

Group Number 17

Name

Samiksha Sachdeva

Kartikay Goel

Falak Chhikara

Rahul Choudhary

Roll Number

180123040

180101033

180101023

180101060

Part A: Lazy Memory Allocation

What is lazy memory allocation?

Lazy allocation of memory means not allocating memory to a process until it is actually needed. By delaying allocation of memory until you actually need it, we can decrease startup time, and even eliminate the allocation entirely if we never actually use the process. Most modern operating systems perform lazy allocation of heap memory, though **xv6 does not**. Xv6 applications ask the kernel for heap memory using the `sbrk()` system call. In the original xv6 kernel, **`sbrk()` allocates physical memory and maps it into the process's virtual address space.**

Task: We have to add support for this lazy allocation feature in xv6 by delaying the memory requested by `sbrk()` until the process actually uses it.

Step 1: Eliminate allocation from `sbrk()`:

To delete the page allocation from the `sbrk(n)` system call implementation, which is in the function `sys_sbrk()` in `sysproc.c`, we have used the given patch file in which the call to the function `growproc(n)` is commented out. The new `sbrk(n)` will just increment the process's size by `n` and return the old size. It does not allocate memory. However, it still increases the size of the process by `n` bytes to trick the process into believing that it has the memory requested.

Typing `echo hi` to the shell gives us the following output:

```
kartikay@kartikay-goel:~/Desktop/OS LAB3/xv6org1$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc
c |
```

The “pid 3 sh: trap...” message 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 that the virtual address that caused the page fault is 0x4004.

Step 2: Adding Lazy Allocation

We have modified the code of `trap()` function 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 have allocated a new memory page, added suitable page table entries, and returned from the trap, so that the process can avoid the page fault the next time it runs. **Commented out portion of `sbrk` in `sysproc.c` ----->**

```
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;
}
```

We have added the following piece of code in trap.c

Now we can begin to handle this page fault error starting from trap.c. In trap.c, there are various cases to handle trap errors using case-switch statements so we add one more for T_PGFLT since we have a page fault error and there is no pre-existing case for handling page-fault errors.

Declared mappages function(defined in vm.c) externally for lazy page allocation.

```
17 // external declaration of mappages function originally defined in vm.c
18 // for lazy page allocation
19 extern int
20 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
21
```

```
54 // Lazy page allocation added by us
55 if(tf->trapno == T_PGFLT) // if page fault occurs
56 {
57     uint va = PGROUNDDOWN(rcr2());
58     char *mem = kalloc();
59     memset(mem, 0, PGSIZE);
60     mappages(myproc()->pgdir, (char*)va, PGSIZE, V2P(mem), PTE_W|PTE_U);
61     return; // we have to return from here
62 }
63
```

```
// 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;
}
```

Important Points:-

1. tf->trapno == T_PGFLT check whether a fault is a page fault.
2. PGROUNDDOWN(rcr2()) is used to round the faulting virtual address down to the start of a page boundary.
3. We have returned in order to avoid the cprintf and the proc->killed = 1 statements.
4. We have deleted the static keyword in the declaration of mappages() in vm.c.
5. We have declared mappages() in trap.c.

Here we can see some important functions which we need to handle our page fault:-

PGROUNDUP(address) which rounds up the address to the starting of a page or we can say it gives us the page-aligned address since we can have errors at any address in-btw page.

kalloc() allocates one-page of 4096 bytes from physical memory & returns a pointer to that page.

memset() makes the whole page null (makes sure that the page allocated is empty/fresh).

