

# CS344 Operating Systems Lab

## Assignment 3

### GROUP 20

Anjali Godara -180101008  
Niharika Bhamer -180101048  
Varhade Amey Anant -180101087  
Tanmay Jain -180123050

## Part A

### Lazy Memory Allocation

#### 1. Eliminate Allocation from `sbrk()`

In this part, we were supposed to eliminate page allocation from `sbrk()`. Whenever a process needs extra memory allotted to it, it calls `sys_sbrk()`. In this pre-defined function, `growproc()` function is called, which one by one starts allotting entries in page tables. In a case where there isn't enough memory to cater to the request, `growproc()` would return -1, hence indicating that the allocation failed. So to stop the allocation of pages in `sbrk()`, we comment out the lines where `growproc()` is called and instead increase the size of the current process by `n` by `n` and return the old size. It does not allocate memory, instead just tricks the process into believing it.

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

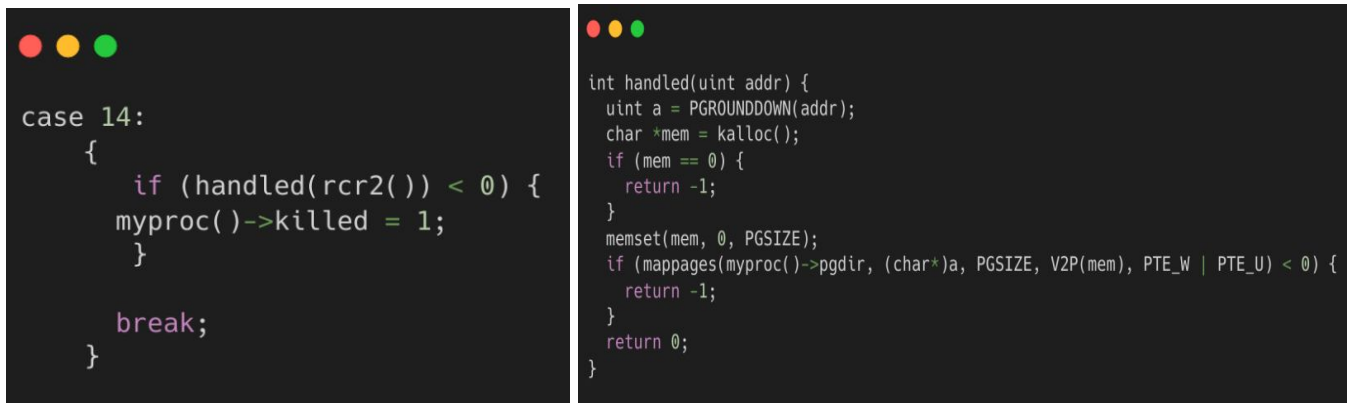
    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    myproc()->sz += n;

    // if(growproc(n) < 0)
    //     return -1;
    return addr;
}
```

## 2.Lazy Memory Allocation

Page allocation to processes were stopped in the above step, hence now when the same process would run and request for certain pages it needs, it would result in page faults.

Page fault errors aren't handled by the default xv6 system. So we have to add a separate case in **trap.c** for the same.



```
case 14:
{
    if (handled(rcr2()) < 0) {
        myproc()->killed = 1;
    }

    break;
}

int handled(uint addr) {
    uint a = PGROUNDDOWN(addr);
    char *mem = kalloc();
    if (mem == 0) {
        return -1;
    }
    memset(mem, 0, PGSIZE);
    if (mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0) {
        return -1;
    }
    return 0;
}
```

The first code snippet is the change made in trap.c, while looking for various cases handled by checking `tf->trapno`.

In **traps.h** variable `T_PGFLT` is given value 14, dedicated to lookout for page faults. Whenever a page fault occurs, `tf->trapno` value is set to 14, and function `handled()` is called. It takes input as the address as to where the page fault occurred, given by `rcr2()` function. In case a page fault can not be dealt with, `handled` returns -1 and the process is killed.

The second code snippet describes the function `handled()`, taking as input the address where the fault originated. A variable stores the round-down value of the address, and `kalloc()` is called to allocate memory. In case there is no memory left to be assigned, `handled` returns -1, if the allocation is successful,

we erase all contents stored in it using `memset`. Next, the allotted memory must be mapped to an entry in the page table, if there is no space left in the page table, `handled()` would again return -1. For each `n` bytes a process requested to be allocated to it,  $(n/\text{page-size})$  faults would be generated and taken care of by `handled()`. To be able to use `mappages()` in trap.c, we removed the static part of it's declaration in vm.c.

## Part B

### Paging

### **How does the kernel know which physical pages are used and unused?**

The list of free pages is maintained as a linked list, with the pointer to the next page being stored within the page itself. Pages are added to this list upon initialization, or on freeing them up.

### **What data structures are used to answer this question?**

Struct `run` defined with just a pointer of the same type is used in struct `kmem` to declare a pointer of type struct `run`, `freelist`. This points to the beginning of a free page.

### **Where do these reside?**

The linked list and the structs defined above are all declared in the `kalloc.c` file.

### **Does the xv6 memory mechanism limit the number of user processes?**

Yes, the memory mechanism limits the number of processes. The max number is defined by `NPROC`, which is set to 64 by default.

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

When the xv6 operating system boots up, there is only one process named `initproc`. Hence, the answer is 1.

## **1. Kernel Process**

A kernel process exists only in the kernel protection domain. The function `void create_kernel_process(const char *name, void (*entrypoint)());` is implemented in the file `proc.c`. Implementation of `create_kernel_process()` is similar to that of `fork()`, `allocproc()`, following the same structure with a few key differences.



