

Operating Systems Lab Assignment - 3

Group 22

Rishav Mondal (190101072)
Siddharth Charan (190101085)
Shrey Verma (190101083)
Karan Raj Sharma (190101043)

1 Lazy Memory Allocation

1.1 TASK 1 : Eliminate allocation from sbrk()

Our first task was to delete page allocation from the `sbrk(n)` system call implementation, which is in the function `sys_sbrk()` in `sysproc.c`.

We got a `sysproc.c` file as part of this lab. We used this file and replaced this file in place of original one. You can see a screenshot of one of the important code segments provided below :

```
int
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    if(growproc(n) < 0)
        return -1;
    return addr;
}
```

In this file(see the attached screenshot for better clarity), our new `sbrk(n)` will just increment the process size by `n` and return the old size. It does not allocate memory, because the call to `growproc()` is commented out. However, it still increases `proc->sz` by `n` to trick the process into believing that it has the necessary memory requested with it. When we ran the code we got this faulty output as shown below:

```
$ echo hl
pid 4 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc
$
```

The “pid 3 sh: trap...” message printed on the terminal is from the kernel trap handler in `trap.c`; it has caught a page fault (trap 14, or `T_PGFLT`), which the xv6 kernel does not know how to handle. The “addr 0x4004” indicates the virtual address location that caused the page fault.

1.2 TASK 2 : Lazy Allocation

In this task we were asked to 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. We were asked to allocate a new memory page, add suitable page table entries, and return from the trap, so that the process can avoid the page fault the next time it runs.

1. `proc.h`: Here we added the variable `oldsz` in struct `proc` as unsigned integer.

2. `proc.c`: Here we initialized the variable `oldsz` with value 0, using line `p->oldsz = 0` in the `allocproc()` function.
3. `trap.c`: Here, we declared mappages in `trap.c` after removing static keyword for mappages function in `vm.c`. You can see the code snippet attached below :

```
//PAGEBREAK: 13
default:
if(myproc() == 0 || (tf->cs&3) == 0){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
        tf->trapno, cpuid(), tf->eip, rcr2());
    panic("trap");
}
if(rcr2() > myproc()->sz){
    cprintf("Unhandled Page Fault \n");
}
else {
    if(tf->trapno == T_PGFLT)
    {
        char *mem;
        uint a;
        if(myproc()->sz < myproc()->oldsz){
            return;
        }

        a = PGROUNDOWN(rcr2());
        mem = kalloc();
        if(mem == 0){
            cprintf("allocvm out of memory \n");
            myproc()->killed = 1;
            return;
        }

        memset(mem, 0, PGSIZE);
        mappages(myproc()->pgdir, (char*) a, PGSIZE, V2P(mem), PTE_W|PTE_U);
        break;
    }
}
```

Here, we modified the default page trap 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 the user space to let the process continue executing. We first check if the trapped fault is indeed a page fault using the condition `(tf->trapno == T_PGFLT)`. To avoid the `cprintf` statement, we return if `kalloc()` returns 0. After recompiling, we find `echo hi` properly working. You can see this in screenshot. Here, we have run a couple of other shell commands like `ls` and `cat` in addition to `echo` to check that our output is properly showing.

2 xv6 Memory Management

Here we answer some of the questions that we were asked to answer:

1. How does the kernel know which physical pages are used and unused?

Our processor xv6 maintains a record of free physical memory, to be used by the processes that are to run. It maintains a linked list of all physical pages currently available and deletes newly allocated pages from the list, and adds freed pages back to the linked list. The allocator (implemented in `kalloc.c`) maintains a free list of memory addresses of main memory pages that are available for data storage. This process is true for other similar processors also. In short kernel explicitly maintains a linked list of all the unused pages and hence it knows which pages are used and which are unused.

