Operating Systems Project3

team28 B04902077江緯璿 B04902079甯芝蓶

Part1. Code reading

mmap

這是最頂端的 `mmap` 函式，他會往下call kernal中的 `mmap_pgof()` 來實現

mmap_pgoff

```
SYSCALL_DEFINE6(mmap_pgoff,
    unsigned long, addr, unsigned long, len, unsigned long, prot,
    unsigned long, flags, unsigned long, fd, unsigned long, pgoff
)
```

從這個函式開始進到kernel裡面。它會將file descriptor轉換成 `struct file` 型態的指標，並將此指標連同其他參數往下餵給 `do_mmap_pgoff()`

do_mmap_pgoff

```
unsigned long do_mmap_pgoff(
    struct file *file, unsigned long addr, unsigned long len,
    unsigned long prot, unsigned long flags, unsigned long pgoff
)
```

這個函數主要有幾個任務：

1. 根據file pointer、offset及其他資訊計算記憶體真正的位置
2. 將原本的flags轉換為VM版本的flag
3. 檢查權限是否合法以及offset、len是否有overflow的狀況。若不合法或是有錯誤，直接返回不再繼續往下。

接著，它用 `get_unmapped_area()` 尋找(或建立)一個可用的virtual memory space(即vma)，將此vma的位址連同其他參數一起交給下層的`mmap_region()

mmap_region

```
unsigned long mmap_region(
    struct file *file, unsigned long addr, unsigned long len,
    vm_flags_t vm_flags, unsigned long pgoff, struct list_head *uf
)
```

這個函式最主要的目的是對在上一層拿到的那塊vma進行設置：若該area原本有舊的mapping，就先把它給清掉。然後再把新的mapping設置上去，並創建一個 `struct vm_area_struct` 裝它。過程中同時檢查是否有against space limit等等錯誤。最後，把這個 `struct vm_area_struct` 的結構傳給該檔案的 `->mmap()` 函式操作。在ext4的file system下，這個函式就是 `ext4_file_mmap`

ext4_file_mmap

```
static int ext4_file_mmap(struct file *file, struct vm_area_struct *vma)
```

在這個函式中，直接把 `vma->ops` 對應到 `&ext4_file_vm_ops` 。而 `filemap_fault` 其實就是在對vma做operation時，發生page fault的對應handeler (終於進到page fault的階段了！)

filemap_fault

```
int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
```

發生page fault時，會先找找這個page是否已經存在在cache中。有的話發送一組非同步read_ahead請求，由 `do_async_mmap_readahead()` 實現；沒有的話則發送同步請求，由 `do_sync_mmap_readahead()` 實現。

do_async_mmap_readahead

```
static void do_async_mmap_readahead(
    struct vm_area_struct *vma, struct file_ra_state *ra,
    struct file *file, struct page *page, pgoff_t offset
)
```

這個函式會透過（`VM_RandomReadHint(vma)`）檢查目前的狀況是否有進行read_ahead的必要，無則返回，有則繼續往下。

page_cache_async_readahead

```
page_cache_async_readahead(
    struct address_space *mapping, struct file_ra_state *ra,
    struct file *filp, struct page *page, pgoff_t offset, unsigned long req_size
)
```

在這個函式中，若檢查到以下三種狀況則直接return不繼續往下：

1. no read-ahead
2. 想要read-ahead的東西已經在write back了
3. I/O congestion狀況不允許

ondemand_readahead

```
static unsigned long ondemand_readahead(
    struct address_space *mapping, struct file_ra_state *ra,
    struct file *filp, bool hit_readahead_marker,
    pgoff_t offset, unsigned long req_size
)
```

這個函式的最主要目的是『依照現有訪問規律去預測接下來的demand狀況，來決定是否要進行read-ahead以及read-ahead的量』。簡單來說，如果offset在文件的一開頭，或著目前的訪問已是按照順序的，便認定接下來應該會繼續按照順序訪問，則呼叫 `__do_page_cache_readahead()` 實作，否則就暫時不進行預讀。

__do_page_cache_readahead

```
static int __do_page_cache_readahead(
    struct address_space *mapping, struct file *filp, pgoff_t offset,
    unsigned long nr_to_read, unsigned long lookahead_size
)
```

到了這個函式才真得把東西讀進來！只要在合法範圍內(即不超過文件最後面)，就依照 `nr_to_read` 參數的指示把盡可能多的page都讀進來(已經在cache的就從cache抓，無則用 `page_cache_alloc_cold()` 清一個空間，然後把它放進來)
最後，再用 `read_pages()` 開始進行I/O。

Part2. Revise the readahead algorithm

> Discussed with 陳宥嘉、楊舒瑄、陳佳佑、莊翔旭
> Testing Environment: Ubuntu 12.04.5 LTS with Linux 2.6.32.60 kernel on VirtualBox with 1GB RAM on Macbook Pro

