INF-2201: OP fundamentals
Project 5 Virtual Memory
Lise Ekhorn Johansen
UiT id: ljo177@uit.no
Github username: Lise-johansen
GitHub classroom: project-5-Lise-johansen
02.05.2023

1. Introduction
This report describes the implementation and design of a simple on-demand-paged virtual memory system using a USB stick as the swapping area. Each process has its own separate virtual address space, with proper protection ensuring isolation and protection of the kernel.

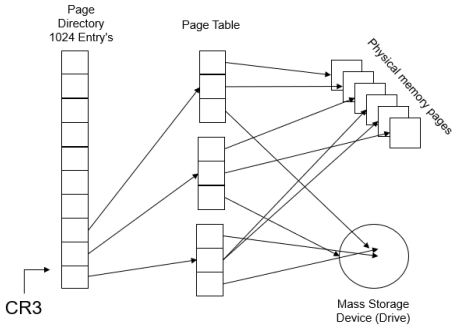
2. Technical background
Virtual memory provides protection and management by allowing the PC to use the disk as an extension of the RAM. And protection comes in the form of memory isolation. **Paging** is a virtual memory technique, that divides memory into pages, enabling easy switching between the RAM and drive. [1] **The virtual memory address** is the memory address that programs use to access memory, while the **physical memory address** is the actual location in RAM/drive where the data is stored. [2] Picture 1 shows how a 32-bit virtual address is split into different parts. **Two-level page** tables are a technique used to manage large amounts of virtual memory. The **page directory** contains the 32-bit memory address (combined with the virtual memory) to the **page table**, which in turn conation the mapping (32-bit memory address + virtual memory address) to memory in RAM. **Bitmasks** are a way to store data compactly and efficiently using boolean logic. A mask defines what bit to keep or clear. And masking bits in a value are accomplished by doing a bit operation. [3][4]

```
1101001011 1001100100 101000110111
Directory index Table index Entry information
```

Picture 1: In a 32-bit address the first 10 bites is the page directory index, the next ten is the page table index, and the least significant are the page directory entry. This entry holds all needed information about an address.

3. Design
3.1 Memory
Using a struct to keep track of memory, the struct is represented as a fixed-size array, that is initialized with appropriate values. One linked list represents and keeps track of all free pages and another for all used pages, both list has a next pointer stored as a header in the struct.

3.2 Two-level paging table
The page directory and page table are represented as a page. Each directory or table represents one page. An entry in the page directory points to an entry in the page table, and the entry in the page table points to a place in memory. A thread shears its directory with the kernel, and a process has a one-page directory, one-page table, one stack table, and a page. Picture 2 shows the design of the painting table.



Picture 2: Each page directory contains information about 1023 page tables. One page directory entry are 4 bytes, and maps 4GB of virtual memory address space. Each page table conation information about 1024 4KB pages. A page table is represented in a single, physical page. One page table entry of 4 bytes.

3.3 Mapping
Have to set that the virtual address is the same as the physical address, for the kernel/thread memory and video buffer. If it's a kernel/thread it maps the rest of physical memory, e.g. identity mapp rest of memory. If it is a process the memory is mapped abstractly, e.g. the virtual memory is not the same as the physical memory address.

3.4 Page fault handler
The thought is when a process is marked as not present in memory/table, it will trigger a read from the disk. The process is read from the disk and gets allocated a page. Then the page to the faulting address page table is mapped to memory.

3.5 Evict algorithm
When the free pages list is full, an unpinned page is taken, a write from disk is triggered, and a now usable page is returned.

3.6 Bit masking
Set the correct bit value for every process using masks and bit shifting and save it in the correct page table.

4. Implementation
4.1 Memory
To divide up the memory into pages, a for loop iterate through all pageable pages, calculate the correct start address, initialize all vaults, and links the pages together in a null-terminated linked list, where the last page points to NULL. The last page is manually set. Instead of using an iterator plus 1 as an index, the index is set to the last page minus one. Lastly, a pointer is set to the address of the first array element.