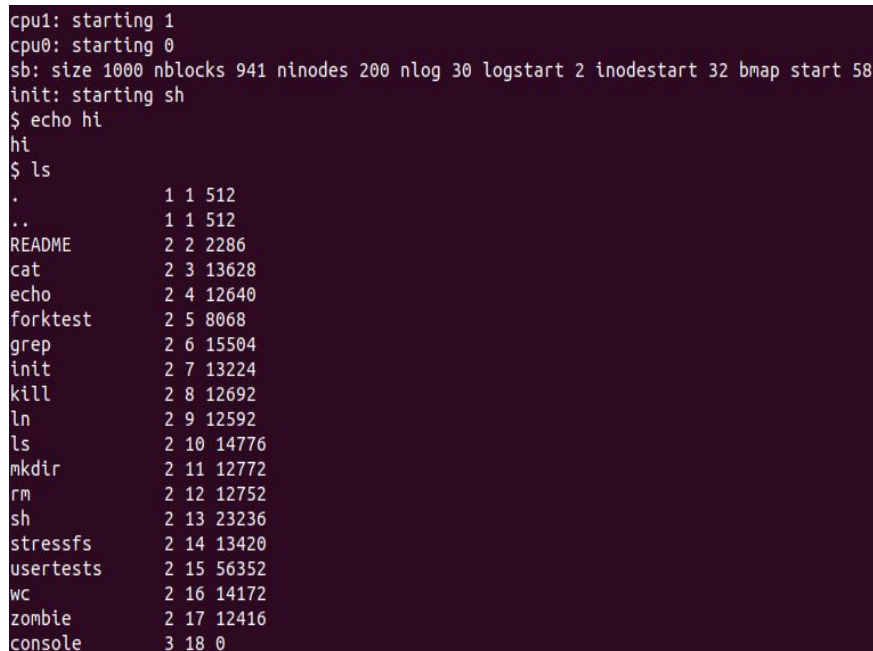
mappages() maps the page to our process's page directory by converting the given virtual address (rounded) to physical address by the use of V2P(mem) which basically subtracts the Kernel base address from the virtual address. It creates a new (page table entry) PTE for that particular virtual address.

We use these functions in our lazy allocation implementation by giving the argument myproc() which contains the current process's directory and rcr2() which gives the address at which page fault occurs and the rest of the implementation is done in the lazy_allocation function. We need not use the old size and new size arguments since we only need to allocate a page so that our program can execute normally in userspace.

The page table flags used in mappages() function are PTE_W or PTE_U which declares that the page is user accessible and writeable as seen below.

Hence our new function takes the faulty address, rounds up and makes it page aligned, allocates a fresh page from the physical memory to the process using mappages() function and the process starts to execute normally if the allocation is done correctly.

After making the following changes, the lazy memory allocation is implemented correctly in xv6 and now we are not encountering any errors after typing echo hi OR ls to the shell as shown in the screenshot attached below:



```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
hi
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 13628
echo       2 4 12640
forktest   2 5 8068
grep       2 6 15504
init       2 7 13224
kill       2 8 12692
ln         2 9 12592
ls         2 10 14776
mkdir      2 11 12772
rm         2 12 12752
sh         2 13 23236
stressfs   2 14 13420
usertests  2 15 56352
wc         2 16 14172
zombie     2 17 12416
console    3 18 0
```

Part B: Disk Swapping of User Process Pages

Key points on Memory:

- Memory is accessed in terms of Blocks, where 1 Block = 512 Bytes
- A page consists of 8 contiguous blocks of memory, which gets allotted to a particular process and 1 Page = 4096 Bytes

Task 1: Kernel Process

In order to implement paging mechanism, we are implementing the function

void create_kernel_process(const char *name, void (*entrypoint)());

This function creates a kernel process and adds it to the processes queue.

1. Implementation can be seen as modified fork() along with some code from allocproc(), and userinit().
2. It follows the same general structure of fork() with some modification. Unlike fork(), create_kernel_process() does not copy entire registers, address space from parent process, rather it acts similar to allocproc() and userinit() by setting up data from scratch. create_kernel_process() sets up the entire trap frame the same way userinit() does.
3. At the end of create_kernel_process(), it sets p->context->eip to the entrypoint function pointer(one of the parameters in the function). The process will start running at the function entrypoint.

The below code has been implemented in proc.c

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

    // Allocate process
    if ((p = allocproc()) == 0)
        panic("Failed to allocate kernel process.");

    qn = freenode;
    freenode = freenode->next;
    if(freenode != 0)
        freenode->prev = 0;

    //setup page table
    if((p->pgdir = setupkvm()) == 0){
        kfree(p->kstack);
        p->kstack = 0;
        p->state = UNUSED;
        panic("Failed to setup pgdir for kernel process.");
    }
    //other parameters
    p->sz = PGSIZE;
    p->parent = initproc; // parent is the first process.
    memset(p->tf, 0, sizeof(*p->tf));
    p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
    p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
    p->tf->es = p->tf->ds;
    p->tf->ss = p->tf->ds;
    p->tf->eflags = FL_IF;
    p->tf->esp = PGSIZE;
    p->tf->eip = 0;
    p->tf->eax = 0;
    p->cwd = namei("/");
    safestrcpy(p->name, name, sizeof(name));
    qn->p = p;
    acquire(&ptable.lock);
    p->context->eip = (uint)entrypoint;
    p->state = RUNNABLE;
    release(&ptable.lock);
}
```

Task 2 and 3: Swapping Out and Swapping in mechanism

