page directory table (the first-level page table in the Linux's multi-level paging scheme) that is needed to convert a virtual address to a physical address.



Another important field is **`struct vm_area_struct *mmap`**, which is used to maintain the memory areas of a virtual address space. As can be seen in the above figure, a virtual (logical) address space of a process consists of multiple memory areas (regions), and each memory area is characterized by start/end addresses and some access rights (e.g., shared/private, read/write/execute). Code segments and data segments are examples of such memory areas. `struct vm_area_struct` is a kernel data structure to manage each memory area. The `struct vm_area_struct *mmap` field of `struct mm_struct` is a pointer to the first memory area of the virtual address space, and it has a pointer (`vm_next`) to the next `struct vm_area_struct`. Below is a pictorial example of `task_struct`, `mm_struct`, and `vm_area_struct`.



3. Build Directory

You need to create a subfolder `proj3/` in the top-level directory of your repository. Your source files to be implemented will fall into three categories:

Category	Location in kernel repo	Build method
kernel modules	`proj3/modules`	standalone using the kernel build system
user-space applications	`proj3/apps`	standalone using user-space libraries

Follow the instructions in the HW #1 handout to setup Makefile.

4. Assignment

4.1. Multiprocessor task allocation (30 points)

Source code location: `<your_repo>/proj3/apps/task_alloc/task_alloc.c`

Write a **user-space** program that determines task-to-CPU allocation for partitioned scheduling. The program should support two bin-packing heuristics: **BFW, WFD, and FFD variants** – start with *B* open bins. Assume **EDF** for task scheduling on each CPU (i.e., you can use up to 100% utilization per CPU). This program takes as input a comma-separated values (CSV) file in the following format:

input1.txt

```
1  4,FFD        # number of open bins, heuristic policy (BFD, WFD, or FFD)
2  6            # number of tasks
3  t1,4,10      # task name, C, T
4  t2,6,14
5  t3,4,16
6  t4,4,16
7  t5,3,10
8  t6,3,10
```

In the above, the first line has two parameters: the number of CPUs (*B*) and the heuristic policy to use. The second line has the number of tasks (*N*) to be allocated to *B* CPUs. Starting from the third line, each task's name (string, not exceeding 20 characters), C (unsigned integer > 0), and T (unsigned integer > 0) values are given.

Your program then performs task allocation and prints the results in a CSV format to terminal. For example, for the above input:

```
1  $ ./task_alloc input1.txt
2  Success
3  CPU0,t1,t2
4  CPU1,t3,t6,t5
5  CPU2,t4
6  CPU3
7  $
```

The first line of output should print "**Success**" if all tasks could be allocated. Then it prints *B* lines, where each line corresponds to each CPU and shows the names of tasks allocated to it. If a CPU has no task allocated to it, your program just prints the CPU's name (CPU3 in the above example).

If not all given tasks could be allocated to *B* CPUs, the program should print only "**Failure**". Do not print intermediate allocation results in case of failure.

The input file can have *B* (number of CPUs) up to 100 and *N* (number of tasks) up to 10000.

4.2. Memory access time (15 points)

4.2.1. User-level memory access time check (5 points)

Source code location: `<your_repo>/proj3/apps/mem_alloc/mem_alloc.c`

Write a **user-space** program that dynamically allocates a given amount of memory. The memory allocation size is given as an input parameter of the program. For example,

```
$ ./mem_alloc 100000000
```

allocates 100,000,000 bytes of memory. You can use `malloc` or similar API in this program.

After memory allocation, the program accesses the allocated memory space with a stride of one page size (by writing only one byte in each page). The program prints out the amount of time consumed to access the allocated memory space, and finally, it pauses. The pseudocode of the `mem_alloc` program is as follows:

```
int main() {
    int n = memory size to be allocated /* parsed from the input */
    char *buf = allocate n bytes of memory

    /* check memory access time */
    t1 = get current time
    for (i = 0; i < n; i += 4096) /* write one byte in each page (4096 bytes) */
        buf[i] = 1;
    t2 = get current time

    print current PID and the total memory access time (t2 – t1) in nsec

    pause() /* wait here; can be terminated by 'Ctrl-C' */
}
```

Use `clock_gettime` with `CLOCK_MONOTONIC` to obtain the current time stamp. The access time should be printed in **nanosecond resolution** (e.g., "12345678 ns").

Example:

```
1  $ ./mem_alloc 10000
2  PID 3705, 1692 ns
3  ^C
4  $ ./mem_alloc 10000000
5  PID 3706, 3905759 ns
6  ^C
7  $
```

In the above example, `^C` means Ctrl-C.

4.2.2. Memory locking (10 points)

Source code location: `<your_repo>/proj3/apps/mem_alloc_lock/mem_alloc_lock.c`

Write a **user-space** program `mem_alloc_lock`. This program is similar to `mem_alloc`, but it **locks the allocated memory** into RAM **before** checking the memory access time. Use `mlock` for memory locking. Print an error message if mlock fails.

