Project 3: Memory Management

CSE 330: Operating Systems - Fall 2023

Due by **28th November, 11:59 pm**

Summary

To support multiprogramming, modern OSes present virtual memory to processes so that each process sees a contiguous logical memory address space which may not be contiguous in physical memory and may not be completely in memory. The OS maintains the mapping between a process' virtual memory and the computer's physical memory through the process' page table. Every page table entry stores the mapping information of a virtual page, indicating whether the page is present in memory and, if yes, the physical frame number it is mapped to. Additional data structures track the temporarily swapped pages from memory to disk.

In Project 3, we will implement new kernel functions to reveal the "magic" that the kernel does to virtualize memories, using the same virtual machine environment you prepared in the previous projects. This project will help you understand how a real-world OS like Linux performs memory management and master the skills to implement it in kernel space

Description

In this project; you will implement a kernel module to walk the page tables of a given process and find out how many of the process' pages are present in the physical memory (resident set size--RSS), how many are swapped out to disk (swap size--SWAP), and how many pages are in the process working set (working set size--WSS).

1. Module interface and module_init

You must name your module "memory_manager". It takes a process ID as the only command-line input argument. Name your input argument "**pid**". Like Project 2, you will use the module_param() macro to pass the input argument to your kernel module. Call this macro at the beginning of your module code.

```
module_param(name, type, perm) /* macro for module command line
parameters. name is the name of the parameter, type is the type of
the parameter, and perm sets the visibility in sysfs. For example,
module_param(buff_size, int, 0) defines an input argument named
buffer_size, type is int, and the default value is 0.*/
```

Reference on passing command line arguments to a kernel module.

- Passing data to a kernel module – module_param

2. Traverse Memory regions

Linux organises process memory using the `mm_struct`data structure, which is a member of `task_struct`. It contains information regarding all memory regions for that process, organised as a list, `mmap`. Memory regions are the kernel representation of allocated address intervals, characterised by a starting address, a length, and access rights. Each memory region is described by a `vm_area_struct` (VMA). Each VMA contains a starting (`vm_start`) and ending address (`vm_end`) for its region. The general organisation of these structures is shown in Figure 1. In your code, you must go through each VMA, and within each VMA, you must check each page. Make sure to loop through this address range by `PAGE_SIZE`.

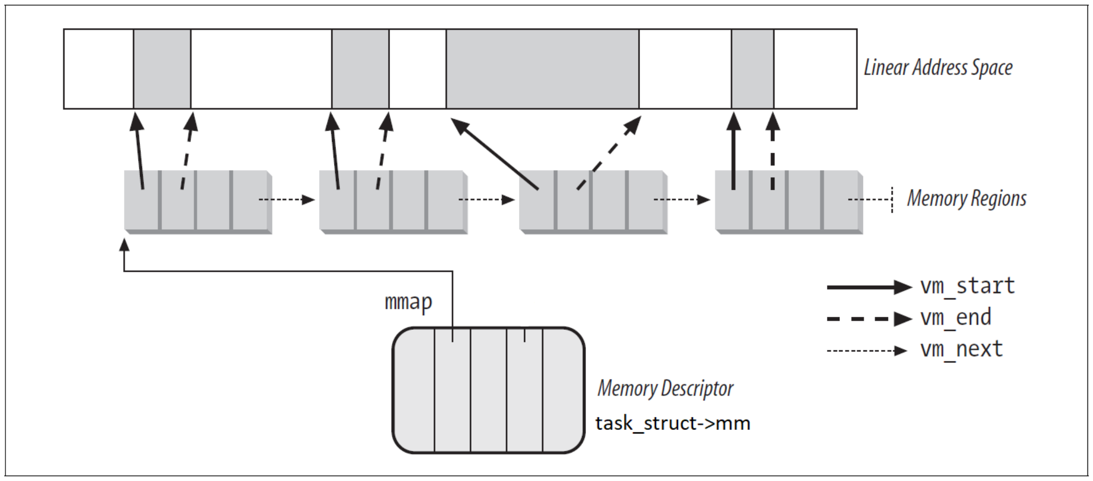

Figure 1

```
task_struct *task;
task->mm->mmap
/* the list of VMAs. struct mm_struct is located in
include/linux/mm_types.h */
```

3. Walk Page Tables

For each valid page in the given process' address space; you need to walk the process' page tables to find out if the page is present in physical memory or in the swap.

The Linux kernel (5.16) implements a 5-level page table with the following five levels: **PGD, P4D, PUD, PMD, and PTE** (see Figure 2). The following tables can be accessed in order, starting from the `mm_struct` of the process. The mm_struct can be found in the process `task_struct`.

