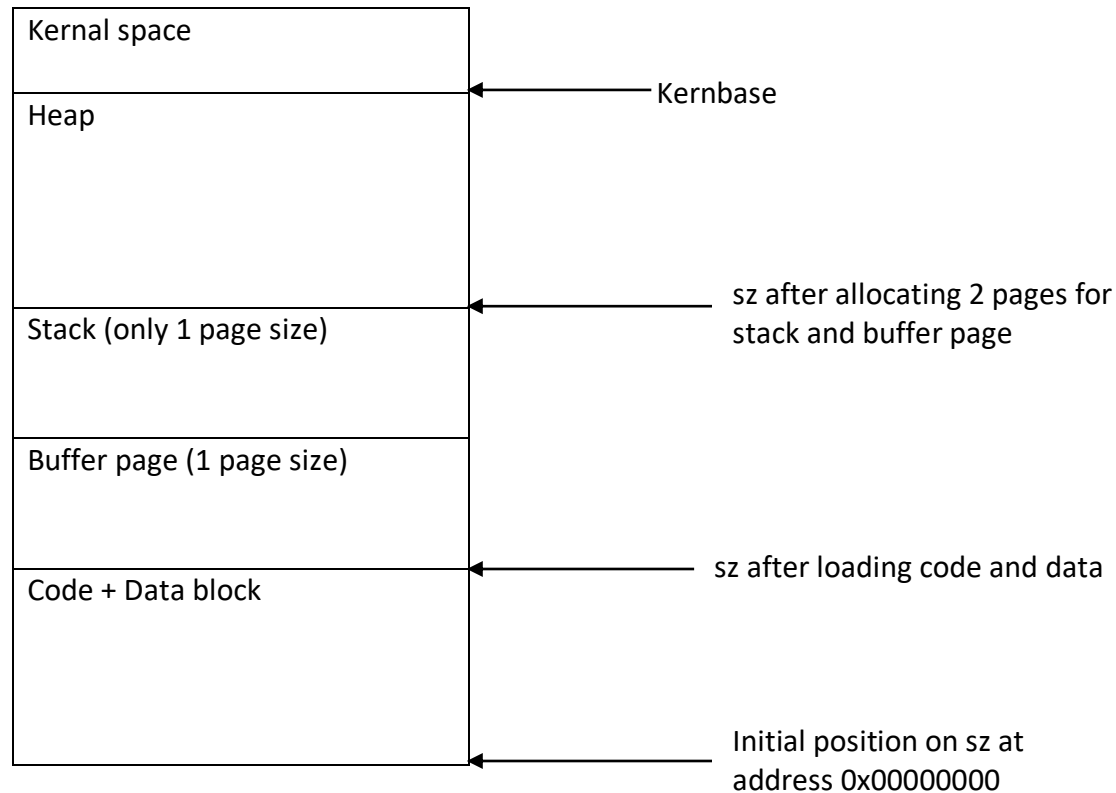


Name: ANUJA PATIL
Student ID: 862123752

1) Changing memory layout

Approach: The original xv6's memory layout is:

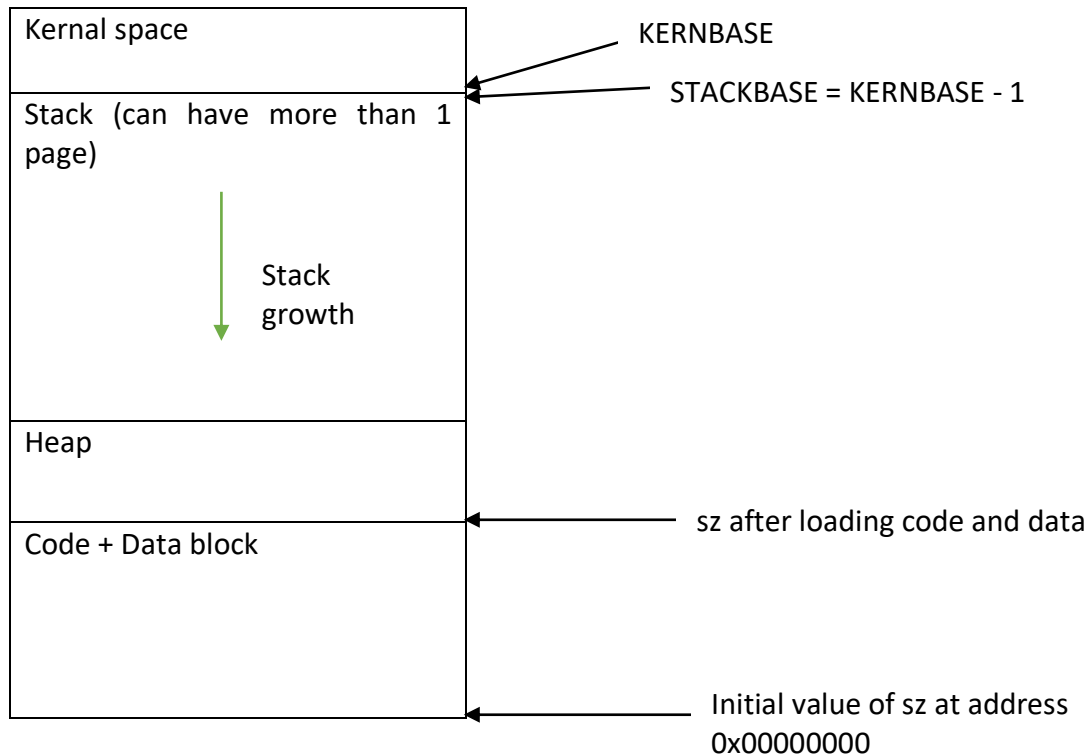


The stack pointer `sp` is initially same as that of `sz` after allocating 2 pages for stack and buffer page. It changes when program finishes executing instruction at `sp` – it basically decrements because stack is growing downwards. We have a buffer page below 1 page of stack to avoid corrupting data in Code+Data block. We clear the PTE in buffer page because we want it to be inaccessible by the stack. When stack accesses the buffer after using its allocated space, it causes a trap to the OS because the PTE of buffer is cleared which means that the page in buffer is unmapped.

For lab 3, we want to move the stack where the heap currently is i.e just below the kernel space. We also want to allocate more pages to the stack as and when required. We do not need buffer page in this implementation (at least for tests in lab 3). `sz` will remain at the point in the above diagram where code and data are loaded. `sp` will point to the base of newly shifted stack.

`PGROUNDUP(x)` is a macro to round the address `x` to a multiple of page size just higher than `x`.
`PGROUNDDOWN(x)` is a macro to round the address `x` to a multiple of page size just lower than `x`.

The new implementation of stack should look like:



Following are the files changed and a brief summary of the changes done in each:

a) proc.h

Added a field 'stackPages' in the proc struct which stores the number of pages allocated to a process stack:

```
struct proc {  
    .  
    .  
    uint stackPages;  
}
```

b) memlayout.h

Defined a variable "STACKBASE" which points to one address lower than the KERNBASE. This will be the base of our newly shifted stack:

```
#define STACKBASE (KERNBASE - 1)
```

c) exec.c

After we round up sz once code and data are loaded, we allocate 1 page size memory for the new stack starting from STACKBASE - PGSIZE to STACKBASE. Then we assign the STACKBASE to the stack pointer sp:

```

if(allocuvm(pgdir, STACKBASE - PGSIZE, STACKBASE) == 0)
    goto bad;
sp = STACKBASE;

```

Further in the code where we “Commit to the user image”, we also define the current process’ stackPages to 1 and print its value:

```

curproc->stackPages = 1;
cprintf("Initial number of pages by the process: %d\n", curproc->stackPages);

```

d) vm.c

I modified the copyuvm() which is called when a process is forked. The child process is created by calling copyuvm() that copies the code and data block, and parent process’ stack. For the original implementation of xv6, the first ‘for’ loop copies the code and data block, and the parent process’ stack. For the new stack, I added another ‘for’ loop that copies the parent process’ stack because now, the first ‘for’ loop copies only the code and data block. Our new ‘for’ loop starts from STACKBASE and goes until top of stack. Top of stack is calculated by subtracting “number of stack pages multiplied by page size” from the STACKBASE. We are decrementing the value of loop counter by page size because we copy memory in terms of 1 page size. The loop body is same as that of ‘for’ loop that copies code and data block.

```

uint stackTop = STACKBASE - (myproc()->stackPages * PGSIZE);
for(i = STACKBASE; i > stackTop; i -= PGSIZE) {
    if((pte = walkpgdir(pgdir, (void*) i, 0)) == 0)
        panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
        panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
        goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*) PGROUNDDOWN(i), PGSIZE, V2P(mem), flags) < 0) {
        kfree(mem);
        goto bad;
    }
}