Note: `mlock` of a large amount of memory (e.g., 100MB) may fail for normal users (non sudoers) depending on system configurations. It is safe to assume that the user will always run this program with sudo (e.g., `$ sudo ./mem_alloc_lock 100000000`)

Example:

```
1  $ ./mem_alloc_lock 10000
```

```
PID 3722, 229 ns
^C
$ ./mem_alloc_lock 10000000
PID 3724, 39759 ns
^C
$ ./mem_alloc_lock 100000000
./mem_alloc_lock: mlockall failed
$ sudo ./mem_alloc_lock 100000000
[sudo] password for ubuntu:
PID 3843, 515890 ns
^C
$
```

4.3. Process memory viewer (45 points)

This part implements two virtual drivers as **LKMs** (4.3.1 & 4.3.2). They both do ***not*** need an inclusion of any new data field in the existing memory-related data structures or TCB. In other words, these LKMs should work without recompiling the kernel image.

4.3.1. Process memory segment (15 points)

Source code location: `<your_repo>/proj3/modules/segment_info/segment_info.c`

Write a misc device LKM that creates `/dev/segment_info` upon loaded. When a user writes PID onto this device file, your device prints the start and end addresses and the size of the code and data segments of the corresponding process to the kernel log. Below shows an example:

```
$ sudo su                           # change to root user
# echo 4142 > /dev/segment_info     # write PID 4142 to the device file
# dmesg
...
[79991.234344] [Memory segment addresses of process 4142]
[79991.234347] 56100f9d7000 - 56100f9d73c5: code segment (965 bytes)
[79991.234351] 56100f9d9d88 - 56100f9da010: data segment (648 bytes)
```

Lines 5-7 are the LKM output. Line 6 and 7 print the start and end addresses of each segment, along with its size (end address - start address). The addresses should be printed in hexadecimal, but the size in decimal.

The maximum length of input written to the device at a time does not exceed 20 characters. In case of error such as the input longer than 20 characters or a process with the given PID cannot be found, your device prints a single-line kernel message: `segment_info: error` (if you want, it is ok to append additional information after this string).

Notes:

- To get PID from the user, you need to implement the `write` operator of the misc device. See the tips section.
- Use the VM segment info variables already defined in the process's memory descriptor (`struct mm_struct`). No need to traverse the list of memory areas here.
- Look into `include/linux/mm_types.h` for the definition of `struct mm_struct`.
- Use "%08lx" in printk to display 64bit memory addresses as a zero-filled hexadecimal.

4.3.2. Memory map of a process (30 points)

Source code location: `<your_repo>/proj3/modules/vm_areas/vm_areas.c`

Write a misc device LKM that creates `/dev/vm_areas` upon loaded. When a user writes PID onto this device file, your device iterates over the entire list of memory areas for the process and prints the start and end addresses of each area and the size of the area to the kernel log. In addition, if the memory area is a locked area, print "`[L]`" to indicate it.

An example of the output printed to the kernel log is as follows:

```
1  $ sudo su                          # change to root user
2  # echo 4603 > /dev/vm_areas        # write PID 4142 to the device file
3  # dmesg
4  ...
5  [83077.101002] [Memory-mapped areas of process 4675]
6  [83077.101009] 565244efe000 - 565244eff000: 4096 bytes
7  [83077.101011] 565244eff000 - 565244f00000: 4096 bytes
8  [83077.101013] 565244f00000 - 565244f01000: 4096 bytes
9  [83077.101014] 565244f01000 - 565244f02000: 4096 bytes
10 [83077.101016] 565244f02000 - 565244f03000: 4096 bytes
11 [83077.101018] 565244f03000 - 565244f24000: 135168 bytes
12 [83077.101101] 7fa7c9f39000 - 7fa7ca8c3000: 10002432 bytes [L]
13 [83077.101103] 7fa7ca8c3000 - 7fa7ca8e5000: 139264 bytes
14 [83077.101117] 7fa7ca8e5000 - 7fa7caa5d000: 1540096 bytes
15 ...
```

In this example, the virtual memory area `7fa7c9f39000 - 7fa7ca8c3000` (line 12) has '[L]' since it is locked to physical memory.

Similar to the previous device, the maximum length of input written to the device at a time does not exceed 20 characters. In case of error such as the input longer than 20 characters or a process with the given PID cannot be found, your device prints a single-line kernel message: `vm_areas: error` (if you want, it is ok to append additional information after this string).

Notes:

- See `include/linux/mm_types.h` for the definition of the memory area structure (`struct vm_area_struct`), and `include/linux/mm.h` for the meaning of `vm_flags` in `struct vm_area_struct`.

