SHREY VERMA
190101083
v.shrey@iitg.ac.in

Assignment 4
CS344: Operating Systems Laboratory

18-11-2021

Link to **PPT**: - https://iitgoffice-my.sharepoint.com/:p:/g/personal/v_shrey_iitg_ac_in/EdZWJlVUPMpNj-r3hCELvrIBHInxWtyIzXr1cwXajkEXAw?e=fWwL5J
Link to **Video: -** https://bit.ly/3DvZdOU

What is Memory Management?

In a multiprogramming computer, the operating system resides in a part of memory and the rest is used by multiple processes. The task of subdividing the memory among different processes is called memory management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Why is Memory Management important?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Memory Management system in xv6

xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations.

Paging is used to manage its memory allocations. It uses a page size of 4KB, and a two-level page table structure.

The CPU register CR3 contains a pointer to the page table of the current running process.

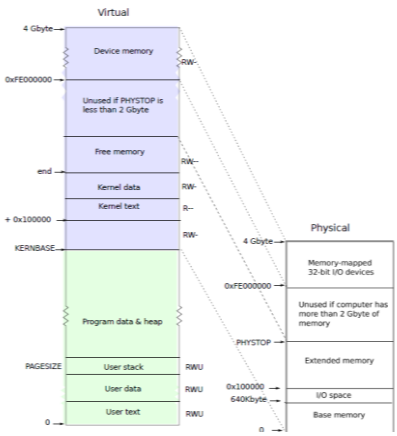
The translation from virtual to physical addresses is performed by the MMU as follows: -

The first 10 bits of a 32-bit virtual address are used to index into a page table directory, which points to a page of the inner page table. The next 10 bits index into the inner page table to locate the page table entry (PTE).The PTE contains a 20-bit physical frame number and flags. Every page table in xv6 has mappings for user pages as well as kernel pages. The part of the page table dealing with kernel pages is the same across all processes.

In the virtual address space of every process, the kernel code and data begin from KERNBASE (2GB in the code), and can go up to a size of PHYSTOP (whose maximum value can be 2GB).

This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory.

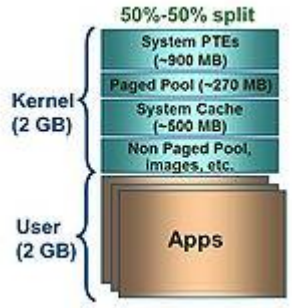
The kernel is mapped into the address space of every process, and the kernel has a mapping for all usable physical memory as well, restricting xv6 to using no more than 2GB of physical memory.



Memory Management system in Windows 10

Each process on 32-bit Microsoft Windows has its own virtual address space that enables addressing up to 4 gigabytes of memory. With 4 GB of physical memory available, 2 GB will be allocated to the kernel and 2 GB to application memory. The kernel mode address space is shared across processes and the application mode access space is allocated for each user process (each process has it's own space).

The 32-bit Windows Operating System addressable memory space is shared between active applications and the kernel as shown in Figure B1.1. The kernel address space includes a System Page Table Entry (PTE) area (kernel memory thread stacks), Paged Pool (page tables, kernel objects), System Cache (file cache, registry), and a Non Paged Pool (images, etc).



Each process on 64-bit Windows has a virtual address space of 8 terabytes.

All threads of a process can access its virtual address space. However, threads cannot access memory that belongs to another process, which protects a process from being corrupted by another process.

Windows 10 supports multiple features, such as shared libraries, file mapping, copy-on-write, and memory compression.

Windows 10 also uses features like Demand Paging with Clustering and a Working Set Model.

Windows is a far more advanced in terms of memory management features when we compare it to xv6.

xv6 is a very simple operating system developed by MIT for educational purposes.

Here are few of the advanced features that Windows offers, which are not implemented in xv6: -

1. **File Mapping**
2. Working Set Model
3. Disc Paging
4. Shared Libraries
5. **Copy-on-write fork**
6. Shared memory
7. **Demand Paging**
8. Automatically extending stacks

Implementation of File Mapping, Copy-on-Write and Demand Paging have been discussed later in this report.

