

ASSIGNMENT 3

Group 33:

Arpan Khanna 200101121

Mukul Garg 200101068

Kartikey Gupta 200101055

Part A

Lazy Memory allocation

A brief description:

In this part of the lab, we have implemented the **Lazy Memory Allocation** for xv6, which is a feature in most modern operating systems. In case of the original xv6, it makes use of the **sbrk()** system call, to allocate physical memory and map it to the virtual address space. In the first section, we modified the **sbrk()** system call to remove

the memory allocation, and cause a page fault. In the second section we have modified

the trap.c file to resolve this page fault via lazy allocation.

1. Eliminate allocation from sbrk()

In this section, we have modified the **sbrk()** system call (also provided to us in the patch file). After initial declarations and error handling, the **sbrk()** system call has 4 essential lines in it.

```
53  addr = myproc()->sz;
54  myproc()->sz += n;
55
56      if(growproc(n) < 0)
57          return -1;
58  return addr;
59 }
```

Line 1: Assigns **addr** to the start of the newly allocated region

Line 2: Increases the size for the current process by a factor **n**

Line 3: Calls the **growproc(n)** function in **proc.c**, which allocates **n** bytes of memory for the process.

Line 4: Returns the **addr**

Now we comment-out the line 3, and this makes the process to believe that it has got it's requested memory, while in reality it does not. This will cause a **trap error, with the code 14** when we try to run something like **echo hi** or **ls**. The code 14 corresponds to the **page fault error**, or **T_PGFLT**.

```

45 int
46 sys_sbrk(void)
47 {
48     int addr;
49     int n;
50
51     if(argint(0, &n) < 0)
52         return -1;
53     addr = myproc()->sz;
54     myproc()->sz += n;
55
56     if(growproc(n) < 0)
57         return -1;
58     return addr;
59 }

```

The modified code for system call **sbrk()** is as below:

On running `echo hi` in the terminal, we get the following output.

```

cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc

```

2. Lazy Allocation in xv6

In this section, we handle the page fault resulting from the changes in part A.1. For this we make use of the following observations :

- The file `trap.c` has the code that produces the trap error as observed in part A.1.

This is present in the default case for the `switch(tf->trapno)` as follows:

```

111 // In user space, assume process misbehaved.
112 cprintf("pid %d %s: trap %d err %d on cpu %d "
113         "eip 0x%x addr 0x%x--kill proc\n",
114         myproc()->pid, myproc()->name, tf->trapno,
115         tf->err, cpuid(), tf->eip, rcr2());
116 myproc()->killed = 1;
117 }

```

- Comparing with the above output we realise that `rcr2()` represents the contents of the control register 2 which in turn has the faulting virtual address. This is what goes into the input of `PGROUNDDOWN(va)` later.
- Inside the `T_PGFLT` case, we make use of `PGROUNDDOWN(va)` to round down the virtual address to the start of the page boundary.
- In `vm.c`, we have the function `allocvm()` which is what `sbrk()` makes use of via the `growproc()` function.
- Studying `allocvm()`, makes it clear that it assigns 4KB (`PGSIZE`) of pages to a function making use of `kalloc()`, in a loop for as many pages as are needed. In our case we need a similar thing, except that we can do away with the loop and assign 1 page of size 4KB (`PGSIZE`) as and when a page fault occurs.
- A final observation is to remove the static keyword for the `mappages` function in `vm.c` and declare it as `extern` in `trap.c`. This will make sure that we can call it inside

the switch case. We also add a break; statement to make sure that fall-through does not occur and the default statements are not executed.