```
struct page
{
    uint virtualAddr;           //page's virtual address
    uint pagestate;             //state of the page
    uint offset_in_file;        // offset inside the swap file
    uint counter;               // counter to see when last used
    uint pages_array_index;     //The page's index in the page array
};
```

```
struct page_in_mem
{
    struct page pg;
    struct page_in_mem *next;
    struct page_in_mem *prev;
};
```


We declared struct page for each page with attributes virtual address, pagestate and counter to keep track of when the page was last accessed. Struct page_in_mem represents nodes of a doubly linked list, where each of the nodes represents a page that is currently in the main memory

vm.c (for demand paging)

```
/* page fault handler */
void
handle_pgfault()
{
    unsigned addr;
    struct proc *curproc = myproc__();

    asm volatile ("movl %%cr2, %0\n\t" : "=r" (addr));
    addr &= ~0xfff;

    map_address(curproc->pgdir, addr, curproc->pid);
}
```

handle_pgfault()

Whenever a page fault occurs, mapping is done again. Hence, map_address function is called with the modified address.

getswappedblk()

This function returns the block id of the block corresponding to the input virtual address (in case it was swapped)

The first 20 bytes contain the block id.

```
// return the disk block-id, if the virtual address
// was swapped, -1 otherwise.
int
getswappedblk(pde_t *pgdir, uint va)
{
    pte_t *pte = walkpgdir(pgdir, (char*)va, 0);
    int block_id = (*pte) >> 12;
    return block_id;
}
```

```
pte_t* select_a_victim(pde_t *pgdir)
{
    pte_t *pte;
    for(long i=4096; i<KERNBASE; i+=PGSIZE)
    {
        //for all pages in the user virtual space
        if((pte=walkpgdir(pgdir, (char*)i, 0)) != 0) //if mapping exists
        {
            if(*pte & PTE_P)
            {
                if(*pte & ~PTE_A) //access bit not set.
                {
                    return pte;
                }
            }
        }
        else
        {
            cprintf("walkpgdir failed\n");
        }
    }
    return 0;
}
```

select_a_victim()

This function searches all the pages in the user virtual address space sequentially, until a victim is found.

The **Walkpgdir** function returns a non-zero value if there exists a mapping for that page in the PTE. If so, it further checks if that page is non-dirty and also not recently used (i.e. access bit is not set). If all these conditions are satisfied, it is selected as the victim.

If not, the loop continues. Return 0 if no victim is found.

clearaccessbit()

In this function, we clear the access bit of 10% of the pages (103 out of 1024) when selecting a victim fails in the first attempt.

```
// Clear access bit of a random pte.
void clearaccessbit(pde_t *pgdir)
{
    pte_t *pte;
    int cnt=0;
    for(long i=4096; i<KERNBASE; i+=PGSIZE)
    {
        if((pte=walkpgdir(pgdir, (char*)i, 0)) != 0)
        {
            if((*pte & PTE_P) & (*pte & PTE_A))
            {
                *pte &= ~PTE_A;
                cnt=cnt+1;
            }
        }
        if(cnt==103)
        {
            return;
        }
    }
}
```

```

uint
balloc_page(uint dev)
{
    uint alloc_b[NUM];
    int indexNCB=-1;    //pointer for above array, keeps track till where it is filled
    for(int i=0;i<8;i++)
    {
        indexNCB++;
        alloc_b[indexNCB] = balloc(dev);
        if(i>0)
        {
            if((alloc_b[indexNCB]-alloc_b[indexNCB-1])!=1) //this allocated block is non consecutive
            {
                i=0;    //start allocating blocks again
            }
        }
    }
    for(int i=0;i<=indexNCB-8;i++)
    {
        bfree(ROOTDEV,alloc_b[i]);    //free unnecessarily allocated blocks
    }
    numallocblocks++;
    return alloc_b[indexNCB-7]; //return last 8 blocks (address of 1st block among them)
}