```
if ((np = allocproc()) == 0)
    panic("Failed to allocate kernel process.");
```

In `fork()`, it copies the addresses, registers, etc. from the parent process, whereas **`create_kernel_process()`** sets up data from scratch similar to **`allocproc()`**. The trap frame values of a child process in `fork()` are set to the same value as its parent process, while `create_kernel_process()` sets it as follows.

```

np->sz = PGSIZE;
np->parent = initproc; // parent is the first process.
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;
np->tf->eip = 0; // beginning of initcode.S

// Clear %eax so that fork return 0 in the child
np->tf->eax = 0;

```

At the end of `create_kernel_process()`, it sets `np->context->eip` to the entrypoint function pointer that was provided in the parameter. This means that the process will start running at the function entrypoint when it starts.

```

acquire(&ptable.lock);
np->context->eip = (uint)entrypoint;
np->state = RUNNABLE;
release(&ptable.lock);

```

This function creates a kernel process and adds it to the process table `ptable` from where it will be picked up by the scheduler appropriately.

## Task 2: swapping out mechanism

For this the major code is in the `proc.c` file. We have implemented a queue which contains the processes that are waiting for their turn for the allotment of memory as earlier there was no free page available. All the functions related to this queue have been implemented which can also be seen in the code snippets attached below:

```

void
pinit(void)
{
    initlock(&ptable.lock, "ptable");
    initlock(&readyqueue.lock, "readyqueue");
    initlock(&sleeping_channel_lock, "sleeping_channel");
    initlock(&readyqueue2.lock, "readyqueue2");
}

```

```

struct ready_queue{
    struct spinlock lock;
    struct proc* queue[NPROC];
    int start;
    int end;
};

```

```

struct proc* kpop(){

    acquire(&readyqueue.lock);
    if(readyqueue.start==readyqueue.end){
        release(&readyqueue.lock);
        return 0;
    }
    struct proc *p=readyqueue.queue[readyqueue.start];
    (readyqueue.start)++;
    (readyqueue.start)%=NPROC;
    release(&readyqueue.lock);

    return p;
}

```

```

int kpush(struct proc *p){

    acquire(&readyqueue.lock);
    if((readyqueue.end+1)%NPROC==readyqueue.start){
        release(&readyqueue.lock);
        return 0;
    }
    readyqueue.queue[readyqueue.end]=p;
    readyqueue.end++;
    (readyqueue.end)%=NPROC;
    release(&readyqueue.lock);

    return 1;
}

```

```

void
userinit(void)
{
    acquire(&readyqueue.lock);
    readyqueue.start=0;
    readyqueue.end=0;
    release(&readyqueue.lock);
}

```

First of all the process demands for memory from the kalloc function. If the kalloc can not provide sufficient memory, the processes are shifted to sleeping mode by the sleeping channel via notifying the allocvm() function. We created sleeping channels for the swap functions. These channels are for the respective processes to sleep on when there is nothing to swap-in/swap-out. The swap function also contains a spinlock to ensure mutual exclusion to the swap functions(so that only one process can swap-in or swap-out at one time). Sleeping\_channel\_count is also there for the corner cases when the os boots.