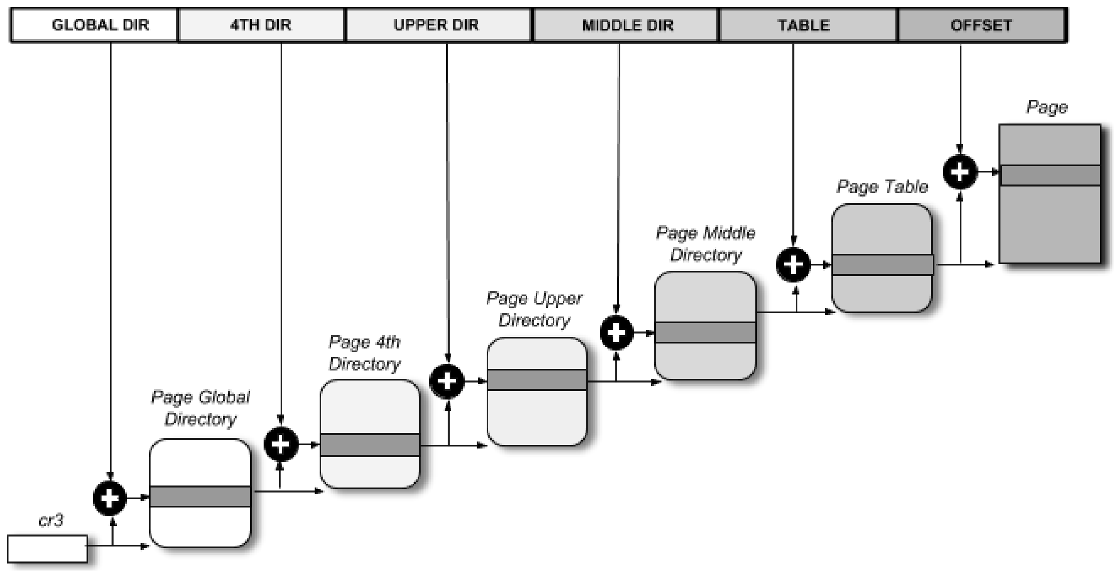

Figure 2

```
pgd_offset(mm, address) /* Return pointer to the PGD. mm is the mm_struct
of the process, address is the logical address in the virtual memory space*/

p4d_offset(pgd_t *pgd, unsigned long address) /* Return pointer to the
P4D. pgd is the pointer of PGD, address is the logical address in the virtual
memory space.*/

pud_offset(pgd_t *p4d, unsigned long address) /* Return pointer to the
PUD. p4d is the pointer of P4D, address is the logical address in the virtual
memory space.*/

pmd_offset(pgd_t *pud, unsigned long address) /* Return pointer to the
PMD. pud is the pointer of PUD, address is the logical address in the virtual
memory space.*/

pte_offset_map(pgd_t *pmd, unsigned long address) /* Return pointer to
the PTE. pmd is the pointer of PMD, address is the logical address in the
virtual memory space*/
pte_present(pte) /* Return 1 if a page table entry is in the main memory,
pte is a pointer to the page table entry.*/

/*Required header file: <linux/mm.h>*/
```

A simple example of the page table walk is the function __follow_pte_pmd() in mm/memory.c

```
pgd_t *pgd;
p4d_t *p4d;
pmd_t *pmd;
pud_t *pud;
pte_t *ptep, pte;

pgd = pgd_offset(mm, address);        // get pgd from mm and the page address
if (pgd_none(*pgd) || pgd_bad(*pgd)) {
 // check if pgd is bad or does not exist
      return;
}
p4d = p4d_offset(pgd, address); //get p4d from from pgd and the page address
if (p4d_none(*p4d) || p4d_bad(*p4d)) {
// check if p4d is bad or does not exist
      return;
}
pud = pud_offset(p4d, address); // get pud from from p4d and the page address
if (pud_none(*pud) || pud_bad(*pud)) {
// check if pud is bad or does not exist
      return;
}
pmd = pmd_offset(pud, address); // get pmd from from pud and the page address
if (pmd_none(*pmd) || pmd_bad(*pmd)) {
// check if pmd is bad or does not exist
      return;
}
ptep = pte_offset_map(pmd, address); // get pte from pmd and the page address
if (!ptep){
// check if pte does not exist
       return;
}
pte = *ptep;
```

A page that is present in memory is part of the process' RSS. A page that is valid but not present in memory is in SWAP. While walking the page tables, count how many pages are in RSS and how many pages are in SWAP.

4. Measure Working set size

To measure the working set size (WSS), you will need to start a timer of 10 seconds and count the number of pages accessed during this time by checking the bit of every page table entry.

We can use the high-resolution timer (HRT) to implement this timer.

```
hrtimer_init (struct hrtimer *timer, clockid_t clock_id, enum
hrtimer_mode mode)
/*Initialize a timer to a given clock, timer is the timer to be initialized,
clock_id is the clock to be used, mode is the timer mode. HRTIMER_ABS mode
indicates that the timer is set using an absolute value. HRTIMER_REL
indicates that the time is set relative to the current time. We use
CLOCK_MONOTONIC for setting a clock that represents the elapsed time since
some fixed point in the past. We choose the CLOCK_MONOTONIC because the
elapsed time between two events will never be negative and the result will
not be affected by any changes in the system clock*

//Return the current monotonic time in ktime_t format.
ktime_get (void)

//Return a ktime struct set using the given seconds and nanoseconds.
ktime_set (const long secs, const unsigned nsecs)

/*Advance the timer's expiration time by the given interval from the current
time
*/hrtimer_forward (struct hrtimer *timer, ktime_t now, ktime_t
interval)

/* Start the timer that expires by the given time (set either by an absolute
value or relative to the current time, depending on the given mode) */
hrtimer_start (struct hrtimer *timer, ktime_t time, const enum
hrtimer_mode mode)

hrtimer_cancel(struct hrtimer * timer) // Cancel the given timer

/*Required header file: <linux/hrtimer.h>*/
```