To allocate a page from the free page list, first, check if there are any pages available. If available, the pointer to the page is set to the next page, and the page is marked as not pined. If there are no available pages, the replacement algorithm picks the first unpinned used page. More on this in 4.2. Lastly, for both cases, the pointer to the memory address of the page is returned. Remember to manage the used list before returning.

Supervising the used list includes inserting a page into the lined list of used pages. If the used list is empty then the page next points to NULL. Else in a while loop iterate to the last page, and sets its pointer to point at the new last page. The new last page next pointer points to NULL.

4.2 Evict algorithm: Extra created
Find an unpinned page from the used list, then extract the base address bit of the sonn-evicted pages page table using the page directory index and a constant base address mask. This address is cast to a 32-bit unsigned integer and saved in a variable. Unmappes the page by marking it as not present in the page table. Write its data from RAM to disk, flushes the cache, zeroes out the page, and finally puts the evicted page at the end of the used list.

4.3 Mapping
A page directory and page table are made. The first 640KB is iterated through and identity-mapped as present in the page table. The video buffer is also identity-mapped present in memory by using its memory address. Then the kernel set the rest of the memory as present in its table.

4.4 Two-level paging table
A kernel page directory is made as a static unsigned int. A page is allocated for the directory, a page table is made, mapped, and inserted inside the kernel directory. A threads directory

points to this directory. For a process, a page directory is allocated and mapped. The page table is allocated and inserted inside the page directory e.g. the directory points to the start of the page table. Processes are marked as not present (more in 4.5). A stack table is made in inserted in a directory, and the allocated page is mapped as present in the stack table. All this happens as an atomic operation.

4.4 Page fault handler

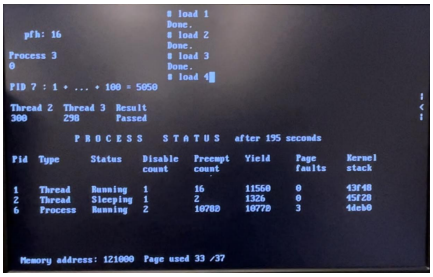
Acquires a lock and checks the error code of the currently running process. In a switch case if the error code is ether 0b000, 0b100, 0b010, 0b110, e.g. a read or write operation increments the fault count. Calculate the index of the page directory table and page table using the fault address (also done in 4.2). Allocate and map a page to the faulting page, pins the keyboard, and read the page from the disk. Then the page is market present in memory and mapped to the page table to the faulting address. Finally, the lock is released. The default case in the switch case is to do nothing.

4.5 Bitmaskin: mark not present

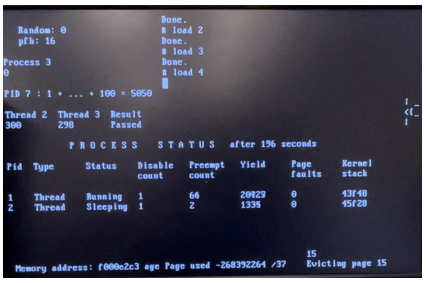
Iterate over a range of virtual addresses corresponding to the whole process memory. The end of the process is a calculator by multiplying the process size by the size of a disk section. For each address in this range, the table index is extracted. And a bitwise OR operation between the virtual address and a mask value that is then NOT-ed by the integer value 6 or the bit value 110. The resulting value is then stored in the page table as an entry.

4. Experiments and Result

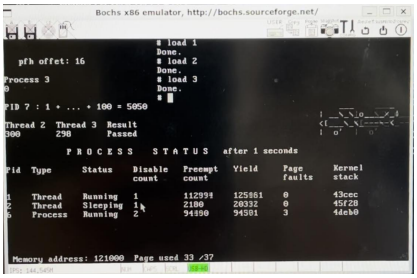
Correctness is evaluated by running the OS image on a PC without an existing OS. If the OS runs the implementation is deemed correct. To see if the eviction algorithm works a simpler test is conducted.
The first set of pictures is the result when running the image on a PC without an existing OS. The second set is when the image is run in an emulator, it is this set of pictures referenced when discussing the result.