```

struct spinlock sleeping_channel_lock;
int sleeping_channel_count=0;
char * sleeping_channel;

```

```

void
kfree(char *v)
{
    struct running *r;

    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
        panic("kfree");
    }

    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = (struct running*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    if(kmem.use_lock)
        release(&kmem.lock);

    //Wake up processes sleeping on sleeping channel.
    if(kmem.use_lock)
        acquire(&sleeping_channel_lock);
    if(sleeping_channel_count){
        wakeup(sleeping_channel);
        sleeping_channel_count=0;
    }
    if(kmem.use_lock)
        release(&sleeping_channel_lock);
}

```

```

if(mem == 0){
    deallocvm(pgdir, newsz, oldsz);

    //SLEEP
    myproc()->state=SLEEPING;
    acquire(&sleeping_channel_lock);
    myproc()->chan=sleeping_channel;
    sleeping_channel_count++;
    release(&sleeping_channel_lock);

    kpush(myproc());
    if(!swap_out_present){
        swap_out_present=1;
        create_kernel_process("swap_out_process", &swap_out);
    }

    return 0;
}
memset(mem, 0, PGSIZE);
if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) <
0){
    cprintf("allocvm out of memory (2)\n");
    deallocvm(pgdir, newsz, oldsz);
    kfree(mem);
    return 0;
}

```

Then there is a modification in kalloc.c file under the kfree() function which triggers the wakeup() function. This awakens the sleeping processes in the queue. Now in the swap\_out() function we run a while loop that takes out the sleeping processes until the queue is empty. Now the pages that need to be replaced also need a file to write their content into. We have implemented all the functions related to the file reading and writing etc required for this process. LRU policy is being followed to swap out the pages. At last there are some instructions for the termination of the swap\_out function. Following are the snippets for the file reading and writing functions.

```

int write_process(int fd, char *p, int n)
{
    struct file *f;
    if(fd < 0 )
        return -1;
    if(fd >= NOFILE)
        return -1;
    if((f=myproc()->ofile[fd]) == 0)
        return -1;
    return fwrite(f, p, n);
}

```

```

int close_process(int fd)
{
    struct file *f;
    if(fd < 0 )
        return -1;
    if(fd >= NOFILE)
        return -1;
    if((f=myproc()->ofile[fd]) == 0)
        return -1;
    myproc()->ofile[fd] = 0;
    fclose(f);
    return 0;
}

```



Now when the contents of the page table has been stored in a file, we memset that page table entry for further use and we set its bit to 1 so that it can be added to free page queue.

```
void swap_out(){
    acquire(&readyqueue.lock);
    while(readyqueue.start!=readyqueue.end){
        struct proc *p=kpop();
        pde_t* pd = p->pgdir;
        for(int i=0;i<NPENTRIES;i++){
            //skip page table if accessed. chances are high, not every
            //page table was accessed.
            if(pd[i]&PTE_A)
                continue;
            pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
            int j=0;
            while(j<NPTENTRIES){
                //Skip if found
                if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P)){
                    j++;
                    continue;
                }
                pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));
                //for file name
                int pid=p->pid;
                int virtual = ((1<<22)*i)+((1<<12)*j);
                //file name
                char carray[50];
                to_string(pid,carray);
                int x=strlen(carray);
                carray[x]='_';
                to_string(virtual,carray+x+1);
                safestrcpy(carray+strlen(carray),".swp",5);
                // file management
                int fd=open_process(carray, O_CREATE | O_RDWR);
                if(fd<0){
                    fprintf("error creating or opening file: %s\n", carray);
                    panic("swap_out_process");
                }
                if(write_process(fd,(char *)pte, PGSIZE) != PGSIZE){
                    fprintf("error writing to file: %s\n", carray);
                    panic("swap_out_process");
                }
                close_process(fd);
                kfree((char*)pte);
                memset(&pgtab[j],0,sizeof(pgtab[j]));
                //mark this page as being swapped out.
                pgtab[j]=((pgtab[j])^(0x080));
                break;
            }
        }
        release(&readyqueue.lock);
        struct proc *p;
        if((p=myproc())==0)
            panic("swap out process");
        swap_out_present=0;
        p->parent = 0;
        p->name[0] = '*';
        p->killed = 0;
        p->state = UNUSED;
        sched();
    }
}
```

## Suspending kernel process when no requests are left:

When we put a kernel process to sleep because it's request queue is empty, i.e there are no more swap-in/swap-out requests left to cater to, we follow the below 2 steps.

- 1) In the respective swapping functions, upon detecting an empty queue, swap process existence variable is set to 0, it's name changed to "\*", and the state set to "UNUSED".