// Free disk blocks allocated using balloc_page.
void
bfree_page(int dev, uint b)
{
    for(uint i=0;i<8;i++){
        bfree(ROOTDEV,b+i);
    }
    numallocblocks-=1;
}

```

balloc_page()

balloc_page works similar to **balloc**. It **allocates eight consecutive free blocks** i.e. 1 free page.

balloc_page works by finding 8 contiguous blocks from an array of blocks to allocate a page to the calling process.

We also keep a track of the number of blocks and pages allocated in total.

bfree_page frees the disk blocks which were allocated by **balloc_page**.

bio.c (for read and write of pages from disk)

read_page_from_disk()

This function sequentially reads the blocks for the page, first it checks buffercache (cache in file system), if not found then **bread()** allocates a buffer with enough pages and reads the specified disk block into it. No need of applying locks in read, as read can be concurrently done.

write_page_to_disk()

This function writes a page to disk, by performing write on 8 blocks in the page sequentially, starting from **start_block**.

We apply locks (**begin_op** and **end_op**) to ensure atomicity for writing a block to disk.

Bget returns a locked buffer. **Brelse** releases the buffer.

```

// Read 4096 bytes from the eight consecutive starting at blk into pg.
void
read_page_from_disk(uint dev, char *pg, uint blk)
{
    struct buf* buffer;
    int blockno=0;
    int ithpart=0;    //ith part of the current page
    for(int i=0;i<8;i++)
    {
        ithpart=i*512;
        blockno=blk+i;
        buffer=bread(ROOTDEV,blockno);    //if present in buffer, returns from buffer else from disk
        memmove(pg+ithpart, buffer->data,512);    //write to pg from buffer
        brelse(buffer);    //release lock
    }
}

//Write 4096 bytes pg to the eight consecutive starting at blk.
void
write_page_to_disk(uint dev, char *pg, uint blk,int pid,pte_t *pte)
{
    struct file* towrite=createSwapFile(pg,pid,pte);

    struct buf* buffer;
    int blockno=0;
    int ithPart=0;    //ith part of the current page
    for(int i=0;i<8;i++)
    {
        ithPart=i*512;
        blockno=blk+i;
        buffer=bget(ROOTDEV,blockno);
        towrite->off=i*512;
        memmove(buffer->data,pg+ithPart,512);    // Writing 512 bytes to the block
        filewrite(towrite,(char *)buffer,512);
        bwrite(buffer);
        brelse(buffer);    //releasing the lock
    }
}

```

```
void
swap_page_from_pte(pte_t *pte,int pid)
{

    uint physicalAddress=PTE_ADDR(*pte);    //PTE_ADDR returns address in pte
    if(physicalAddress==0)
        printf("physicalAddress address is zero\n");
    uint diskPage=ballo_page(ROOTDEV);

    write_page_to_disk(ROOTDEV,(char*)P2V(physicalAddress),diskPage,pid,pte);    //write this page to disk

    /*
     * Store block id and swapped flag in the pte entry whose page was swapped to the disk
     * So, when next time this pte is dereferenced, we know that the page has been swapped to
     * the disk and we can bring this page again to memory
     */
    *pte = (*pte & 0x000000);    //make pte = null;
    *pte = (diskPage << 12) | PTE_SWAPPED;
    *pte = *pte & ~PTE_P;

    kfree(P2V(physicalAddress));
    printf("\nReturning from swap page from pte\n");
}
}
```

swap_page()

As we can see, Swap_page selects a victim via Least Replacement Policy. It continues until it gets a victim. By calling swap_page_from_pte, it swaps the data to disk. Furthermore, old TLB entries are flushed by Lcr3.

swap_page_from_pte()

This function is responsible for swapping out a page from main memory. Ballo_page allocates 8 blocks. Write_page_to_disk writes the content of the page to be swapped in the allocated space on disk. Next the block id and swapped flag of the swapped page is stored in the pte entry so that the next time we try to access this page, we will know that it had been swapped out.

```
/* Select a victim and swap the contents to the disk.
 */
int
swap_page(pde_t *pgdir,int pid)
{
    pte_t *pte=select_a_victim(pgdir);    //returns *pte
    if(pte==0){
        //If this is true, victim is not found in 1st attempt. Inside this function
        printf("No victim found in 1st attempt. Clearing access bits.");
        clearaccessbit(pgdir);    //Accessbits are cleared,

        printf("Finding victim again, after clearing access bits of 10%% pages.");
        pte=select_a_victim(pgdir);    //then victim is selected again. Victim is found this time.

        if(pte!=0) printf("victim found");
        else printf("Not found even in second attempt." );
    }
    else{
        //This else is true, then victim is found in first attempt.
        printf("Victim found in 1st attempt.");
    }

    swap_page_from_pte(pte,pid);    //swap victim page to disk
    lcr3(V2P(pgdir));    //This operation ensures that the older TLB entries are flushed
    return 1;
}
```