4.3.3. Resident pages in memory areas (bonus: 5 points)

Extend the `vm_areas` LKM such that, for each memory area, it also prints the number of pages that are resident in the physical memory (RAM). For example:

```
[Memory-mapped areas of process 2367]
561eb1324000 - 561eb1325000: 4096 bytes, 1 pages in physical memory
561eb1325000 - 561eb1326000: 4096 bytes, 1 pages in physical memory
561eb1326000 - 561eb1327000: 4096 bytes, 1 pages in physical memory
561eb1327000 - 561eb1328000: 4096 bytes, 1 pages in physical memory
561eb1328000 - 561eb1329000: 4096 bytes, 1 pages in physical memory
561eb25f3000 - 561eb260c000: 102400 bytes [L], 25 pages in physical memory
561eb260c000 - 561eb2614000: 32768 bytes, 0 pages in physical memory
...
```

If a page resides in the physical memory, the corresponding page table entry (PTE) is valid and has the physical page number. Recall that Linux uses multi-level page tables. Therefore, to check the PTE, you will need to walk through multiple page table entries (i.e., pgd, p4d, pud, pmd, pte). Kernel functions like `follow_pte` (which actually calls `follow_invalidate_pte`) in `mm/memory.c` would be a good reference (ignore `pmd_huge` as our kernel does not use it). To implement this part, your code would need to include `linux/mm.h`.

4.6. Writeup (10 points)

Writeup location: `<your_repo>/proj3/proj3_team00.pdf` or `proj3_team00.txt`

Submit a **plain text** or **PDF** document with brief answers to the following questions:

1. Report the memory access time of `mem_alloc` and `mem_alloc_lock` (assignment 4.2.1 & 4.2.2) with the memory size of 1 MB, 10 MB, and 100MB. For each case, run at least 10 times and provide average memory access. Give a brief discussion comparing the results.
2. Report the kernel logs of `/dev/segment_info` (assignment 4.3.1) for `mem_alloc_lock` (assignment 4.2.2) with the memory size of 1 KB and 100MB.
3. Report the kernel logs of `/dev/how_vm_areas` (assignment 4.3.2) for `mem_alloc_lock` (assignment 4.2.2) with the memory size of 1 KB and 100MB.
4. If the OS kernel uses virtual memory with demand paging but does not provide mlock-like functions, then what would be a workaround that a user-level program can do to mitigate unpredictable memory access delay at runtime? Recall lecture slides.
5. Give a brief summary of the contributions of each member.

5. How to submit?

Follow the instructions in the HW #1 handout.

6. Tips

How to get PID from user in the misc device? How to implement the `write` operator?

- The write operator function can be defined in the following format:
  `ssize_t write_operator(struct file *filp, const char __user *ubuf, size_t count, loff_t *f_pos)`
- The second parameter, `ubuf`, is the pointer to input string written to the device, and the third parameter, `count`, is the length of `ubuf`. You don't need to consider the first and the last arguments in this assignment.
- `ubuf` points to the data in the user space. So, you need to use `copy_from_user()` to copy `ubuf` to another buffer in the kernel space before reading the data. For instance, if your function a local buffer `char kbuf[20]`, do `copy_from_user(kbuf, ubuf, count)`. To avoid buffer overflow, make sure `count` does not exceed your buffer size.
- On success, the write operator function should return `count` on success (telling the kernel that your device finished reading the given `count` bytes). You can return -1 when there is an error.

How to convert a string to integer in the misc driver LKM?

- Use two kernel library functions: `strstrip()` and `kstrtoint()`
- Example (assuming `kbuf` has a character string in it):

```
const char *p;
int val;
p = strstrip(kbuf);  // strstrip() removes unnecessary empty spaces and line breaks;
                     // returns the starting address of the cleaned-up string
if (kstrtoint(p, 0, &val)) { // kstrtoint() converts p to int and stores in val
    printk("error\n");
    return -1;
}
```

How to find a process's TCB by PID in LKM? I cannot use `find_task_by_pid_ns()` in LKM.

- Use `pid_task(find_vpid(pid), PIDTYPE_PID)`, which returns a pointer to struct task_struct on success, and NULL on failure.

How can I verify if `/dev/vm_areas` is working correctly?

- Use the command line tool, `pmap` (e.g., `$ sudo pmap -x <pid>`). The "Rss" field of output means Residential Set Size (memory residing in physical memory) in KB (1 KB = 1024 bytes). So, 4 and 12 in Rss mean 1 and 3 pages in physical memory, respectively.

References

1. Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," O'Reilly Media, ISBN: 978-0596005658, 2005 (3rd Edition). – **Chapter 9.**
2. Robert Love, "Linux Kernel Development," Addison-Wesley Professional, ISBN: 978-0672329463, 2010 (3rd Edition). – **Chapter 15.**