Shahriar Attar - 810100186 Sobhan Alaeddini - 810100188 Matin Bazrafshan - 810100093

Repository Link: https://github.com/Shahriar-0/Operating-System-Lab-Projects-F2024
Latest Commit Hash: 4bd14d2eba63b491c5647b50885564af7e422016

Memory Management

1. VMA[1]

The kernel uses VMAs to keep track of a process's memory mappings. For example, a process has one VMA for its code, one VMA for each type of data, one VMA for each distinct memory mapping (if any), and so on.

VMAs are processor-independent structures with permissions and access control flags. Each VMA has a start address and a length, and their sizes are always a multiple of the page size (`PAGE_SIZE`). A VMA consists of several pages, each with an entry in the page table. Memory regions described by VMA are always virtually contiguous, not physically.

Every process has its own virtual memory space, which is divided into several VMAs. Each VMA represents a contiguous virtual memory region and is described by the `vm_area_struct` structure.

VMAs could represent various types of memory regions, such as:

- The contents of a file on disk.
- The memory that contains the program.
- The memory the program uses during execution.
- These memory regions are crucial for the functioning of any program run on Linux.

VMAs have several attributes, including:

- **Start address:** The beginning of the VMA.
- **Length:** The size of the VMA, is always a multiple of the page size (`PAGE_SIZE`).
- **Permissions and access control flags:** These determine how the associated memory can be accessed.

The kernel performs two main operations on VMAs: `lookups` and `modifications`. `Lookups` are used to find the actual memory or where to place the memory on page faults. `Modifications` are important when memory is no longer needed (such as during `exec()`) or when new memory is needed during runtime (such as `stack expansion`, `heap expansion`, `mmap()`, or loading a new library).

We can view the VMAs associated with a process by looking at the `/proc/<pid>/maps` file or using the `pmap` command on a process ID.

[1] Virtual Memory Area

In xv6 the concept of VMA is implemented differently compared to Linux.

- **Memory Allocation:** Unlike Linux, xv6 doesn't have a memory allocator in the kernel. Therefore, it's common to declare a fixed-size array of VMAs and allocate from that array as needed.
- **Address Space Layout:** xv6 loads user code into the very first part of the address space.
- **VMA Structure:** In xv6, a structure corresponding to the VMA records the address, length, permissions, file, etc., for a virtual memory range created by `mmap`.

In xv6, the process's virtual address space is built so that 2GB - 4GB virtual addresses map to 0 to `PHYSTOP` physical addresses. This is done by subtracting and adding the `KERNBASE` constant, which is marked at 2GB. This mapping helps the kernel do the conversion easily, for example, for building and manipulating page tables.

2. Hierarchy Structure in the Paging System

The `Page Directory` is the first level of the hierarchy. It contains pointers to the `Page Tables`, which are the second level. Each entry in the `Page Table` then points to a frame in physical memory. This two-level structure helps manage memory efficiently, especially when the size of the page table is larger than the size of a frame.

Two-level paging, with its hierarchy of `Page Directories` and `Page Tables`, helps reduce memory usage in several ways:

I. **Selective Loading:** Not all `page tables` need to be loaded into memory at once. Only the `Page Directory` and the specific Page Table that is needed for a particular memory reference need to be in memory. This selective loading significantly reduces the amount of memory needed to store the `page tables`.

II. **Efficient Use of Memory:** Two-level paging allows for more efficient use of memory. By using larger page sizes, it can lead to improved memory utilization, reducing the amount of unused memory.

III. **Reduced Memory Overhead:** Multilevel paging can help to reduce the memory overhead associated with the `page table`. This is because each level contains fewer entries, which means that less memory is required to store the `page table`.

IV. **Paging Improves Efficiency:** By dividing memory into pages, the operating system moves pages in and out of memory as needed. Keeping only the frequently used pages reduces the number of page faults, which improves system performance and responsiveness.

3. Bits Meanings in Entries

In a two-level paging system, each entry in the `Page Directory` and `Page Table` is typically 32 bits. Here's a breakdown of what each bit represents:

I. Page Directory Entry:

- The high 20 bits are used to specify the starting physical address of the corresponding `Page Table`. This is because pages are aligned to a multiple of 4 KB, and the address of a page needs 12 bits less than the address bus size. For 32-bit addresses, this ends up being 20 bits.
- The remaining bits can be used for various flags or left unused.