The following example shows how to use a 10-second HR timer.

```
unsigned long timer_interval_ns = 10e9 // 10-second timer
enum hrtimer_restart no_restart_callback(struct hrtimer *timer)
{
    ktime_t currtime , interval;
    currtime  = ktime_get();
    interval  = ktime_set(0, timer_interval_ns);
    hrtimer_forward(timer, currtime , interval);
    // Do the measurement
    return HRTIMER_NORESTART;
}
```

To clear the accessed bit of a given page table entry, you can use `ptep_test_and_clear_young()`. This function first checks if the given `pte` was accessed and clears the accessed bit of this `pte` entry; it returns 1 if the `pte` was accessed.

```
int ptep_test_and_clear_young (struct vm_area_struct *vma, unsigned
long addr, pte_t *ptep)
/* Test and clear the accessed bit of a given pte entry. vma is the pointer
to the memory region, addr is the address of the page, and ptep is a pointer
to a pte. It returns 1 if the pte was accessed, or 0 if not accessed. */

/* The ptep_test_and_clear_young() is architecture dependent and is not
exported to be used in a kernel module. You will need to add its
implementation as follows to your kernel module. */
int ptep_test_and_clear_young(struct vm_area_struct *vma,
                              unsigned long addr, pte_t *ptep)
{
      int ret = 0;
      if (pte_young(*ptep))
            ret = test_and_clear_bit(_PAGE_BIT_ACCESSED,
                                     (unsigned long *) &ptep->pte);
      return ret;
}
```

5. Measure resident set size, swap size, and working set size periodically.

To invoke the HRT periodically, you only need to change the return value of the timer callback function from `HRTIMER_NORESTART` to `HRTIMER_RESTART`, as illustrated in the below example.

```
unsigned long timer_interval_ns = 10e9; // 10-second timer
enum hrtimer_restart no_restart_callback(struct hrtimer *timer)
{
    ktime_t currtime , interval;
    currtime  = ktime_get();
    interval  = ktime_set(0, timer_interval_ns);
    hrtimer_forward(timer, currtime , interval);
    // Do the measurement
    return HRTIMER_RESTART; // Use HRTIMER_RESTART to invoke the HRT
periodically
}
```

You will use your timer to measure the given process' resident set size, swap size, and working set size periodically (every 10 seconds) for several minutes to observe how these different memory usages change over time.

For a given PID, your kernel module should print the statistics in the provided format below.

```
PID [PID]: RSS=[RSS] KB, SWAP=[SWAP] KB, WSS=[WSS] KB
```

For example:

```
PID 8975: RSS=1296 KB, SWAP=1296 KB,WSS=1296 KB
…
```

6. Testing

Test your implementation thoroughly. To trigger swap during intensive test cases, you **MUST** set your **VM to use 4 GB memory or less**. Your kernel module should meet the following three criteria.

- Module should be loaded and unloaded successfully
- The RSS, WSS, and SWAP results from your kernel module output should match the expected values
- Use the following link to test your kernel module

Submission Requirements & Guidelines

Project 3 is **due** in four weeks by **November 28th, 2023 11:59 pm**. Submit the project work following the below guidelines

1. Only one submission is required per group.

   2.1. Source code (Only the Kernel Module. No need to submit the Makefile if you use the one provided in the testing module.)

   2.2. README file, listing the following:

   a. Full names of your group members

   b. Video link explaining your solution to the problem. How did you approach this problem? How long did it take for your group to agree on a solution? How long did it take to implement your solution? Explain

3. the relevant steps your group took to come up with a solution.
4. **Do not** submit any binary
5. Create a .zip file with all your submission files. Name the zip file following the below-naming convention. **"Project-3-Group-<GroupNo>.zip"**
   Example: If you belong to Group-1, the name of your zip file should be **Project-3-Group-1.zip**
6. Individual group members MUST make reasonable contributions to ensure their ideas and solutions to the project are marked.
   Failure to do so & you will be awarded in 0-grade points for the Project.

Bonus !!

There is an opportunity for you to earn a total of "**5**" bonus grade points in this project. If you submit the completed project **a week before the deadline**. If you make any submission after **"21st November 2023 at 11:59 PM",**; you are **NOT** eligible for these bonus points.

Policies

1. Late submissions will *absolutely not* be graded (unless you have verifiable proof of emergency). It is much better to submit partial work on time and get partial credit for your work than to submit late for no credit.
2. Every group needs to *work independently* on this exercise. We encourage high-level discussions among students to help each other understand the concepts and principles. However, a code-level discussion is prohibited, and plagiarism will directly lead to the failure of this course. We will use anti-plagiarism tools to detect violations of this policy.