MISSING FEATURE 1
FILE MAPPING

The Problem in xv6: -

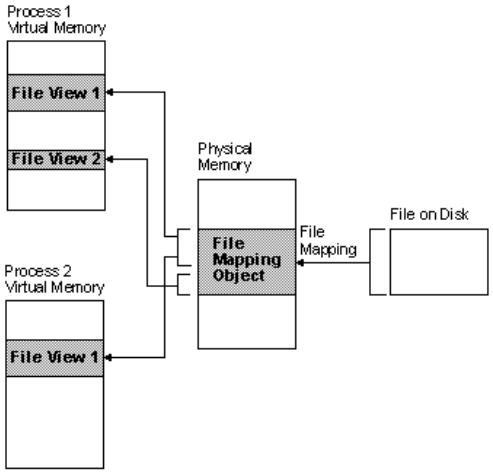
If a program reads a file, the data for that file is copied twice. First, it is copied from the disk to kernel memory by the driver, and then later it is copied from kernel space to user space by the read system call. If the program then sends the data over the network, the data is copied twice more: from user space to kernel space and from kernel space to the network device.

What is File Mapping?

File mapping is the association of a file's contents with a portion of the virtual address space of a process. The system creates a file mapping object (also known as a section object) to maintain this association. A file view is the portion of virtual address space that a process uses to access the file's contents. File mapping allows the process to use both random input and output (I/O) and sequential I/O. It also allows the process to work efficiently with a large data file, such as a database, without having to map the whole file into memory. Multiple processes can also use memory-mapped files to share data.

How does file mapping work in Windows?

Processes read from and write to the file view using pointers, just as they would with dynamically allocated memory. The use of file mapping improves efficiency because the file resides on disk, but the file view resides in memory.



Reference: - https://docs.microsoft.com/en-us/windows/win32/memory/file-mapping

Implementation of File Mapping

Add System Calls: -
Add `mmap` and `munmap` system calls and associated flags in the system
Define VMA (virtual memory area) Per Process: -

Define VMA to keep track what `mmap` has mapped for each process.

The VMA should contain a pointer to a struct file for the file being mapped; `mmap` should increase the file's reference count so that the structure doesn't disappear when the file is closed
Create a `vm_area_struct` with variables such as start address, end address, flags, pointer to file and other required variables.

Since each Process has a fixed-size array of VMAs we have to add `vm_area_struct[n]` inside `struct proc` (stores information related to process).

How to determine VMA start and end address

We design the initial max virtual address of VMA is MAXVA - 2 * PGSIZE, which is 2 pages below the max. The reason behind is first 2 pages are trampoline and trap frame. The VMA list is allocated from top to bottom. The current max VA is adjusted after we create a new VMA. So next allocation's end address can be set to current max.

Implementing `mmap` code

Implement `mmap`: find an unused region in the process's address space in which to map the file, and add a VMA to the process's table of mapped regions. The VMA should contain a pointer to a struct file for the file being mapped; mmap should increase the file's reference count so that the structure doesn't disappear when the file is closed
Lazy page allocation for the File

Idea: -

Fill in the page table lazily, in response to page faults. That is, `mmap` should not allocate physical memory or read the file. Instead, do that in page fault handling code in `usertrap`, as in the lazy page allocation lab. The reason to be lazy is to ensure that `mmap` of a large file is fast, and that `mmap` of a file larger than physical memory is possible

Approach

In `trap handler` add code to cause a page-fault in a mmap-ed region to allocate a page of physical memory, read 4096 bytes of the relevant file into that page, and map it into the user address space.

The change in trap handler should perform following function: -

1. Find which VMA owns the VA. Error out if not found.
2. Allocate a new physical page.
3. Map it into user address space, by installing to user page table.
4. Read one page of the file starting from offset into user space

Implement `int mmap_read(struct file *f, uint64 va, int off, int size);` to read file from offset.

Implement `munmap`

Approach: -

- find the VMA for the address range and unmap the specified page.
- If munmap removes all pages of a previous mmap, it should decrement the reference count of the corresponding struct file.
- If an unmapped page has been modified and the file is mapped `MAP_SHARED`, write the page back to the file

The implemented code should perform following functions: -

- Find VMA to free
- Write back if Shared flag is set
- Remove mappings from a page table
- Fix VMA metadata

Modify exit to `unmap` the process's mapped regions as if `munmap` had been called.

Modify fork to ensure that the child has the same mapped regions as the parent.

For this we will need helper function `void copy_vma(struct vm_area_struct *dst, struct vm_area_struct *src);` it will save all the data of parent's VMA.

And after creation of child update the child's data.