```

swap_in_present=0;
p->parent = 0;
p->name[0] = '*';
p->killed = 0;
p->state = UNUSED;
sched();

```

- 2) In the scheduler, upon encountering a process in UNUSED state and with "\*" as it's name, the scheduler frees the kstack of the process. Along with it, name and pid of the process is also set to 0.

```

if(p->state==UNUSED ){
    if(p->name[0]=='*'){
        cprintf("A kernel process was killed. PID: %d\n", p->pid);
        kfree(p->kstack);
        p->kstack=0;
        p->name[0]=0;
        p->pid=0;
    }
}

```

### Task 3 : Swapping in mechanism

Just like for the swap out we have maintained a separate queue for the swap-in requests also. Most of the functions are used are the same as before. We also declare an extern prototype for readyqueue2 in defs.h. All the code snippets can be found above.

The swap\_in function come into picture when there is a page fault. In Part A, we find the virtual address at which the page fault occurred by using rcr2(). We then put the current process to sleep with a new lock called swap\_in\_lock (initialised in trap.c and with extern in defs.h). We then obtain the page table entry corresponding to this address (the logic is identical to walkpgdir). Now if the corresponding page was swapped out we can simply initiate the swap in process. But if the case is otherwise we can simply exit the function then and there.

Implementation of swap in :



In `swap_in()`, it sleeps on the channel if there is nothing to swap in. The swap function will wake up when a page needs to be swapped in. When it wakes, it finds the needed file on the disk, finds a free page in the table, writes the stream of bytes from the file to the page and then deletes the file from the disk with the help of the functions we have implemented. We had modified the code in `trap.c` such that when a page fault occurs it initiates a swap in. Here is the code snippet for it:

```
void handled(uint addr) {
    uint a = PGROUNDDOWN(addr);
    struct proc *p=myproc();
    pde_t *pde = &(p->pgdir)[PDX(a)];
    pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    if((pgtab[PTX(a)])&0x080){
        //This means that the page was swapped out.
        //virtual address for page
        p->addr = a;
        kpush2(p);
        create_kernel_process("swap_in_process", &swap_in);
    } else {
        exit();
    }
}
```

## Task 4 : Sanity Test

Now we will create a test file named `memtest.c` to check all the functionalities that we have implemented till now. This test file has the following properties:

- The main process forks 20 child processes.
- Each child process executes a loop with 10 iterations.
- At each iteration the process will allocate 4KB of memory using `malloc` system call.
- Next it will fill the memory with values obtained from the function `('a'+k%26)` and finally validates the content of the memory using the same function.
- A counter named `cnt` is maintained which stores the number of bytes that contain the right values.

Necessary additions to the makefile were made as well for proper execution.

A user program is implemented whose code snippet can be found below:

```

for(int i=0;i<20;i++){
    if(!fork()){
        printf(1,"CHILD %d\n",i+1 );
        for(int j=0;j<10;j++){
            char *arr = malloc(4096);
            for(int k=0;k<4096;k++){
                arr[k] = 'a'+k%26;
            }
            int cnt=0;
            for(int k=0;k<4096;k++){
                if(arr[k] == 'a'+k%26)
                    cnt++;
            }
        }
    }
}

```

Following is the output for the above file:

```

niharika@niharika-G3-3579: ~/Desktop/SEM 5/OS LAB/LAB 3/LAB3_...
CHILD 19
Number of bytes matched= 4096 in iteration 1
Number of bytes matched= 4096 in iteration 2
Number of bytes matched= 4096 in iteration 3
Number of bytes matched= 4096 in iteration 4
Number of bytes matched= 4096 in iteration 5
Number of bytes matched= 4096 in iteration 6
Number of bytes matched= 4096 in iteration 7
Number of bytes matched= 4096 in iteration 8
Number of bytes matched= 4096 in iteration 9
Number of bytes matched= 4096 in iteration 10
CHILD 20
Number of bytes matched= 4096 in iteration 1
Number of bytes matched= 4096 in iteration 2
Number of bytes matched= 4096 in iteration 3
Number of bytes matched= 4096 in iteration 4
Number of bytes matched= 4096 in iteration 5
Number of bytes matched= 4096 in iteration 6
Number of bytes matched= 4096 in iteration 7
Number of bytes matched= 4096 in iteration 8
Number of bytes matched= 4096 in iteration 9
Number of bytes matched= 4096 in iteration 10
$ 

```

In order to check the paging mechanism, we can reduce the value of PHYSTOP. This can prevent the kernel to be able to hold all processes memory in RAM.

The default value of PHYSTOP is 224 MB. The minimum of the same can be set to 4MB and then we can see that the outputs match, hence indicating a successful implementation.