The code changed in various files are as below:

In trap.c

```
17 extern int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
82     case T_PGFLT:
83     {
84         // code from allocvm
85         //cprintf("Trap Number : %x\n", tf->trapno);
86         cprintf("rcr2() : 0x%x\n", rcr2());
87
88         uint newsz = myproc()->sz;
89         uint a = PGROUNDDOWN(rcr2());
90         if(a < newsz){
91             char *mem = kalloc();
92             if(mem == 0) {
93                 cprintf("out of memory\n");
94                 exit();
95                 break;
96             }
97             memset(mem, 0, PGSIZE);
98             mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U);
99         }
100         break;
101     }
```

In vm.c

```
65 int
66 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

The output on running commands like echo and ls is shown below. As can be seen, the trap error does not occur any more. Additionally, we have used cprintf in our code to print the faulting virtual address in each case, which shows up in the terminal output.

```

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
rcr2() : 0x4004
rcr2() : 0xbfa4
hi
$ ls
rcr2() : 0x4004
rcr2() : 0xbfa4
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 13620
echo       2 4 12632
forktest   2 5 8060
grep       2 6 15496
init       2 7 13216
kill       2 8 12684
ln         2 9 12580
ls         2 10 14768
mkdir     2 11 12764
rm         2 12 12740
sh         2 13 23228
stressfs   2 14 13412
usertests  2 15 56344
wc         2 16 14160
zombie     2 17 12404
console    3 18 0

```

Answers to some Question

Q1 How does the kernel know which physical pages are used and unused?

Ans - kernel maintains the list of free pages in **kalloc.c** called **kmem**.

Q2 What data structures are used to answer this question?

Ans - A linked list is used as data structure for storing free pages.

Q3 Where do these reside?

Ans - Linked List is declared inside **kalloc.c** inside structure **kmem**.

Q4 Does xv6 memory mechanism limit the number of user processes?

Ans - Number of user processes are limited to 64 as defined by **NPROC** in **param.h**

Q5 If so, what is the lowest number of processes xv6 can 'have' at the same time?

Ans - Minimum number of processes in xv6 can be 1 as while starting only one process named initproc which initiates all other user processes.

Part B

Task 1

The `create_kernel_process()` function was created in `proc.c`. It took the name of the process and entrypoint function as arguments. It always remains in kernel mode. The parent process was set to `initproc` and `p->context->eip` was set to entrypoint argument and all other values as default. `allocproc` allocates the process a spot in the ptable. `setupkvm` maps the virtual address to physical address (from 0 to PHYSTOP).

```
void create_kernel_process(const char *name, void (*entrypoint)())
{
    struct proc *kp = allocproc();
    if(kp == 0){
        panic("Failed to allocate kernel process.");
    }
    kp->pgdir = setupkvm();
    if(kp->pgdir == 0)
    {
        kfree(kp->kstack);
        kp->kstack = 0;
        kp->state = UNUSED;
        panic("Failed to setup pgdir for kernel process.");
    }
    kp->sz = PGSIZE;
    kp->parent = initproc;
    memset(kp->tf, 0, sizeof(*kp->tf));
    kp->tf->cs = (SEG_KCODE << 3) | DPL_USER;
    kp->tf->ds = (SEG_KDATA << 3) | DPL_USER;
    kp->tf->es = kp->tf->ds;
    kp->tf->ss = kp->tf->ds;
    kp->tf->eflags = FL_IF;
    kp->tf->esp = PGSIZE;
    kp->tf->eip = 0;
    kp->tf->eax = 0;
    kp->cwd = namei("/");

    safestrcpy(kp->name, name, sizeof(name));

    acquire(&ptable.lock);

    kp->context->eip = (uint)entrypoint;
    kp->state = RUNNABLE;

    release(&ptable.lock);

    return;
}
```


Task 2

We create a container to contain the processes who have asked for additional memory but do not have any free pages. Therefore, we implement a circular queue and functions to give entry (cq_push) and exit (cq_pop) from the circular queue in proc.c .