2. What data structures are used to answer this question?

Each and every struct run denotes a free page's list element. It stores the page's run structure in the free page itself, since it is currently empty. The linked list run is protected by a spin lock. The lock needs to be acquired before changing the list. This prevents simultaneous access of the link list by multiple programs. The list and the lock are enclosed in a structure to emphasize that the lock protects the **linked list**. So, we use Singly linked list with no data and a single pointer to the next code.

We can see the implementation of this in the code snippet below:

```

struct run {
    struct run *next;
};

struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;

```

Figure 1: Data structures used

3. Where do these reside?

The allocator implementation and the corresponding data structures can be found in file `kalloc.c`. You can see two structs in above code snippet. Each struct `run` represents a free page's list element. One is to define the linked list and other is main data structure containing a single pointer and a singly linked list.

4. Does xv6 memory mechanism limit the number of user processes?

Because xv6 does not enable paging to disc by default, that implies that if it allocates memory for the userprocess (using the `sbrk()` syscall, which is utilised by `malloc()` in userspace), it retains it in physical memory until the process is terminated. Because physical memory is limited (from end to `PHYSTOP`), only a finite number of processes can be stored in it at any given time. We won't be able to assign memory to any additional processes since the newly launched process will demand free physical memory (not available due to currently running processes).

5. If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

With respect to the xv6 operating system, the minimum number of processes running at the same time will be 1 (`theinit()` process). `Init` is a daemon process that continues running until the system is shut down.

And, we have set the variable `NPROC` to 64 in the file `param.h`. So, the maximum number of processes is 64 (including both kernel and user process) at a given instance. This limit has been set to avoid memory overflow due to a higher number of running processes.

2.1 TASK 1 : Kernel Processes

1. proc.c: The function `create_kernel_process()` was introduced to the `proc.c` file. 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. For further information on the function `create_kernel_process()`, see the screenshot below.
2. main.c: The `create_kernel_process()` function is used twice in the `main()` function in `main` to test the aforementioned function: once for the `swpin()` process and once for the `swapout()` process.

```

void
create_kernel_process(const char *name, void (*entrypoint)())
{
    struct proc *np;
    struct qnode *qn;

    if ((np = allocproc()) == 0) panic("Failing allocating kernel process");

    qn = freenode;
    freenode = freenode->next;

    if(freenode != 0) {
        freenode->prev = 0;
    }

    if((np->pgdir = setupkvm()) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        panic("Failed setup pgdir for kernel process");
    }

    np->sz = PGSIZE;
    np->parent = initproc;
    memset(np->tf, 0, sizeof(*np->tf));
    np->tf->cs = (SEG_UCODE << 3) | DPL_USER;
    np->tf->ds = (SEG_UDATA << 3) | DPL_USER;
    np->tf->es = np->tf->ds;
    np->tf->ss = np->tf->ds;
    np->tf->eflags = FL_IF;
    np->tf->esp = PGSIZE;

    // beginning of initcode.S
    np->tf->eip = 0;

    // Set eax = 0 so that fork return 0 in the child
    np->tf->eax = 0;
    np->cwd = namei("/");
    safestrcpy(np->name, name, sizeof(name));
    qn->p = np;

    // lock to force the compiler to emit the np-state write last.
    acquire(&table.lock);
    np->context->eip = (uint)entrypoint;
    np->state = RUNNABLE;
    release(&table.lock);
}

```

Figure 2: create_kernel_process() function

Note that in the submission, both functions are commented out.

2.2 TASK 2 : Swapping Out Mechanism

We have a swap_out() function.