```

2) Growing the stack

Approach: I added a case in trap to handle page faults. For now, our trap handler simply checks if the page fault was caused by an access to the page right under the current top of the stack. If this is the case, we allocate and map the page, and we are done. If the page fault is caused by a different address, we go to the default handler and do a kernel panic like in original xv6. We get

the address of page fault (offending address) by calling `rcr2()`. `rcr2()` returns the value stored in CR2 (Control Register) register. For this, I have modified the file trap.c:

```
case T_PGFLT:
;
uint offendingAddr = PGROUNDDOWN(rcr2());
uint stackTop = STACKBASE - (myproc()->stackPages * PGSIZE);
if(offendingAddr <= stackTop && offendingAddr >= (stackTop - PGSIZE)) {
    if(allocvm(myproc()->pgdir, offendingAddr, stackTop) == 0) {
        cprintf("case T_PGFLT from trap.c: allocvm failed. Number of current allocated pages:
%d\n", myproc()->stackPages);
        exit();
    }
    myproc()->stackPages += 1;
    cprintf("case T_PGFLT from trap.c: allocvm succeeded. Number of pages allocated:
%d\n", myproc()->stackPages);
    break;
}
```

//PAGEBREAK: 13

3) Modified `syscall.c` to take care of some system calls – `fetchint()`, `fetchstr()`, and `argptr()`.

```
int fetchint(uint addr, int *ip) {
    //Commenting following curproc for lab3 as it is not needed.
    //struct proc *curproc = myproc();

    //Commenting following 'if' and adding another 'if' for lab3.
    //if(addr >= curproc->sz || addr+4 > curproc->sz)
    if(addr > STACKBASE || addr+4 > STACKBASE)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

```
int fetchstr(uint addr, char **pp) {
    char *s, *ep;
    //Commenting following curproc for lab3 as it is not needed.
    //struct proc *curproc = myproc();

    //Commenting following 'if' for lab3 and added another 'if' containing
    //STACKBASE
    //if(addr >= curproc->sz)
    if(addr > STACKBASE)
        return -1;
```

```

*pp = (char*)addr;
//Commenting following ep assignment and adding another for lab3.
//ep = (char*)curproc->sz;
ep = (char*) STACKBASE;

//Commenting following 'for' for lab3 and adding another 'for'.
//for(s = *pp; s < ep; s++){
for(s = *pp; s <= ep; s++) {
    if(*s == 0)
        return s - *pp;
}
return -1;
}

int argptr(int n, char **pp, int size) {
    int i;
    //Commenting following curproc for lab3 as it is not needed.
    //struct proc *curproc = myproc();

    if(argint(n, &i) < 0)
        return -1;
    //Commenting following 'if' for lab3 and added another 'if' containing
    //STACKBASE
    //if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    if(size < 0 || (uint)i > STACKBASE || (uint)i+size > STACKBASE)
        return -1;
    *pp = (char*)i;
    return 0;
}

```

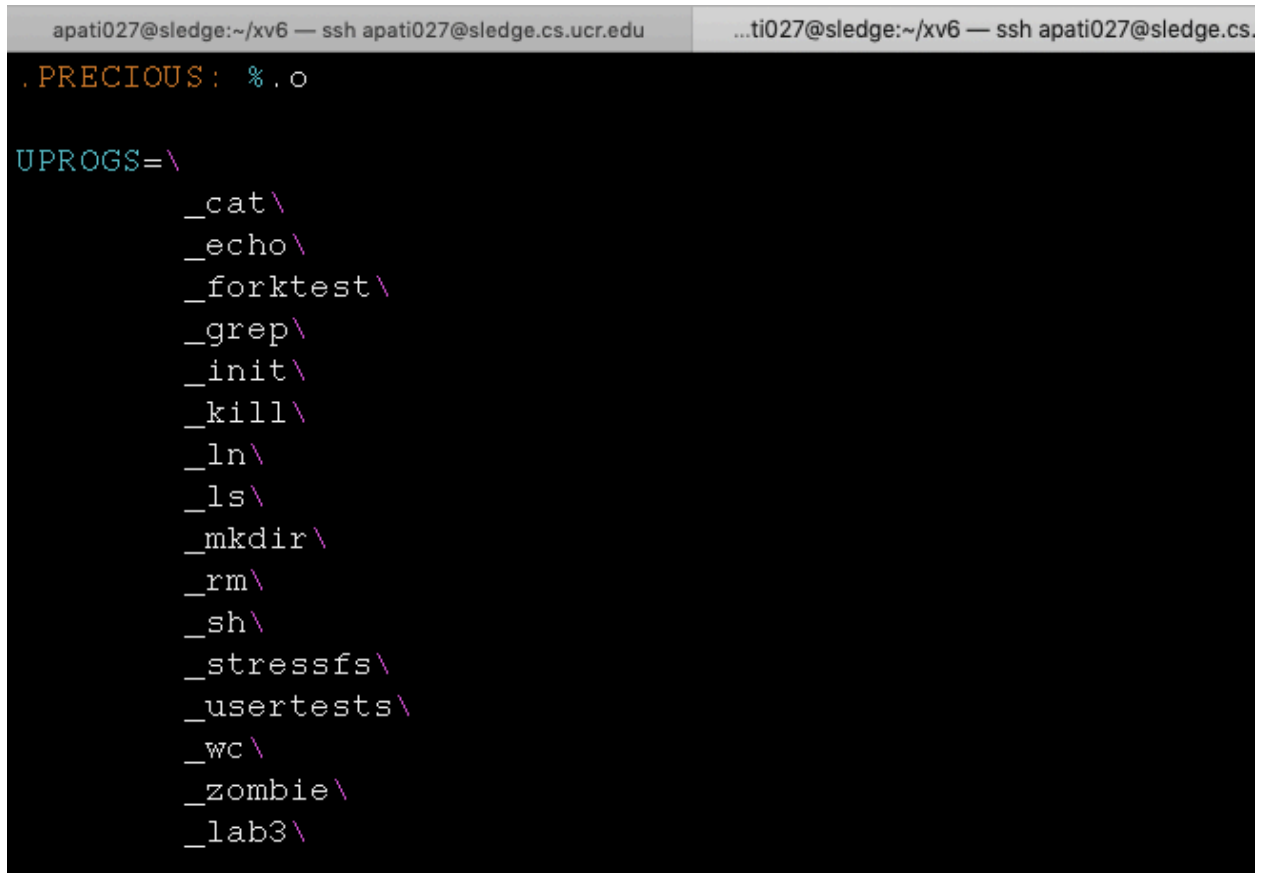
4) Modified proc.c for copying stackPages field from parent process to child process in fork(). This change is not needed for the test cases for lab 3. Hence, it is optional to modify proc.c.