II. Page Table Entry:

- The high 20 bits are used to specify the physical `page number`. This is the physical address where the page is stored in memory.
- The remaining bits can be used for various flags. For example, one common flag is the "Present" bit. If the Present bit is set, the page is available in RAM. If the Present bit is not set, a page fault occurs and the operating system reads the data from storage into RAM before continuing.
- Another common flag is the "Valid-Invalid" bit. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space.

So, the high 20 bits of each entry in both the `Page Directory` and `Page Table` are used to specify addresses (either of a Page Table or a physical page), and the remaining bits are used for various flags or left unused.

Memory Management in xv6

4. `kalloc()`

The `kalloc()` function is used to allocate physical memory. This function is responsible for returning an address of a new, currently unused, page in RAM. If it returns 0, that means there are no available unused pages currently. The allocated physical memory is then mapped into the process's virtual address space.

```
// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.
```

kalloc.c:1

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char* kalloc(void) {
    struct run* r;
    if (kmem.use_lock)
        acquire(&kmem.lock);
    r = kmem.freelist;
    if (r)
        kmem.freelist = r->next;
    if (kmem.use_lock)
        release(&kmem.lock);
    return (char*)r;
}
```

kalloc.c:71

- `struct run* r:` This line declares a pointer `r` to a `struct run`. In xv6, a `struct run` represents a contiguous region of physical memory.
- `if (kmem.use_lock) acquire(&kmem.lock);:` This line acquires a lock if it's being used. This is to ensure that only one process can allocate memory at a time.
- `r = kmem.freelist:` This line gets the first block from the free list. The free list is a list of free (i.e., currently unused) memory blocks.
- `if (r) kmem.freelist = r->next:` If the block `r` exists (i.e., it's not `NULL`), this line moves the free list pointer to the next block. This is because the current block is being allocated, so it's no longer free.
- `if (kmem.use_lock) release(&kmem.lock);:` This line releases the lock if it's being used. This allows other processes to allocate memory.
- `return (char*)r:` Finally, this line returns the block `r`. The block is cast to a `char*` because the `kalloc()` function is designed to return a pointer to a character.

5. `mappages()`

This function is used to map a range of virtual addresses to a range of physical addresses

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t* pgdir, void* va, uint size, uint pa, int perm) {
    char *a, *last;
    pte_t* pte;
    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for (;;) {
        if ((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if (*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if (a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

vm.c:77

```
// Page table/directory entry flags.
#define PTE_P  0x001 // Present
#define PTE_W  0x002 // Writeable
#define PTE_U  0x004 // User
#define PTE_PS 0x080 // Page Size
```

mmu.h:93

- `pgdir`: This is a pointer to the first-level page table, also known as the Page Directory.
- `va`: This is the starting virtual address that needs to be mapped.
- `size`: This is the amount of memory that needs to be mapped.
- `pa`: This is the starting physical address to which the virtual address will be mapped.
- `perm`: This is the permission flag that will be set for this page.

The `mappages()` function modifies the page directory so that `size` bytes of virtual memory starting at the specified virtual address and point to the specified physical address. It will panic if the specified virtual address is already mapped to a physical address. The code will be explained in more detail shortly.

6. walkpgdir()

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t*
walkpgdir(pde_t* pgdir, const void* va, int alloc) {
    pde_t* pde;
    pte_t* pgtab;

    pde = &pgdir[PDX(va)];
    if (*pde & PTE_P) {
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    }
    else {
        if (!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
        // The permissions here are overly generous, but they can
        // be further restricted by the permissions in the page table
        // entries, if necessary.
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

vm.c:30

- pgdir: This is a pointer to the first-level page table, also known as the Page Directory.
- va: This is the virtual address for which we want to find the corresponding page table entry.
- alloc: This is a flag that indicates whether the function should allocate a new page table if one does not already exist.

The walkpgdir() function takes a page directory and a virtual address and returns a pointer to the page table entry for that virtual address. If the alloc parameter is set, the function will allocate any needed second-level page tables.

a step-by-step breakdown of how it works:

I. The function first extracts the index into the page directory from the virtual address using the PDX() macro.
II. It uses this index to get a pointer to the corresponding page directory entry.
III. If the page directory entry is present (i.e., the page table exists), it gets a pointer to the page table.

IV. If the page directory entry is not present and the `alloc` parameter is set, it allocates a new page table, initializes it, and updates the page directory entry.
V. Finally, it returns a pointer to the page table entry for the virtual address.

The `walkpgdir()` function in xv6 imitates the hardware action of *translating a virtual address to a physical address*.

In hardware, this translation is performed by the MMU[2] using a process called `paging`. The MMU uses a page table, which is a data structure that stores the mapping between virtual and physical addresses.

The `walkpgdir()` function essentially performs the same operation in software. It takes a virtual address and a page directory as input and returns a pointer to the PTE[3] for that virtual address. This PTE contains the physical address that corresponds to the input virtual address.

Now after understanding how `walkpgdir` we go back to `mappages` to see how it exactly works:

- `a = (char*)PGROUNDDOWN((uint)va);`: This line rounds down the virtual address `va` to the nearest page boundary and assigns it to `a`. The `PGROUNDDOWN` macro is used to perform the rounding.
- `last = (char*)PGROUNDDOWN(((uint)va) + size - 1);`: This line calculates the last virtual address that needs to be mapped by adding `size - 1` to `va`, rounding down to the nearest page boundary, and assigning the result to `last`.
- `if ((pte = walkpgdir(pgdir, a, 1)) == 0) return -1:` This line calls the `walkpgdir()` function to get the page table entry for the virtual address `a`. If `walkpgdir()` returns `NULL` (indicating that the page table entry could not be found or created), the function returns `-1`.
- `if (*pte & PTE_P) panic("remap");`: This line checks if the `PTE_P` flag is set in the page table entry. If it is, this means that the virtual address is already mapped, and the function panics with the message "remap".
- `*pte = pa | perm | PTE_P:` This line sets the page table entry to the physical address `pa`, combined with the permissions `perm` and the `PTE_P` flag. This effectively maps the virtual address to the physical address.
- `if (a == last) break:` This line checks if `a` is equal to `last`. If it is, this means that all the required virtual addresses have been mapped, and the loop is broken.
- `a += PGSIZE; pa += PGSIZE:` These lines increment `a` and `pa` by the size of a page. This moves on to the next virtual and physical addresses that need to be mapped.
- `return 0:` Finally, if the function has not encountered any errors, it returns `0` to indicate success.

[2] Memory Management Unit
[3] page table entry

7. `mappages()` and `allocuvm()`

The `mapapges()` wes explained before, now let’s take a look at `allocuvm()`