Revised Code & Rebuild Kernel

```
include/linux/mm.h: VM_MAX_READAHEAD 128 -> 2048
```

Another Testing Method without rebuilding kernel:

```
sudo /sbin/blockdev --setra 2048 /dev/sda
```

Experiment process

Case 1: Original Settings with VM_MAX_READAHEAD = 128

Avg. Running Time (10 times): 1.5654465 sec
# of Major Page Faults: 4201

```
w4a2y4@w4a2y4-VirtualBox: ~/hw3
-1041259189
517861973
573805618
1441864162
831873783
1954997726
-802210018
-571331550
687073953
-1702804749
-2118621831
871876224
-186862404
-1924523629
1808037273
1187597919
361373333
102010677
671903510
1608468492
# of major pagefault: 4201
# of minor pagefault: 2595
# of resident set size: 26680 KB
w4a2y4@w4a2y4-VirtualBox:~/hw3$
```

```
[ 1214.550794]  page fault test program starts !
[ 1216.082419]  page fault test program ends !
[ 1245.996495]  page fault test program starts !
[ 1247.484299]  page fault test program ends !
[ 1251.768179]  page fault test program starts !
[ 1253.229478]  page fault test program ends !
[ 1258.358737]  page fault test program starts !
[ 1259.895927]  page fault test program ends !
[ 1263.012348]  page fault test program starts !
[ 1264.556845]  page fault test program ends !
[ 1268.420336]  page fault test program starts !
[ 1269.882997]  page fault test program ends !
[ 1278.717076]  page fault test program starts !
[ 1280.201140]  page fault test program ends !
[ 1285.903382]  page fault test program starts !
[ 1287.460786]  page fault test program ends !
[ 1290.198583]  page fault test program starts !
[ 1292.284711]  page fault test program ends !
[ 1320.389263]  page fault test program starts !
[ 1321.891056]  page fault test program ends !
w4a2y4@w4a2y4-VirtualBox:~/hw3$
```

Case 2: VM_MAX_READAHEAD = 2048:

Avg. Running Time (10 times): 0.4850969 sec
# of Major Running Time: 186

```
w4a2y4@w4a2y4-VirtualBox: ~/hw3
-1041259189
517861973
573805618
1441864162
831873783
1954997726
-802210018
-571331550
687073953
-1702804749
-2118621831
871876224
-186862404
-1924523629
1808037273
1187597919
361373333
102010677
671903510
1608468492
# of major pagefault: 186
# of minor pagefault: 6609
# of resident set size: 26676 KB
w4a2y4@w4a2y4-VirtualBox:~/hw3$
```

```
w4a2y4@w4a2y4-VirtualBox: ~/hw3
clear_cache.sh   pro.py      syslog.sh   test.h     time_interval
input.log        random.sh   temp        test_new   time_interval_ans
w4a2y4@w4a2y4-VirtualBox:~/hw3$ cat test_new/dmesg2 | grep 'page fault'
[  359.712459]  page fault test program starts !
[  360.206737]  page fault test program ends !
[  367.080794]  page fault test program starts !
[  367.515776]  page fault test program ends !
[  370.338671]  page fault test program starts !
[  370.736982]  page fault test program ends !
[  373.374054]  page fault test program starts !
[  373.969958]  page fault test program ends !
[  376.290552]  page fault test program starts !
[  376.817969]  page fault test program ends !
[  379.322853]  page fault test program starts !
[  379.774616]  page fault test program ends !
[  382.427904]  page fault test program starts !
[  383.048677]  page fault test program ends !
[  387.074381]  page fault test program starts !
[  387.511493]  page fault test program ends !
[  389.771366]  page fault test program starts !
[  390.217475]  page fault test program ends !
[  392.473584]  page fault test program starts !
[  392.917904]  page fault test program ends !
w4a2y4@w4a2y4-VirtualBox:~/hw3$
```

Explanations and Discussions :

這次測試對於原先版本與改過的版本各測試10次，並將dmesg中的時間取平均以進行比較。實驗結果發現，將 `VM_MAX_READAHEAD` 調大後，對於效能有極大的成長，耗費時間減少了69.01%。原因在於，將該參數調大後會直接影響到 `mm/backing-dev.c` 內的 `default_backing_dev_info -> ra_pages` ，將可使得單一操作能預讀的最大page數增加，進而使Major Page Fault次數減少而讓整體的運作效率

提升。而除了修改mm.h外，也可藉由 `/sbin/blockdev --setra` 進行暫時性的測試以節省一直重新編譯kernel的麻煩。運用此方式，我們也同時測得當將參數設為8192時，將有近85%的效率提升。由此即可驗證調整預讀的page量可影響運行的效率。

Experiment records of outputs and dmesg are attached in the zip folder.