

CSIS0230A Principle of Operating Systems (Class A)

Assignment 3

Finding shared pages among processes

Tutor: mhzhang@csis.hku.hk , Deadline 5pm Nov 23, 2003.

During the tutorials, we noticed that many memory pages in Linux are shared. In this assignment we want to answer the following question efficiently: given a set of processes, what frames are they sharing, and what each process can do to each of them?

We need to add a system call, with a hybrid approach: the system call is implemented by in the kernel, but it only calls to a function to be defined by a kernel module. In principle the work should be the same for all 2.4 kernels. However, we suggest using Linux stock kernel 2.4.22 or the Redhat-modified version 2.4.20-8, since the TA will use them to run the code you submit.

1. The system call to implement: *get_shared_frames*

int *get_shared_frames*(*pid_t pid*, **unsigned long** **mm*, **struct frame_record** **frs*, **int** *size*)

Find all pages within the address space of the process *pid* whose frame has a *count* (in the frame table) that indicate sharing **with other process page(s)**¹. An array of *size* elements is given in *frs* for returning such information. Each array element has the following type.

```
struct frame_record {  
    unsigned long    laddr;  
    int              region_flags;  
    unsigned long    pte;  
    short            count;  
    short            buffers;  
};
```

The fields has the following meaning:

- *laddr*: the virtual address of the page.
- *region_flags*: the *vm_flags* of the memory region including the virtual address.
- *pte*: the full 32-bit page table entry which include the physical frame number and the frame permission.
- *count*: the number of other process pages sharing the frame.
- *buffers*: 1 if the frame holds a disk block in the page cache, 0 if otherwise.

The (virtual) address of the *mm* structure of the process is also returned, through the *mm* argument. If the process has no *mm*, the system call returns 0. Otherwise, it returns the number of frames that are found to be shared. This number can be larger than *size*, in which case only the first *size* such frames are recorded in *frs*.

The system call may cause these errors:

- *ESRCH*: no such process. The process named by *pid* does not exist.
- *EFAULT*: bad address. A buffer provided for storing values is not writable.

¹The information is obtained by the *count* within the frame table. By reading the header files, you'd understand that it might happen that the *count* is slightly larger than what it should be, and there is no way compensate for it. Luckily, the slightly larger *count* only slightly reduce the efficiency of the program with minimal effects elsewhere.

For multi-processor safety, you should lock the semaphore of the *mm* in the *task_struct* during most of the time of a system call. You can use *down_read* and *up_read* to perform a read-lock for the read-write semaphore of the *mm* (*mm->mmap_sem*).

The system call should be efficient. Avoid making the same translation again and again from the PGD down to the frame, and don't scan pages with no virtual memory mappings.

2. Hybrid system call support

In the not so distant past, the kernel allows modules to install a system call like this:

```
#define __NR_my_syscall 249
extern long sys_call_table[];
asm linkage int sys_my_syscall(void) { /* implementation here */ }
int mymod_init(void) { sys_call_table[__NR_my_syscall] = sys_my_syscall; }
void mymod_exit(void) { sys_call_table[__NR_my_syscall] = sys_ni_syscall; }
```

But there is a subtle race condition: what would happen if some process makes the system call, and just after the system call starts to execute, the user unload the module? The kernel could then crash. Even worse, locking cannot prevent this race. Added to other political reasons¹, the kernel developers close this race by altogether hiding *sys_call_table* from modules.

So to add a system call we must modify the kernel. But it is inconvenient to have to wait a reboot whenever we modify the system call implementation slightly. Luckily, nobody says that a system call cannot depend on a module. E.g., the kernel might contain:

```
int default_my_syscall(void) {
    return -ENOSYS;
}
typedef int (*my_syscall_t)(void); // must match the actual type
static my_syscall_t my_syscall = default_my_syscall;
asm linkage int sys_my_syscall(void) {
    my_syscall(); // call the function in the my_syscall function pointer
}
int register_my_syscall(int (*new_my_syscall)(void)) {
    // Should check whether my_syscall is unused first
    my_syscall = new_my_syscall;
}
int unregister_my_syscall(int (*new_my_syscall)(void)) {
    // Should check whether my_syscall is overwritten first
    my_syscall = default_my_syscall;
}
EXPORT_SYMBOL(register_my_syscall); /* need <module.h> */
EXPORT_SYMBOL(unregister_my_syscall);
```

In other words, we can have a function pointer *my_syscall* that points to the real system call implementation. By default this pointer points to a function that simply returns the “Function not implemented” error, while we provide (export) a function *register_my_syscall* for modules to modify this pointer to its own implementation.

¹Kernel developers in general think that binary module providers should stick to the interface provided by the kernel and should not add system calls.

Use this strategy to allow the module to provide a new system call. Also perform proper locking (using a semaphore) to prevent the above race condition.

