Lab 7 Report

Ringo Groenewegen and Federico Casenove

We have written most of the code required for the assignment, but there were too many bugs from the previous lab to properly test it.

FIFO LRU with CLOCK

```
FILES: swap.c, swap.h, swap_util.c, swap_util.h
```

In our implementation of FIFO LRU we decided to keep it simple. The clock algorithm is implemented in the `get_page()` function which returns the LRU page that can be swapped to disk.

We use a single list to store pages that are accessed inside the page fault handler. If a page is not yet in the swap list, we add it using `add_swap_page()`. If the page is already in the list, we move it to the front of the queue to make it the most-recently used page using `mru_swap_page()`. But before handing the page to the `swap_out()` function each page is checked by the `check_clock()` function. This function checks the `PAGE_ACCESSED` bit of the pages in all address spaces that map to this page by using the rmap. If any of the address spaces have accessed the page we move the page to the head of the swap list and disable the `PAGE_ACCESSED` bit. If the page has not been accessed by any task then the page is passed to the `swap_out()` function. This makes the algorithm LRU instead of just FIFO.

OOM

```
FILES: oom.c, oom.h
```

The function `oom_kill()` will go through all running tasks and determine their OOM score. This score is the number of pages that are mapped which indicates how much memory is used by the task. The task with the largest memory footprint will be deleted and its memory will be freed.

Ideally `oom_kill()` should only be called as a last resort - we don't want to kill a task unless there is no alternative. Therefore it should only be triggered if the swap disk is already full so we also can't swap memory. In that case we should kill the task with the largest OOM score.

Reverse Mapping

```
FILES: rmap.c, rmap.h
```

In the lectures a distinction is made between file-backed and anonymous pages, and that the method of reverse mapping are different for each. In our implementation we just treat everything as anonymous to keep

it simple.

The purpose of a reverse mapping is to know which virtual address spaces map to a particular physical address. This is used when swapping memory out as well as in. When swapping out a page frame, we need to know all address spaces that may be mapped to this page frame, so that when we swap the page frame out we have to set the PTE entry as “swapped” instead of “present” to indicate that it is on disk. We then write the disk address inside the PTE instead of the page address. When we swap memory back in, we allocate a new physical page in memory and we want to write this address back in the PTE of all address spaces that initially mapped to this entry, and replace that by the page address.

To implement the reverse mapping we add all VMA’s that map to the same memory regions to a linked list. This happens during cloning of a task, so when we create a VMA we create a head node for the linked list, and then during each fork we will add the cloned VMA to the linked list as well. In addition to this, we add a pointer to the head of the linked list to each VMA. This head is called `rmap`. The rmap also contains a lock to allow for concurrent access.

We only swap user pages, kernel pages will not get swapped. Therefore we add all pages to our swap list inside the page fault handler, because that function will get triggered for every page that is accessed, and therefore mapped.

Swapping

```
FILES: swap.c, swap.h
```

Swap out to disk

- `swap_out()` is called inside `swap_thread()` in a loop when the memory threshold has been reached. We want to start swapping out memory once we start running out of free pages, but before they have all run out because that would cause the task to block. Once the memory threshold is reached, we free `SWAP_BLOCK` pages to disk by calling `swap_out()` in a loop, giving us new free memory.
- Inside `swap_out()` we first use `get_page()` as described above (CLOCK algorithm) to get a physical page to swap out.
- Use `get_free_disk_addr()` to find a free address on the disk where we can write to.
- Use the `rmap` pointer to find the reverse map linked list. This list contains all VMAs that may be affected when we swap the physical page out to disk.
- Disable `PAGE_PRESENT` and set the address of the PTE for all VMAs that map to this physical address to the disk address where we write.
- Free the page that we just swapped out.
- We poll the disk to check its status. If the disk is busy, we stop and switch to another task to avoid blocking.

Swap in from disk

- `swap_in()` is called when there is a page fault on a user space address for which the `PAGE_PRESENT` bit is disabled but `PAGE_ADDR(entry)` is not empty. That implies that the page is not mapped but it is on disk, and the value of `PAGE_ADDR(entry)` is the address of the page on disk.
- When we want to swap a page in, we first have to check the page cache because it may be that the kernel thread is currently swapping that page out to disk. To help with that, the kernel thread will put the page that it is swapping out in the cache temporarily.
- If the page is not in the page cache, we use `page_alloc()` to create a new physical page and we read the value from disk into this new page.
- We must first determine the VMA in which the page faulted address lies. Using this VMA we can access the `rmap` linked list, which contains all VMAs that need to map to this new page. We then update the PTEs to map to this new page to which the data has been written from disk.

Limitations and problems

- There are still bugs in the multi-core implementation which regularly causes issues when running. One example is that we need to enable our debugging prints for the program to run, otherwise it crashes very quickly. We don't know why this is the case.
- There is a bug in the memory layout so we have a problem accessing higher addresses. We are therefore unable to allocate all memory: this seems to be a problem with `page_init_ext()`. Therefore we have set our `MEMORY_THRESHOLD` as found in `oom.h` to be below the point where we start allocating pages with large addresses to prevent it from crashing.
- The page cache is missing. Its functionality is described above in the "Swap in from disk" section.
- Our `get_free_disk_addr()` function currently just increases the disk address linearly. There is no option to free data on the disk, so every time we want to write to disk we increase the value by `PAGE_SIZE`.
- `oom_kill()` should be called when the swap disk is full, but because we have used a linear allocation without freeing memory, we are not calling the function when the swap disk has also run out of storage.
- The kernel is non-blocking for swap out but it is not for swap in.
- In `main.c` a line can be uncommented to enable the OOM thread (the swap thread should then be commented out). This can be tested with our test `oomtest.c` which allocates a lot of memory, and it will kill the task that is using the most memory. As stated before this OOM thread is just for testing, the `oom_kill()` should be handled by the swap as a last chance.