Operating System Project 4

Virtual Memory

Team 10: 20195008 곽병혁, 20195052 김희수, 20195182 최승훈

May 26, 2024

1 Definition

Virtual memory is a crucial concept in modern operating systems that provides the illusion of a large and continuous memory space for processes. This technique enables the system to execute processes larger than physical memory and ensures higher program execution concurrency compared to solely using physical memory. However, implementing virtual memory is complex, requiring the system to load the necessary parts of a program on the memory and decouple the memory addresses into physical and logical addresses. This project demands the implementation of four major concepts below.

1.1 Paging

Paging is one of the memory management approaches of non-contiguous allocation in physical memory. Current vanilla PintOS does not support virtual memory, which prevents the use of other memory management techniques such as swapping cannot be applied. By implementing this first problem, physical memory does not need to be contiguous, simplifying memory allocation and reducing fragmentation.

1.2 Stack Growth

Some operating systems including our PintOS, limit the size of the stack, leading to inefficiencies and frequent stack overflow exception errors. By allowing the stack to grow dynamically and allocating extra pages as needed, these issues can be mitigated effectively.

1.3 Memory Mapped Files

Memory-mapped files allow the contents of a file to be mapped directly into the virtual address space of a process, facilitating efficient file I/O and shared memory. This task involves implementing two system calls; *mmap* and *munmap*. These system calls enable it to map and unmap files to virtual memory, with pages of the file being loaded into memory on-demand through lazy loading. This improves I/O efficiency by speeding up file access.

1.4 Accessing User Memory

Accessing memory necessitates attentive handling to ensure security and system stability, as invalid or malicious pointers can lead to vulnerabilities. The kernel must be prepared to handle page faults when invalid access occurs or prevent such access from happening in the first place.

2 Algorithm

2.1 Paging

Ahead of Implementing paging, the PintOS does not have a virtual memory system. Hence enabling a virtual memory with memory management by paging should be implemented simultaneously. his goal requires the following tasks; page table, page fault handler, frame table and page replacement algorithm.

[Modified]

- vm/frame.c
- vm/frame.h
- vm/page.c
- vm/page.h
- vm/swap.c
- vm/swap.h
- process.h & process.c
- thread.h
- syscall.h & syscall.c

2.2 Stack Growth

When a new process is created, a stack base pointer is allocated and several arguments are pushed to the stack. When stack access requests arrive, first check whether the address is valid or not, which means the address should be in the current address space. If the address is valid, an extra page allocation to expand the stack size function is needed on demand. Note that releasing a page if it is no longer needed also needs to be possible. In other words, we need to revise the page fault handler, set valid address space and design the page allocating algorithm.

[Modified]

- userprog/process.c
- userprog/exception.c

2.3 Memory Mapped Files

Similar to the previous Project 3, we created two system calls mentioned above. *mmap* opens the file via file descriptor and maps it to the specified virtual address. This mapped page is not loaded to physical memory immediately since this system call follows lazy loading. As a result, it returns mapping ID if mapping is done, -1 otherwise. *munmap* is an reverse of *mmap*. It unmaps the mapped file. If such a file is modified then it should update the file. Lastly, the page fault handler should cover any error that can occur by those system calls as well. One possible scenario is that a page *mmap* trying to map is already connected with the other file. In this case, the page fault handler should handle this situation by lazy loading, in detail, by mapping newly requested file descriptors right after the desired page is released.

[Modified]

- userprog/syscall.c

2.4 Accessing User Memory

Accessing memory can cause various unexpected situations so ensuring safe access is mandatory to the operating system. As paging is applied to the virtual memory, page faults can occur frequently during the system call executions. Hence page fault handler should be well-implemented. Some functions of the page fault handler are updated from the problem above but still have some defects. First, memory address pointer confirmation for any memory access is required. Furthermore, a page lock or pin similar to the semaphore can help prevent the interference of other processes trying to access the same page. Then frame and page allocation refer to the page

status and take proper action. The page replacement algorithm also takes those pinned pages into account and avoids replacing them.

[Modified]

- vm/page.c
- userprog/syscall.c

3 Result

Acknowledgment All source codes are available in HERE.

```
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
pass tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
pass tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
pass tests/vm/mmap-exit
pass tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
pass tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
pass tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
pass tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 109 tests passed.
make[1]: Leaving directory `/home/aiden/pintos/src/vm/build'
aiden@ubuntu:~/pintos/src/vm$
```

Figure 1: Last project, Final result.