OSPJ3_Team4_Report

Chun-Ju Wu B03202040, Meng-Jin Lin B04901009

June 29, 2018

1 Code Reading

1.1 How readahead is called when page faults occur ?

The whole procedure starts from the system call "mmap()".

mmap(): mmap() is a system call in linux that map files to memory. The system call can be found in "arch/x86/kernel/sys_x86_64.c". It performs error checking first, then call "sys_mmap_pgoff()" to do the following jobs.

sys_mmap_pgoff(): This function can be found in "mm/mmap.c". It gets the parameters that are needed to be input to the next function "do_mmap_pgoff()", and do error checking.

do_mmap_pgoff(): This function can be found in "mm/mmap.c". The process that are calling mmap() is the task_struct* current (task_struct is defined in include/linux/sched.h). What we want to do is to modify the mm_struct* mm (mm_struct is defined in include/linux/mm_types.h) pointed by current. In mm_struct, there contains doubly linked list vm_area_struct* which is a list of VMAs (virtual memory area) and pgd (page global directory). In this function, argument addr is first assigned to be round_hint_to_min(addr), and len is set to be PAGE_ALIGN(len), and checked if overflow happens. If no error, then addr is then set by get_unmapped_area(). Then the function perform error checking, and call mmap_region().

mmap_region(): This function is also defined in "mm/mmap.c". In the function, argument vma is first set to be find_vma_prepare(). If the new VMA overlap with the old mapping, do_munmap() is performed to clean the old one. Other functions such as vma_merge() are also performed. After those procedures, the arguments in vma (e.g. vm_mm, vm_start, ...) are set. Then the file are map to the vma by file->f_op->mmap(file, vma) function. With some error checking, the addr is then return.

file->f_op->mmap(file, vma): The struct file is defined in "include/linux/fs.h". The operations of the files are defined by the struct file_operations in the struct file which is called f_op. The mmap(file,vma) is a function in struct file_operations. Different file system have different mmap function. For a general mmap of a disk file, the function is set to be generic_file_mmap(). This function is defined in "mm/filemap.c". In this function, vma->vm_ops is set to be generic_file_vm_ops, where its fault operation is set to be filemap_fault().

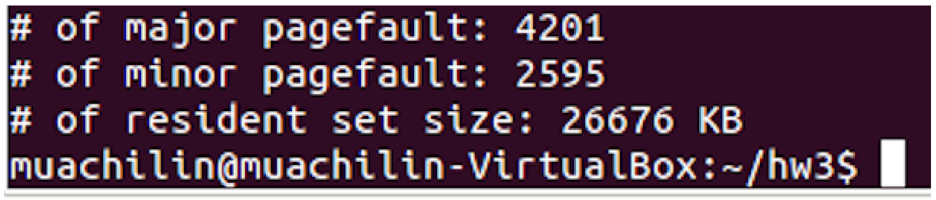
When the page fault happen, filemap_fault() will be called.

filemap_fault():
When it is need to read in file data for page fault handling, the function "filemap_fault()" in the filemap.c will be called. After filemap_fault() is called, this function will check if the page cache is empty or not. If a page is found, the function then calls "do_async_mmap_readahead()". Otherwise, it calls "do_sync_mmap_readahead()" instead. In the function do_async_mmap_readahead(), if PageReadahead(page) is true (means that we want to extend the readahead), it will call "page_cache_async_readahead()" in readahead.c for file readahead for marked pages. In page_cache_async_readahead(), it will defer asynchronous readahead if there is I/O congestion by calling "bdi_read_congested()" and return. It will call ondemand_readahead() to do

readahead.
In the function ondemand_readahead(), there are many steps before calling "__do_page_cache_readahead()". First of all, "max_sane_readahead()" determines a sensible upper limit according to a desired number of PAGE_CACHE_SIZE. Secondly, check the offset and push forwardthe readahead window. Interleaved reads are also be tackled with by the variable "hit_readahead_marker". After other detections, the function "__do_page_cache_readahead()" will finally be called. It will preallocate as many pages as need.