```c
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int allocuvm(pde_t* pgdir, uint oldsz, uint newsz) {
    char* mem;
    uint a;
    if (newsz >= KERNBASE)
        return 0;
    if (newsz < oldsz)
        return oldsz;
    a = PGROUNDUP(oldsz);
    for (; a < newsz; a += PGSIZE) {
        mem = kalloc();
        if (mem == 0) {
            cprintf("allocuvm out of memory\n");
            deallocuvm(pgdir, newsz, oldsz);
            return 0;
        }
        memset(mem, 0, PGSIZE);
        if (mappages(pgdir,(char*)a, PGSIZE, V2P(mem),PTE_W|PTE_U) < 0) {
            cprintf("allocuvm out of memory (2)\n");
            deallocuvm(pgdir, newsz, oldsz);
            kfree(mem);
            return 0;
        }
    }
    return newsz;
}
```

- `if (newsz >= KERNBASE) return 0:` This line checks if the new size `newsz` is greater than or equal to `KERNBASE`. If it is, the function returns `0`. This is because `KERNBASE` is the boundary between the user and kernel parts of the address space.
- `if (newsz < oldsz) return oldsz;`: This line checks if `newsz` is less than `oldsz`. If it is, the function returns `oldsz`. This is because `allocuvm()` is designed to increase the size of the process's memory, not decrease it.
- `a = PGROUNDUP(oldsz);`: This line rounds up `oldsz` to the nearest page boundary and assigns it to `a`.
- `for (; a < newsz; a += PGSIZE)`: This line starts a loop that will continue until `a` is less than `newsz`. In each iteration, `a` is incremented by the size of a page.
- `mem = kalloc();`: This line calls the `kalloc()` function to allocate a new page of physical memory.

- `if (mem == 0) { ... }`: This block checks if `kalloc()` returned `0`, which means that it failed to allocate a new page. If it did, the function prints an error message, deallocates any memory that was allocated during this call to `allocuvm()`, and returns `0`.
- `memset(mem, 0, PGSIZE);`: Sets the first `PGSIZE` bytes of the memory area pointed to by `mem` to `0`. This is to ensure that the newly allocated memory is cleared.
- `if (mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)`: This block calls the `mappages()` function to map the newly allocated page into the process's virtual address space. If `mappages()` fails, the function prints an error message, deallocates any memory that was allocated during this call to `allocuvm()`, frees the newly allocated page, and returns `0`.
- `return newsz:` If the function has not encountered any errors, it returns `newsz`.

8. Loading in Memory by `exec()`

The `exec` system call in xv6 is used to replace the current running program with a new one. A step-by-step explanation of how a program is loaded into memory when the `exec` system call is called:

I. **Wipe Out Memory State:** The `exec` system call first wipes out the memory state of the calling process.

II. **Find the Program File:** It then goes to the filesystem to find the program file requested.

III. **Allocate New Page Table:** Then it allocates a new page table with no user mappings with `setupkvm()`.

IV. **Allocate Memory for Each ELF Segment:** It allocates memory for each ELF[4] segment with `allocuvm()`. The `allocuvm()` function is responsible for increasing the user's virtual memory in a specific page directory. It does this by allocating new pages of physical memory and mapping them into the process's virtual address space.

V. **Load Each Segment into Memory:** It loads each segment into memory with `loaduvm()`. The `loaduvm()` function uses `walkpgdir` to find the physical address of the allocated memory at which to write each page of the ELF segment, and `readi` to read from the file.

VI. **Initialize Register State:** `exec` also initializes the register state, including the `PC` and `EIP` to program entry and also `ESP` to the top of userstack.

VII. **Updating Page Table:** It then uses `switchuvm()` to update the page table in hardware and then uses `freevm()` to remove the previous page table from memory.

[4] Executable and Linkable Format

Shared Memory

To store the shared memory address we added an attribute to our proc struct named shmemaddr so our new struct would look like this:

```
struct proc {
    uint sz;                     // Size of process memory (bytes)
    pde_t* pgdir;                // Page table
    char* kstack;                // Bottom of kernel stack for this process
    enum procstate state;        // Process state
    int pid;                     // Process ID
    struct proc* parent;         // Parent process
    struct trapframe* tf;        // Trap frame for current syscall
    struct context* context;     // swtch() here to run process
    void* chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file* ofile[NOFILE]; // Open files
    struct inode* cwd;           // Current directory
    char name[16];               // Process name (debugging)
    uint ctime;                  // created time
    struct schedparams sched;    // scheduling parameters
    uint shmemaddr;              // address of shared-memory
};
```

proc.h:83

For implementing a shared memory now we define two new structs to help us with that.

1. shpage
   A. id: is a unique identifier and used to define the page.
   B. n_access: number of process which have access to the page
   C. physicalAddr: physical address of the page, will be used for mappings

2. shmemtable
   A. struct shpage pages[NSHPAGE]: an array of pages.
   B. struct spinlock lock: a spinlock to avoid conflicts and race conditions.

```
#define NSHPAGE   64
struct shpage {
    int id;
    int n_access;
    uint physicalAddr;
};
```

```
struct shmemtable {
    struct shpage pages[NSHPAGE];
    struct spinlock lock;
} shmemtable;
```

vm.c:370

The `shmemtable` consists of some tables and a locking mechanism to ensure protection when multiple processes try to access this shared memory, it uses the `spinlock` method which was explained in previous reports.

Now that we have defined a struct for shared memory we implement two system calls that would open and close this shared memory for us.

System calls:

I. `openshmem`

This system call is used when a process tries to create or use a shared memory. Each shared memory is identified with an ID stored in `shpage` struct. `shpage` is a struct for every shared page. In this project, each shared memory consists of only one page(4KB). After calling this function two scenarios could happen:

1. A shared memory with a given ID already exists, then we need to do these things:
    A. Adjust a virtual memory, starting with the address of process size. The variable `sz` is used to determine a process size, so if we want to expand the size of a process the starting address will be `sz`.
    B. Map virtual memory to physical memory. The physical memory address is saved in `shpage` struct, so by finding a proper `page` in `shmemtable` with the corresponding `id`, we can use it for do the mapping.
    C. increase the size of the process to avoid `remap panic`, if the process tries to expand its size, it must use virtual addresses which start after shared memory.
2. If a shared memory with a given ID does not exist, then we need to do these things:
    A. First, we need to allocate memory for our shared memory.
    B. After allocation, we need to map the virtual address to its physical address using the `V2P` macro.
    C. We assign the `id` to `shpage` struct.
    D. The rest of the procedure is the same as the first scenario.