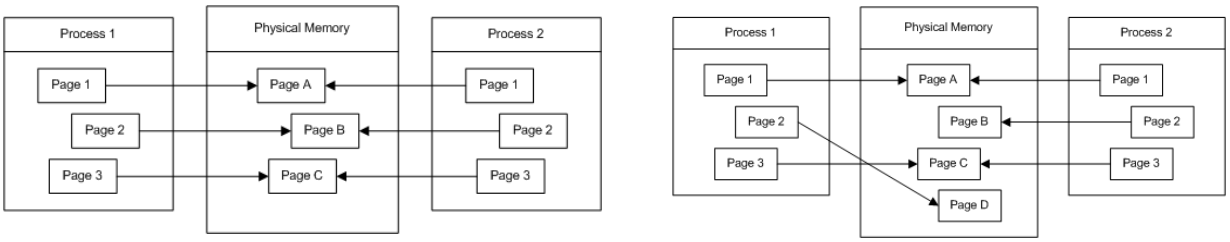
MISSING FEATURE 2
COPY-ON-WRITE

What is Copy-on-write protection?

Copy-on-write protection is an optimization that allows multiple processes to map their virtual address spaces such that they share a physical page until one of the processes modifies the page. This is part of a technique called lazy evaluation, which allows the system to conserve physical memory and time by not performing an operation until absolutely necessary.

How does copy-on-write work in Windows 10?

When multiple instances of the same Windows-based application are loaded, each instance is run in its own protected virtual address space. However, their instance handles typically have the same value. This value represents the base address of the application in its virtual address space. If each instance can be loaded into its default base address, it can map to and share the same physical pages with the other instances, using copy-on-write protection. The system allows these instances to share the same physical pages until one of them modifies a page. If for some reason one of these instances cannot be loaded in the desired base address, it receives its own physical pages.



In this example Page B was initially mapped to both process's pages to optimize memory usage. But when one of the process tried to write, Page D (a copy of Page B) was created and the other process virtual page was linked to it. Therefore, it is not possible for one process to write to a shared physical page and for the other process to see the changes.

The Problem in xv6: -

The `fork()` system call in xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time.

In addition, the copies often waste memory; in many cases neither the parent nor the child modifies a page, so that in principle they could share the same physical memory.

The inefficiency is particularly clear if the child calls `exec()`, since `exec()` will throw away the copied pages, probably without using most of them. On the other hand, if both parent and child use a page, and one or both writes it, a copy is truly needed.

Reference: - https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection

Implementation of Copy-on-Write

The goal of copy-on-write (COW) `fork()` is to defer allocating and copying physical memory pages for the child until the copies are actually needed, if ever.

Idea: -

COW `fork()` creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages. COW `fork()` marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

Add Definition in defs.h: -

```
int uvmcopy(ptable_t, ptable_t, uint64);
```

Implement function uvmcopy in vm.c: -

`fork` calls this function to allocate child memory.

This function should perform the following: -

Given a parent process's page table, copy its memory into a child's page table. Copies both the page table and the physical memory. Set pages flag as read_only

returns 0 on success, -1 on failure. frees any allocated pages on failure.

Modifying trap.c to handle error when one process tries to write: -

Trap should perform the following: -

Copy the original page into the new page, and modify the relevant PTE in faulting process to refer to new copied page.

Freeing the pages when all the reference disappears.

COW `fork()` makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears.

Create void `add_ref` and void `dec_ref`

`add_ref` increases reference index when a new process references the same page.

`dec_ref` decreases reference index when a process exits, it also checks if `index == 0`

and in that case frees the page.

`add_ref` will replace `kalloc` (replace all the calls correspondingly)

`dec_ref` will replace `kfree` (replace all the calls correspondingly)

Modify `copyout()` to take page out only if all the references to this page have disappeared when it encounters a COW page.

MISSING FEATURE 3
DEMAND PAGING

What is Demand Paging?

Demand Paging is key to using the physical memory when a number of processes must run with a combined memory demand exceeding the available physical memory. This is achieved by segmenting the process into smaller tasks. Only the threads corresponding to a specific task are loaded to memory which are required for immediate processing, instead of allowing the entire process to get into the memory. Services that run in the background are typically threads of a larger process. The exception is the Graphics and Hardware Drivers for which memory is reserved and is not available to the Memory manager.

How does Demand Paging work in Windows 10?