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()(defined in file.c) 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() (defined in file.c), and then frees the page from the page table. To simulate the swap out mechanism, we used the functions listed below.

```

int swap_out(pte_t *mapped_victim_pte, unsigned int offset)
{
    struct swap_info_struct *p = &swap_info[0];
    int file_offset = offset + 1, retval = -1;
    uint old_offset;
    char *kernel_addr = P2V(PTE_ADDR(*mapped_victim_pte));

    if(p->swap_file == NULL)
        return -1;

    old_offset = p->swap_file->off;

    // Quick and dirty hack for now. Need a lock-protected state variable later
    myproc()->pages_swapped_out++;

    // Write contents to swapfile
    p->swap_file->off = (unsigned int)(file_offset * PGSIZE);
    retval = filewrite(p->swap_file, kernel_addr, PGSIZE);
    p->swap_file->off = old_offset;

    return retval;
}

```

2.3 TASK 3 : Swapping In Mechanism

We have a function `swap_in()` in `proc.c`. If there is nothing to swap in, `swap_in()` 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, we used the functions listed below:

- `int swap_in(void *page_addr, unsigned int offset)`
 - It estimates the swapfile offset when a needed page needs to be swapped in.
 - After that, we lower the value of the variable pages swapped out.
 - The `P2V()` function transforms virtual addresses to physical addresses. - The victim frame address is converted to a physical frame using the `P2V()` method.
 - After that, we use the `fileread()` function to read pages from the given file.

```
int swap_in(void *page_addr, unsigned int offset)
{
    struct swap_info_struct *p = &swap_info[0];
    int file_offset = offset + 1, retval = -1;
    uint old_offset;

    if (p->swap_file == NULL)
        // SWAPFILE pointer not set yet with ksetswapfileptr() system call
        return -1;

    old_offset = p->swap_file->off;

    // Quick and dirty hack for now. Need a lock-protected state variable later
    myproc()->pages_swapped_out--;

    // Read contents from swapfile
    p->swap_file->off = (unsigned int)(file_offset * PGSIZE);
    retval = fileread(p->swap_file, page_addr, PGSIZE);
    p->swap_file->off = old_offset;

    return retval;
}
```

- `void map_address(pde_t *pgdir, uint addr)`
 - The function above converts a physical page into a virtual address. If the page table entry points to a swapped block, restore the page's content from the swapped block and release it.
 - Create a physical page with `kalloc`.
 - Create a virtual page from a physical page (`addr`).
 - Set the page's access bit (the last 12 bits of the physical and virtual pages are the same), so that they share access bit.

- void handle_pgfault(void)

The above function handles page faults , when the page is not found in the page table.

```
// page fault handler
void handle_pgfault()
{

    unsigned addr;
    struct proc *curproc = myprocxv6();
    asm volatile ("movl %%cr2, %0 \n\t" : "=r" (addr));
    addr &= ~0xfff;
    map_address(curproc->pgdir, addr);

}
```

2.4 TASK 4 : Sanity Test

Changed the value of PHYSTOP in memlayout.h.

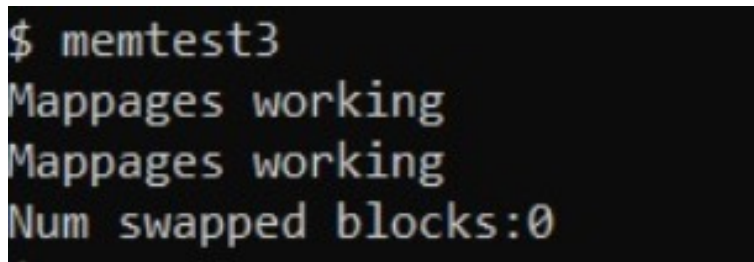
- Initial statement in memlayout.h: #define PHYSTOP 0xE000000
- After changing the statement: #define PHYSTOP 0x4000000

This prevents the kernel from storing all of the processes' memory in RAM; and allows us to see how the paging system works.

We generated two test files: one for **checking processes with memory allocation and counting their swap count**, and another for **checking whether the swap count is zero or not**.

Please refer to the below screenshots for seeing the obtained outputs.

1. Test with no memory allocation task:



```
$ memtest3
Mappages working
Mappages working
Num swapped blocks:0
```

2. Test output with 4kb memory allocation:

```
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
mem ok 17
$
```

The output “mem ok 17” signifies that the memory allocation was successful. We used the `bstat` syscall which helps us to retrieve the global count of swapped pages.