Govind Balaji S
cs18btech11015
K Havya Sree
cs18btech11022

HWP Assignment 3

CS3523
June 21, 2020

Plagiarism statement

We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand our responsibility to report honour violations by other students if we become aware of it.

Names: Govind Balaji S, K Havya Sree
Date: June 21, 2020
Signatures: SGB, KHS

Problem

The goal of this assignment is to implement prefetch and demand paging on a device special file.

Data Structures

Since the file is a device special file, its contents are not in hard disk by default. Instead we allocate 1MB of kernel memory for each inode as its data, encapsulated along with its size in the following `struct`:

```
struct mykmod_dev_info {
        char *data;
        size_t size;
};
```

The above data structure is allocated when a device special file is opened for the first time, and set to be the inode's private data.

```
static struct mykmod_dev_info *devices[MYKMOD_MAX_DEVS];
```

A table of pointers to the device info data structures is maintained, as they need to be freed when the driver unloads.

For each VMA, the following data structure is allocated. It has a pointer to the respective device info data structure, and a counter that counts the number of page faults that occured in this VMA.

```
struct mykmod_vma_info {
        struct mykmod_dev_info *dev_info;
        unsigned long npagefaults;
};
```

The above per VMA data structure is stored in the respective VMA's private data.

File operations

When the kernel module is loaded, using `register_chrdev()`, we register our character device driver with our own `struct file_operations`. We implement `open()` with `mykmod_open()`, `release()` with `mykmod_close()`, and `mmap()` with `mykmod_mmap()`.

In `mykmod_open()`, we allocate the inode's private data with device info if the inode is opened for the first time. We set the filp's private data same as the inode's private data.

In `mykmod_close()`, we just log the arguments.

In `mykmod_mmap()`, we first check the offset and length if the VMA may cross the file bounds. In that case `-EINVAL` is returned. Otherwise, we initialize the vma info data structure for this vma, setting number of page faults to 0.
We set up the vma to use our own `struct vm_operations_struct` .

VM operations

We implement `open()` with `mykmod_vm_open()`, `close()` with `mykmod_vm_close()`, and `fault()` with `mykmod_vm_fault()`.

The `mykmod_vm_open()` justs log the parameters.

The `mykmod_vm_close()` frees the per VMA data structure we used for the VMA.

In `mykmod_vm_fault()`, we build the virtual physical mappings. With `virt_to_phys()`, we get the physical base address of the 1MB data in device info data structure. The faulted page's offset on the file is the sum of VMA's offset on the file and the faulted page's offset on the VMA. With `pfn_to_page()`, we set the `vmf->page` to the `struct page*` of the faulted page. Then we increment the number of page faults in the VMA info data structure .

Prefetching vs Demand Paging

Prefetching or demand paging is specified by the user program in its call to `mmap()`. The flag `MAP_POPULATE` implies prefetching, otherwise it is demand paging. Either way, the kernel handles it and the only difference is when it calls the `fault()` implemented by us. With demand paging,

the `mykmod_vm_fault()` is called whenever a page that is not in the memory already is accessed. With prefetching, the `mykmod_vm_fault()` is called by the kernel right after `mmap()` system call itself for all the pages.

Logging

Throughout the driver routines, we use `printk()` to log the parameters, function status, error messages etc. These can be later checked with `# dmesg`.

memutil.cpp

We parse the command line args into `MAP_READ` for reading and `MAP_WRITE` for writing.

In `MAP_READ`, the device memory is compared with the message repeatedly until the end of the file. But if the length of the message is 0, the device memory will be accessed but not compared.

In `MAP_WRITE`, the message is written into the device memory repeatedly until the end of the file.