```

int fork(void) {
    .
    .
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;
    np->stackPages = curproc->stackPages;
    .
    .
}

```

5) Modified Makefile for including lab3 under UPROGS.

A terminal window with a dark background and light-colored text. The terminal shows a Makefile modification. At the top, there are two tabs: 'apati027@sledge:~/xv6 — ssh apati027@sledge.cs.ucr.edu' and '...ti027@sledge:~/xv6 — ssh apati027@sledge.cs.'. The main content of the terminal is a Makefile snippet. It starts with a line '.PRECIOUS: %.o' in orange. Below it, 'UPROGS=' is in green, followed by a list of programs in backslashes: '_cat\'', '_echo\'', '_forktest\'', '_grep\'', '_init\'', '_kill\'', '_ln\'', '_ls\'', '_mkdir\'', '_rm\'', '_sh\'', '_stressfs\'', '_usertests\'', '_wc\'', '_zombie\'', and finally '_lab3\''.

```
apati027@sledge:~/xv6 — ssh apati027@sledge.cs.ucr.edu ...ti027@sledge:~/xv6 — ssh apati027@sledge.cs.
.PRECIOUS: %.o

UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _lab3\
```

6) To make and run the lab3.c: (We are inside xv6 folder)

> make clean

> make qemu-nox

This will open qemu console. Then run:

\$ lab3 100

We will get sum of first 100 numbers as the output. Then run:

\$ lab3 1000

We can see page faults being handled. Initially we had only 1 page allocated for our stack. Since the argument 1000 causes the recursion stack to grow deep, we get page faults and, in the output, we can see whenever a page fault occurs, a new page memory is allocated (incremented number of stack pages). Finally, we can see the output - sum of first 1000 numbers.

Following is the screenshot of output:

```
apati027@sledge:~/xv6 — ssh apati027@sledge.cs.ucr.edu ...apati027@sledge:~ — ssh apati027@sledge.cs.ucr.edu ...r2020/CS153-NaelAbu-Ghazaleh/Labs/Lab3

SeaBIOS (version 1.11.0-2.el7)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF94780+1FED4780 C980

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Initial number of pages by the process: 1
init: starting sh
Initial number of pages by the process: 1
|$ lab3 100
Initial number of pages by the process: 1
Lab 3: Recursing 100 levels
Lab 3: Yielded a value of 5050
|$ lab3 1000
Initial number of pages by the process: 1
Lab 3: Recursing 1000 levels
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 2
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 3
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 4
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 5
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 6
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 7
case T_PGFLT from trap.c: allocuvvm succeeded. Number of pages allocated: 8
Lab 3: Yielded a value of 500500
$
```

Summary for lab3:

The lab3 was about getting familiar with the memory layout of xv6 and modifying its current stack implementation. We also learnt how page faults can be handled.