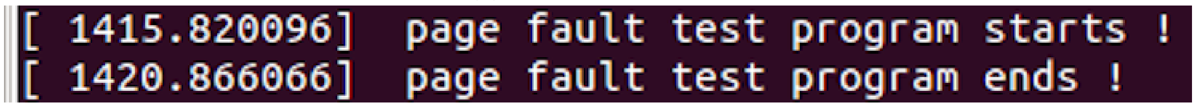
2 Revise the Readahead Algorithm for Smaller Response Time

The default output of testing program:



```
# of major pagefault: 4201
# of minor pagefault: 2595
# of resident set size: 26676 KB
muachilin@muachilin-VirtualBox:~/hw3$
```

(a) Default Output



```
[ 1415.820096]  page fault test program starts !
[ 1420.866066]  page fault test program ends !
```

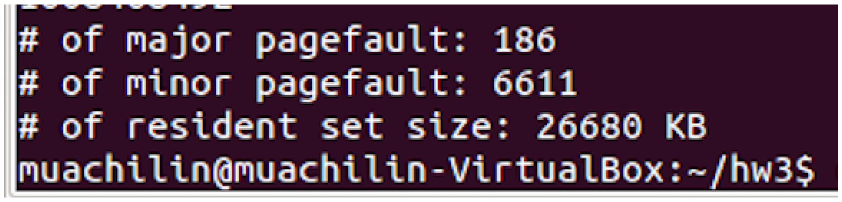
(b) Default dmesg

Figure 1: The output of default VM_MAX_READAHEAD.

We increase VM_MAX_READAHEAD defined in /include/linux/mm.h to make response time smaller. The following figures are the results of modifying VM_MAX_READAHEAD to 1024. We can see that the number of major pagefault decreases from 4201 to 186 and the time cost is reduced from 5.04597 to 0.791485.

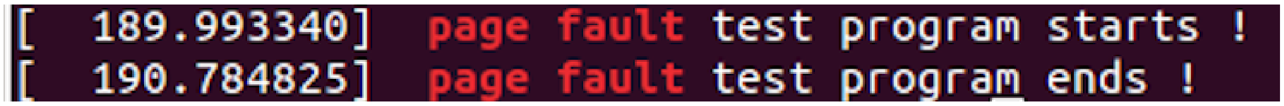
3 Bonus: Reduce Latency or Increase Throughput in Disk I/O

To reduce the latency or increase the throughput of the disk I/O, there are several kinds of methods, which can be referenced from the website "https://cromwell-intl.com/open-source/performance-tuning/disks.html". Here we present the method of changing the disk scheduler. In our system, there are four kinds of disk schedulers, which are "deadline", "cfq", "anticipatory", and "nood". The "deadline" scheduler is the default scheduler, which could be verified using the command "grep . /sys/block/sda/queue/scheduler". To change the



```
# of major pagefault: 186
# of minor pagefault: 6611
# of resident set size: 26680 KB
muachilin@muachilin-VirtualBox:~/hw3$
```

(a) VM_MAX_READAHEAD = 1024 Output



```
[  189.993340]  page fault test program starts !
[  190.784825]  page fault test program ends !
```

(b) VM_MAX_READAHEAD = 1024 dmesg

Figure 2: The output of VM_MAX_READAHEAD = 1024.

scheduler, we can simply type the command inside the terminal "echo [scheduler name] | sudo tee /sys/block/sda/queue/scheduler". The four kinds of schedulers are described below:

1. deadline: The deadline scheduler limit the maximum latency, and let the I/O request finish before its own deadline. There are two queues maintained per device. One sorted by sector and the other by deadline. If no deadline expires, the I/O requests are done in sector order

2. cfg: The Completely Fair Queuing scheduler put synchronous requests into per-process queues. Then time slices are allocated for each queues to access the disk. The time slices could depend on the I/O priority of the process. The asynchronous requests are batched into fewer queues.

3. anticipatory: The anticipatory scheduler is a scheduler that anticipates subsequent block requests and caches them for use[1].

4. nood: This is one of the simplest scheduler. It inserts all requests into FIFO queue, and perform request merging.

The results of changing the scheduler are listed in Table 1.

In the table, we can see that the default scheduler have the worst performance, and the anticipatory scheduler give the best performance. The performance is about 3% better than the default one. Thus changing the scheduler could be a possible way to give a better performance for the disk I/O.

[1]http://www.linux-mag.com/id/7724/

Scheduler	cost time	average cost time
Deadline	1.7042585	
	1.733984	
	1.684106	
	1.68391	1.701564625
cfq	1.674306	
	1.649779	
	1.696204	
	1.714974	1.68381575
anticipatory	1.651434	
	1.651394	
	1.652563	
	1.63203	1.64685525
nood	1.622905	
	1.646105	
	1.646105	
	1.737357	1.6585295

Table 1: The performance of different scheduler.