```
 char* openshmem(int id) {
    struct proc* proc = myproc();
    acquire(&shmemtable.lock);
    int size = PGSIZE;
    for (int i = 0; i < NSHPAGE; i++) {
        if (shmemtable.pages[i].id == id) {
            shmemtable.pages[i].n_access++;
            char* vaddr = (char*)PGROUNDUP(proc->sz);
            if (mappages(proc->pgdir, vaddr, PGSIZE,
shmemtable.pages[i].physicalAddr, PTE_W | PTE_U) < 0)
                return -1;
            proc->shmemaddr = (uint)vaddr;
            proc->sz += size;
            release(&shmemtable.lock);
            return vaddr;
        }
    }
    int pgidx = -1;
    for (int i = 0; i < NSHPAGE; i++) {
        if (shmemtable.pages[i].id == 0) {
            shmemtable.pages[i].id = id;
            pgidx = i;
            break;
        }
    }
    char* paddr = kalloc();

    memset(paddr, 0, PGSIZE);
    char* vaddr = (char*)PGROUNDUP(proc->sz);
    shmemtable.pages[pgidx].physicalAddr = (uint)V2P(paddr);

    if (mappages(proc->pgdir, vaddr, PGSIZE,
shmemtable.pages[pgidx].physicalAddr, PTE_W | PTE_U) < 0)
        return -1;

    shmemtable.pages[pgidx].n_access++;
    proc->shmemaddr = (uint)vaddr;
    proc->sz += size;
    release(&shmemtable.lock);
    return vaddr;
}
```

II. `closeshmem`

This system call is a little tricky. First look at this code which is placed in `deallocuvm` which will be called by `freevm` which will be called by `wait`!

```
for (; a < oldsz; a += PGSIZE) {
        pte = walkpgdir(pgdir, (char*)a, 0);
        if (!pte)
            a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
        else if ((*pte & PTE_P) != 0) {
            pa = PTE_ADDR(*pte);
            if (pa == 0)
                panic("kfree");
            char* v = P2V(pa);
            kfree(v);
            *pte = 0;
        }
    }
```

vm.c:248

When a child process exits, the parent process frees the child process' memory, if the child process uses a shared memory, the physical memory of shared memory is linked to the virtual address of that process, so by freeing its memory, the shared memory contents will be lost. So when a process tries to close its shared memory, we remove the link of its virtual memory to the shared memory. We achieve this goal by setting the `pte` of the shared memory(whose address is saved in `shmemaddr` in the process) to zero. so when the parent process tries to free the child process, the shared memory address is no longer valid for the child process.

In addition to the given information, we also need to do the following:

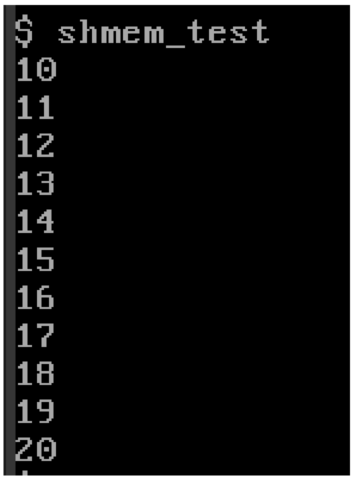
A. check if the number of access to the memory reaches zero, we remove that page from `shmemtable` and make it available for further needs
B. Reduce the size of the process by shared memory size.

```
int closeshmem(int id) {
    struct proc* proc = myproc();
    int size = PGSIZE;
    acquire(&shmemtable.lock);

    for (int i = 0; i < NSHPAGE; i++) {
        if (shmemtable.pages[i].id == id) {
            shmemtable.pages[i].n_access--;

            uint a = (uint)PGROUNDUP(proc->shmemaddr);
            pte_t* pte = walkpgdir(proc->pgdir, (char*)a, 0);
            *pte = 0;

            if (shmemtable.pages[i].n_access == 0)
                shmemtable.pages[i].id = 0;

            release(&shmemtable.lock);
            return 0;
        }
    }

    release(&shmemtable.lock);
    cprintf("No shared memory with this ID.\n");
    return -1;
}
```

Testing

To check our new-defined memory we wrote this user program. In this program, we change the value of shared memory each time by 1 and then we print it in every process

```
#define NPROCESS 10
#define ID       25
void test_shmem(void) {
    char* adr = openshmem(ID);
    adr[0] = 10;
    printf(1, "%d\n", adr[0]);

    for (int i = 0; i < NPROCESS; i++) {
        if (fork() == 0) {
            sleep(100 * i);
            char* adrs = openshmem(ID);
            adrs[0] += 1;
            printf(1, "%d\n", adrs[0]);
            closeshmem(ID);
            exit();
        }
    }
    for (int i = 0; i < NPROCESS; i++)
        wait();
    closeshmem(ID);
}
int main(int argc, char* argv[]) {
    test_shmem();
    exit();
}
```



```
$ shmem_test
10
11
12
13
14
15
16
17
18
19
20
```

As can be seen, although we change the value in each of the processes by one, they are the same and the results in all of them is consistent.