OS Team 14 Report

Part 1: Code Reading

`mmap()`

In `.../arch/x86/kernel/sys_x86_64.c`:

This function sets the return value to `-EINVAL` and immediately returns that value if an error occurs. Otherwise, it will call `sys_mmap_pgoff()`, which is a system call defined in `/arch/x86/kernel/syscall_table_32.S`, i.e. the same place introduced in Project 1. Note that we can simply call the function without the leading `sys_`, as with other system calls.

`mmap_pgoff()`

Again, the return value is first set ahead of error checking. If no errors occur, it turns off flags, then calls a semaphore to lock the critical section. Then, `do_mmap_pgoff` is called, setting the `retval` to its return value instead.

`do_mmap_pgoff()`

`PAGE_ALIGHT` first rounds up the size of the would-be area to exactly fit a page size, to reduce fragmentation. If there is no more memory, `-ENOMEM`, which signifies this, is thrown. Then, `get_unmapped_area()` searches for a valid place to map the memory. This address is then passed to `mmap_region()`.

`mmap_region()`

This function first clears out old maps, then attempts to reduce overhead by simply merging with existing maps. Upon failing that, it calls the mapper, initializes `vma`, and if a file (to be mapped) was specified (not `NULL`), it finally calls the file's `file_operations` struct:

`struct file`

In `.../include/linux/fs.h`, with everything irrelevant omitted:

```
struct file {
//everything else omitted
 const struct file_operations *f_op;
};
```

In which case, `f_op` holds the following:

`struct file_operations`

Again, in `.../include/linux/fs.h`, with everything irrelevant omitted:

```
struct file_operations {
//everything else omitted
 int (*mmap) (struct file *, struct vm_area_struct *);
}
```

This `mmap()` function is the generic `mmap` function `generic_file_mmap()`.

`generic_file_mmap()`

In `.../mm/filemap.c`, the function sets the `vma` 's operations to `generic_file_vm_ops`:

```
int generic_file_mmap(struct file * file, struct vm_area_struct * vma)
{
//everything else omitted
 vma->vm_ops = &generic_file_vm_ops;
}
```

We can see that `vm_ops` (of type `vm_operations_struct`), which is also declared in the same file,

```
const struct vm_operations_struct generic_file_vm_ops = {
 .fault   = filemap_fault,
};
```

Which means that, under this system, the handler for page faults will be `filemap_fault()`.

`filemap_fault()` and the readahead algorithm

When this function is invoked, it first checks if the page is already in the cache. If so, it will call `do_async_mmap_readahead()`:

`do_async_mmap_readahead()`

Which calls `page_cache_async_readahead()`:

`page_cache_async_readahead()`

This function, which is declared in `.../mm/readahead.c`, then calls `ondemand_readahead()`:

`ondemand_readahead()`

This function, which is in the same file, then calls `__do_page_cache_readahead()`, which actually does the readahead.

`__do_page_cache_readahead()`

What this function does is first preallocate pages that are requested (specifically, `nr_to_read` pages), then submit these for I/O, ending the readahead process.

`page_cache_async_readahead()` and `ra_submit()`

However, if the page cache was initially empty, the `filemap_fault()` will instead call `page_cache_sync_readahead()`, which will then call `ra_submit()`, which then calls `__do_page_cache_readahead()` anyway.

Part 2

The result is gathered via the following method:

1. Compile `test.c`.
2. Run `sudo ./clear_cache.sh`.
3. Run `sudo ./a.out`.
4. Run `dmesg > o.txt` to redirect contents to a file.
5. Observe the start and ending time of the algorithm (from the first column of `o.txt`). The total time is simply the difference of these two times.
6. For the non-default method, the improvement is the percentage change from the default time to the new observed time.

First we show the default running time and statistics.

- Results:

```
# of major pagefault: 4200
# of minor pagefault: 2590
# of resident set size: 26660 KB
[  179.360500]  page fault test program starts !
[  179.361076] a.out, B76C5000
[  179.361090] a.out, B76C4000
[  179.361094] a.out, B76CC000
[  203.421577]  page fault test program ends !
[  203.422099] a.out, B76CE000
[  203.422109] a.out, B762C000
```

Default: total time = 24.061077 seconds

Implementation (Method 1)

Increase `VM_MAX_READAHEAD` from 128 to 2048. This will allow more to be read each time.

In `mm.h`:

```c
/* readahead.c */
#define VM_MAX_READAHEAD    2048    /* kbytes */
#define VM_MIN_READAHEAD    16  /* kbytes (includes current page) */
```

- Results:

```
# of major pagefault: 95
# of minor pagefault: 6695
# of resident set size: 26660 KB
[  119.236476]  page fault test program starts !
[  119.237123] a.out, B76A6000
[  119.237138] a.out, B76A5000
[  119.237142] a.out, B76AD000
[  121.668748]  page fault test program ends !
[  121.669538] a.out, B76AF000
[  121.669546] a.out, B760D000
```

Modified: total time = 2.432723 seconds

Change from default: improvement = 89.889384419%

Implmentation (Method 2)

Increase the readahead buffer size nearly 32768 times. This is also to improve the amount read each time readahead is invoked.
Note that this method does not required touching the kernel code at all, just root access.

- Default read ahead buffer size: 256

```
ubby@ubby-VirtualBox:~/Desktop/OS3/hw3$ sudo blockdev --getra /dev/sda
[sudo] password for ubby:
256
```

- Modified buffer size: 8388600

```
ubby@ubby-VirtualBox:~/Desktop/OS3/hw3$ sudo blockdev --setra 8388600 /dev/sda
ubby@ubby-VirtualBox:~/Desktop/OS3/hw3$ sudo blockdev --getra /dev/sda
8388600
```

- Results:

```
# of major pagefault: 2
# of minor pagefault: 6791
# of resident set size: 26664 KB
```

```
[  621.621174]  page fault test program starts !
[  621.621820] a.out, B76FF000
[  621.621837] a.out, B76FE000
[  621.621841] a.out, B7706000
[  622.890143]  page fault test program ends !
[  622.890760] a.out, B7708000
[  622.890769] a.out, B7666000
```

Modified: total time = 1.265545 seconds
Change from default: improvement = 94.7402812 %

Notes and observations

The idea with the two methods above is that, in exchange for higher memory usage, a larger amount of pages can be read into memory, improving response time for many sequential reads.

That being said, for some reason, increasing the number of pages to read (modifying `ra->ra_pages`) and the maximum limit (modifying `ra->async_size`) by 100 barely helped at all.

Modified: total time = 23.880848 seconds
Change from default: improvement = 0.749047933 %

- Results:

```
# of major pagefault: 4154
# of minor pagefault: 2635
# of resident set size: 26660 KB
```

```
[   69.482460]  page fault test program starts !
[   69.482972] a.out, B7659000
[   69.482986] a.out, B7658000
```

```
[   93.363308]  page fault test program ends !
[   93.363847] a.out, B7662000
[   93.363856] a.out, B75C0000
```

We are not quite sure why this is the case, as we also increased the limit (via `max_sane_readahead`). That being said, the amount of major pagefaults did change, albiet insignificantly.

Sources

- Most of the code reading from part 1 relied on this website: https://elixir.bootlin.com/linux/v2.6.32.60/ident/__do_page_cache_readahead
- Method 1: the hint was very useful.
- Method 2: http://fibrevillage.com/storage/291-blockdev-command-examples
- Method 2: https://unix.stackexchange.com/questions/30286/can-i-configure-my-linux-system-for-more-aggressive-file-system-caching