```
void
map_address(pde_t *pgdir, uint addr,int pid)
{
    struct proc *curproc = myproc__();
    uint cursz= curproc->sz;
    uint a= PGROUNDDOWN(rcr2());    //rounds the address to a multiple of page size (PGSIZE)

    pte_t *pte=walkpgdir(pgdir, (char*)a, 0);
    int blockid=-1;    //disk id where the page was swapped

    char *mem=kalloc();    //allocate a physical page

    if(mem==0){
        swap_page(pgdir,pid);
        mem=kalloc();    //now a physical page has been swapped to disk and free, so this
        printf("kalloc success\n");
    }

    if(pte!=0){
        if(*pte & PTE_SWAPPED){
            blockid=getswappedblk(pgdir,a);    //disk id where the page was swapped
            read_page_from_disk(ROOTDEV, mem, blockid);

            *pte=V2P(mem) | PTE_W | PTE_U | PTE_P;
            *pte &= ~PTE_SWAPPED;
            lcr3(V2P(pgdir));
            bfree_page(ROOTDEV,blockid);
        }
        else{
            memset(mem,0,PGSIZE);
            if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_P | PTE_W | PTE_U )<0)
            {
                panic("allocuvm out of memory xv7 in mappages/n");
                deallocuvmXV7(pgdir,cursz+PGSIZE, cursz);
                kfree(mem);
            }
        }
    }
}
}
```

map_address()

PGROUNDDOWN function is used to round up the address to a multiple of page size (PGSIZE). kalloc() will try to allocate a physical page, if failed then swap_page is called, because of which a new free page would be generated for kalloc. The physical page is mapped to virtual page (address) by first reading the page, then calling V2P i.e. Virtual to Physical on its address. Also the access bit of the page is set, it is common to both virtual and physical pages since last 12 bits are the same in both

Summary points:-

In **trap.c**, we handle the actual working of page swapping so that processes whose virtual memory size is greater than physical memory size can also execute. When a page is required by a process which is currently not in the main memory, we get a page fault, Let FLT. So as soon we get the page fault we look if it is due to a user process and note down the virtual address. We then check if the page resides in the and call for the swap Page function to execute the swapping, so that we get a victim page from the physical memory and then swap it out from the main memory and bring the required page to the main memory.

In the file memlayout.h we lowered the value of the variable **PHYSTOP** to enable a successful execution of the test cases.

In **proc.c**, we have implemented a function that updates the access time of the particular page to global counter variable value. We walk through the processes and for every process we traverse the list of pages currently in the main memory and check if the page was accessed. If it is, then we update the counter attribute else skip to the next page.

Whenever we need to replace a page from the table, we choose the victim through the Least Recently Used Policy. The access time for a page will be updated every time it is accessed. Thus the page with the min access time will be the Least Recently Used. Thus, we store that in the pageToRemove variable and that is replaced from the page table.

The function **createSwapFile** for each process we create a swap file for the pages being swapped out. The naming convention given in the question has been followed. The variable in of type struct Inode is used to store the file created. For each process swapFile variable stores the type (**FDINODE**) and the permissions of the file.

Task 4: Sanity Test

Test case description:

As specified in the assignment, we forked 20 children. If a child is running, then we call a function `child_mem(int child)`. In each function, we malloc 4kB memory 10 times. We treat the memory allocated as an integer array and assign each block in the array with the value of child. We further check whether data has been correctly saved. If the data is not correctly saved or the memory is not allocated, we print "Test case failed" and exit the function. When we reach the end of the function since everything has worked correctly, we print "__th child: Test case passed".

```
$ memtest
0th child: Test case passed
1th child: Test case passed
2th child: Test case passed
3th child: Test case passed
4th child: Test case passed
5th child: Test case passed
6th child: Test case passed
7th child: Test case passed
8th child: Test case passed
9th child: Test case passed
10th child: Test case passed
11th child: Test case passed
12th child: Test case passed
13th child: Test case passed
14th child: Test case passed
15th child: Test case passed
16th child: Test case passed
17th child: Test case passed
18th child: Test case passed
19th child: Test case passed
```

Output:

As we have described, when everything runs fine for a single fork, we print the statement Test case passed. In the screenshot above, the lines have been printed for each fork child. Also, it signifies that swapping is done correctly.