3. The user program: `shared_frames`

Write a user program `shared_frames` (using C or C++) that utilizes the system call described above to find all frames shared among a set of processes. It should be invoked as `shared_frames pid1 pid2 pid3 ...`, where `pidN` is the PID of a running process. The program should show a map like this:

```
> shared_frames 2 17625 17698 17636 17667 17626 17627 17670
Process 2 has no memory map.
Process 17626 shares memory map with 17625, skipped.
Process 17670 does not exist.
Frame at 12a000: in page cache and shared by 5 process pages:
    Process 17625 at location 40010000 (copy-on-write)
    Process 17667 at location 40010000 (copy-on-write)
    Process 17698 at location 4023c000 (read-only)
Frame at 12b000: shared by 15 process pages:
    Process 17636 at location 4024c000 (read only)
    Process 17667 at location 4024c000 (read-write)
```

If two `frame_record` give you different counts (e.g., because a process forks between two calls to `get_shared_frames()`), use the larger one. If any of them indicates the frame is in the page cache, print as such.

You may assume that no process will be given twice in the argument list. Frames should be listed in ascending order of frame address. For each frame, the processes should be listed in ascending PID. A process can have multiple mapping to the same frame, and in such case they should be listed in ascending order of process linear address. If a frame is used only by one process pages in the given processes, it should not be printed.

4. Hints

Once you modify the `export-obj` list in the kernel, you have to use `make dep` to update the kernel dependencies, and use `make bzImage` to rebuild the kernel. This will recompile the kernel completely. Depend on the configuration of your computer, it might take up to half an hour. Luckily, this only has to be done once. (If you use a shared computer, you might want to save the kernel header directory and the kernel image.)

Within the `mm` structure of a task, the `mmap` holds a list of virtual memory regions (of type `struct vm_area_struct`), linked with the `vm_next` pointer. For each of the VM entry, you should scan the page table entries to find the relevant frame table entries (`struct page`) if they exist. You will need to refer to the kernel header files to understand how these structures are organized (see the reference section at the end of the assignment sheet).

You may assume your system call will be used only in 80x86 computers, without PAE (Physical Address Extension). In other words, you may assume that PMD is always present.

To test whether your system call is interpreting the `count` correctly, you can write a simple program and compile it with the static C library. If you run the program, and your system call report

many pages of this program is shared, then probably you have interpreted the *count* wrongly.

To test the locking behaviour, it is usually useful to artificially lengthen the duration that the system call lasts. You can do so using *schedule_timeout()* as in the last assignment.

The output format of the user program is designed so that it readily reveals the amount of sharing while at the same time remains easy enough to implement. It is easiest if you know enough STL of C++, in which case a few maps and sets will solve all the problems (you simply need to map frame numbers to information about the frames, including a set of process information). On the other hand, if you use C, your basic recourse will be to keep a large array of frame information entries, each recording both the frame and page information. The *realloc()* library function can help you enlarging the array on demand. Finally, a single call to *qsort()* would put the elements in a convenient order for output.

You will need the bits contained in the header files `linux/mm.h` and `asm/pgtable.h` to know whether a page is read-only, read-write or copy-on-write. However, you cannot include these files from a user program. Thus you might want to copy some of the constants defined there to your program.

5. Handin

Send 4 files for this assignment:

- The kernel modification in `get_shared_frames.c` (to be added to the kernel).
- The system call implementation in `get_shared_frames_mod.c` (to be compiled as module).
- The user program `shared_frames.cc` or `shared_frames.c`.
- A file called `readme.txt` telling the names and IDs of the members in your group, and the version of the kernel that your programs are written against.

6. References

Tutorial 7 reading and solution. (Page table.) **Note:** In the Redhat kernel, the name of the function *pte_page()* is changed to *pte_offset_kernel()*. In a stock kernel, it is named *pte_offset()*, and the PMD function for *pmd_page_kernel()* is called *pmd_page()*.

Kernel header: `<linux/mm.h>` about **struct** *vm_area_struct* and **struct** *page* (together with the meanings of the flags and count, in comments).

Kernel header: `<linux/sched.h>` about **struct** *mm_struct*.

ULK: Chapter 7—The Memory Descriptor/Memory Region, page 198–206; or LDD: Chapter 13—Virtual Memory Areas, page 378–382. (Memory regions.)

ULK: Chapter 7—Copy On Write, page 225–227. (Frame counts.)

ULK: Chapter 6—Page Frame Management, page 158–160; LDD: Chapter 13—The Memory Map and struct page, page 373–375. (struct page.)

ULK: Chapter 11—Kernel Synchronization, page 305–310; or LDD: Chapter 3—A Brief Introduction to Race Conditions, page 76–78 (Kernel semaphores).

ULK: Chapter 11—Atomic operations, page 301–302; or LDD: Chapter 9—Atomic integer operations, page 285–286 (`atomic_t`).