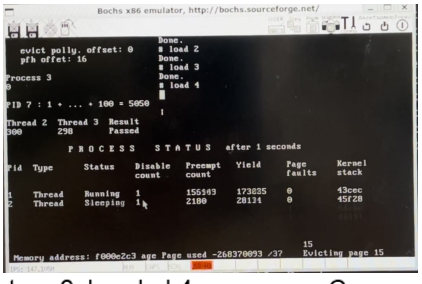
Picture 3: Loaded 3/4 processes. Can see this on the print statement (left coroner), that 33/37 page are used..



Picture 4: After the last process is loaded, the print statement (left coroner) shows a trash value indicating that it failed to write correctly from disk.



Picture 5: Loaded 3/4 processes. Can see a green light indicating that the program is reading from disk.



Picture 6: Loaded 4 processes. Can see a read light indicating that the program is writing from disk.

5. Discussion

5.1 Implementation

The page number e.g. number of pages in memory is still small enough so that an eviction still happens, indicated by the red light in picture 6.

A non-needed if-else statement is in the mapping function for the readability of another function. The choice was made to have a part of the setup for the process page table and page directory be in the mapping function (in the else statement). Did this so the setup table function does not contain so much code since the mapping function is called in the setup table function, it makes no difference.

5.2 Extra Credit: FIFO

Not possible to test if the FIFO algorithm actually work as intended, since the program does not load in process 4 correctly. But know that the used list is managed correctly, by printing the content of the used list it was possible to confirm that the used list contains all entries from the page list.
The thought is that when an unpinned page is found it is inserted at the end of the used list. The used list works as a queue where the oldest page in the used list is the first to be evicted. And inserted at the end of the used list. The next time the used list is used, it will start from the beginning again (confirmed by the use of print statements) and get the first unpinned page which is now the oldest page in the list. This part of the code is not confirmed to work but is strongly believed so on a theoretical basis.

5.3 Faults

The program fails to load all 4 processes, a fault occurs while loading the 4th process. The program can read from the disk correctly, as evidenced by the green light flashing before a process is displayed on the window (see picture 5). But when the 4th process is loaded in, an eviction need to happen, not enough pages left, but when the write-to-disk operation happens an error occurs (indicated by the red light in picture 6). The program writes to the wrong location on the disk. Supported by the print statement that shows that the thrash page pid is given. Also when the offset used for both the read and write operation is printed, it came out as different from each other. Again indicating that the program writes to the wrong place in the drive.

5.4 Improvement

The mark as not present code could be in a helper function but the choice not to was made since it is not much code and it is only used twice. The way all the pages are pinned would be improved, instead of manually doing it a bit operation could be used.

5.5 Completion of project

The write from disk to not work as intended, but everything else does. And as discussed in 5.2, the extra credit for making a better replacement algorithm was done as the FIFO part of the replacement policy is believed to work. Since the used list works as intended.

6. Conclusion

A working OS was made that implements virtual memory for memory management, though it does not include a working swapping pages from disk function. Despite this limitation, the virtual memory aspect of the OS performs as intended.

Sources

[1] (TechTarget), Anderson. Julia, “memory paging”, read: 20.04.2023
URL: What is memory paging? | Definition from TechTarget

[2] (Wikipedia), “Physical address”, read 20.04.2023.
URL: Physical address - Wikipedia

[3] (stackoverflow.com), “What is bit masking”, read: 19.04.2023
URL: c - What is bit masking? - Stack Overflow

[4] (learn-c.org), “bitmasks”, read: 18.04.2023
URL: Bitmasks - Learn C - Free Interactive C Tutorial (learn-c.org)