The Windows memory manager uses a demand-paging algorithm with clustering to load pages into memory. When a thread receives a page fault, the memory manager loads into memory the faulted page plus a small number of pages preceding and/or following it.
This strategy attempts to minimize the number of paging I/Os a thread will incur. Because programs—especially large ones—tend to execute in small regions of their address space at any given time, loading clusters of virtual pages reduces the number of disk reads.
Page Fault –A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
For page faults that reference data pages in images, the cluster size is three pages. For all other page faults, the cluster size is seven pages.

How does Demand Paging works in an OS?

Without demand paging if a program tried to access a virtual address that is not currently mapped (i.e., the corresponding present bit is zero in the page table), an exception would get generated. This exception is also called a page fault. The corresponding OS exception handler (also called page fault handler) would get to execute and would likely kill the process. With demand paging however, the page fault handler will additionally check if this address is logically present (but physically not present due to its demand paging optimization), and if so, will load it from the disk to the physical memory on-demand.

After the OS loads the page from disk to physical memory, it creates a mapping in the page table, such that a page fault is not generated again on that address, and restarts the faulting instruction. Notice that in doing so, the OS assumes that it is safe to restart a faultidng instruction. On the x86 platform, the hardware guarantees that if an instruction causes an exception, the machine state is left unmodified (as though the instruction never executed) before transferring control to the exception handler. This property of the hardware is called precise exceptions. If the hardware guarantees precise exceptions, it is safe for the OS to return control to the faulting instruction, thus causing it to execute again without changing the semantics of the process logic.

Implementation of Demand Paging

Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing.
After this modification trap handler will allocate a new physical memory and install to user page table once passed safety checks.

Create kernel process

proc.c: The function create kernel `process()` was introduced to the proc.c _le. The implementation of create kernel `process`() is a mix of functions such as fork(), `allocproc`(), and `userinit`().
It works in a similar way to fork(), but with a few differences. While `fork`() obtains the address space, registers, and other data from its parent process, create kernel `process`() does not.
In the same way as `allocproc`() and `userinit`() copy data from the parent process, create kernel process() initialises the data with random values in the places where fork() transfers data from the parent process.
In the create kernel `process`() function, the fork() function's *np->tf = *proc->tf (setting up of trap frame) is replaced with setting up the complete trap frame in the same way `userinit`() does. It sets np->context->eip = (uint)(entrypoint) at the end of create kernel `process`(). This means that when the process starts, it will start at the function entrypoint.
main.c: The create kernel process() function is used twice in the main() function in main to test the aforementioned function: once for the swapin() process and once for the swapout() process.

Implement SwapIn Mechanism

We have a function `swapin()` in proc.c. If there is nothing to swap in, `swapin()` waits on the channel. When it wakes up (when a page needs to be swapped in), it looks for the required file on the disc, locates a free page in the page table, writes the file's stream of bytes to the page, and then deletes the file from disc.

To simulate the swap in mechanism the following functions were implemented: -

```
int swapin(void *page addr, unsigned int offset)
```

Pseudo Code: -

get page table lock.
Find the place to put the page in the page table.
Load contents of page from file into page table.
release page table lock.
delete the file from disk.

Implement SwapOut Mechanism

If there is nothing to swap out, it sleeps on the channel. When it wakes up (when a page needs to be swapped out), it uses `filealloc()` to create a file on the disc, locates the least recently used page, reads the page from the page table as a stream of bytes, saves the bytes to the file created with `filewrite()`, and then frees the page from the page table.
To simulate the swap out mechanism, following functions were implemented: -

```
int swapout(pte_t *mapped_victim_pte, unsigned int offset);
```

Pseudo Code: -

Find the least recently used page.
Save contents of that page to a file.
Take the memory and put it on `kmem.freelist` (in `kalloc.c`)
Get the process to run `kalloc` again (by moving program counter back so that it executes kalloc).

Other things to take care of for proper functioning of implementation: -

Fork

Currently, fork system call copies entire data from the parent process. If the memory gets exhausted during the copying, then swap pages from the parent process to make sure that copy succeeds. Modify: fork, `copyuvm` routines.

Process termination

Free all the swap blocks when the process exits. Modify: `freevm`, `deallocuvm`.

Adjust the `numallocblocks` properly, when the swapped blocks are allocated and deallocated.

Synchronization

Properly use locks similar to given in `spinlock.c` and `sleeplock.c`