```
169 struct circular_queue{
170     struct spinlock lock;
171     struct proc* queue[NPROC];
172     int head;
173     int tail;
174 };
175
176 // circular process queue for swapping out requests
177 struct circular_queue cq;
178
179 int cq_push(struct proc *p){
180     acquire(&cq.lock);
181     if ((cq.tail + 1) % NPROC == cq.head){
182         release(&cq.lock);
183         return 0;
184     }
185     cq.queue[cq.tail] = p;
186     cq.tail = (cq.tail + 1) % NPROC;
187     release(&cq.lock);
188
189     return 1;
190 }
191
192 struct proc* cq_pop(){
193     acquire(&cq.lock);
194     if(cq.head == cq.tail){
195         release(&cq.lock);
196         return 0;
197     }
198     struct proc *p = cq.queue[cq.head];
199     cq.head = (cq.head + 1) % NPROC;
200     release(&cq.lock);
201     return p;
202 }
```

We initialise the queue in userinit (user initialisation) function and lock for the queue in pinit function.

```
512 void
513 userinit(void)
514 {
515     acquire(&cq.lock);
516     cq.head = 0;
517     cq.tail = 0;
518     release(&cq.lock);
519
520     acquire(&cq1.lock);
521     cq1.head = 0;
522     cq1.tail = 0;
523     release(&cq1.lock);
524
525     struct proc *p;
```

```

370 void
371 pinit(void)
372 {
373     initlock(&ptable.lock, "ptable");
374     initlock(&cq.lock, "cq");
375     initlock(&sleeping_channel_lock, "sleeping_channel");
376     initlock(&cq1.lock, "cq1");
377 }

```

We want to use the circular queue globally therefore we give its definition in defs.h

```

12 struct circular_queue;
124 extern struct circular_queue cq;
125 int cq_push(struct proc *p);
126 struct proc* cq_pop();
127 extern struct circular_queue cq1;

```

Whenever a process needs to access some data it calls the **walkpgdir** function: If the data is not present in main memory, then **growproc** function is called which calls the **allocvm** function which ultimately calls the **kalloc** function. If any free page is available then kalloc assigns it to the process else we need to swap out a page according to LRU policy to get a free page.

```

240     if(mem == 0){
241         // cprintf("allocvm out of memory\n");
242         deallocvm(pgdir, newsz, oldsz);
243
244         // SLEEP
245         myproc()->state = SLEEPING;
246         acquire(&sleeping_channel_lock);
247         myproc()->chan=sleeping_channel;
248         sleeping_channel_count++;
249         release(&sleeping_channel_lock);
250
251         cq_push(myproc());
252         if(!swap_out_process_exists){
253             swap_out_process_exists = 1;
254             create_kernel_process("swap_out_process", &swap_out_process_function);
255         }
256         return 0;
257     }
258     memset(mem, 0, PGSIZE);
259     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
260         cprintf("allocvm out of memory (2)\n");
261         deallocvm(pgdir, newsz, oldsz);
262         kfree(mem);
263         return 0;
264     }

```

To swap out the page, we first need to move the process in the sleeping state on a special channel called **sleeping_channel**. We create this special channel in vm.c

```

14 struct spinlock sleeping_channel_lock;
15 int sleeping_channel_count = 0;
16 char *sleeping_channel;

```

Then we declare it in defs.h to use it globally.

```

201 void                clearpteu(pde_t *pgdir, char *uva);
202 extern struct spinlock sleeping_channel_lock;
203 extern int           sleeping_channel_count;
204 extern char*         sleeping_channel;

```

When we have a free page (either already had or after swap out) we need to assign it to the process, for this processes sleeping on sleeping_channel need to be woken up by **wakeup()** system call.

```
78 // wake up processes sleeping on a sleeping channel
79 if(kmem.use_lock)
80     acquire(&sleeping_channel_lock);
81 if(sleeping_channel_count){
82     wakeup(sleeping_channel);
83     sleeping_channel_count = 0;
84 }
85 if(kmem.use_lock)
86     release(&sleeping_channel_lock);
87 }
```

We now implement the swap_out_process to really swap out the page.

To determine the victim page using LRU policy, we iterate through each entry in the process page table and look at the accessed bit which is obtained by bitwise & of the entry and PTE_A. The access bit indicates whether the page was accessed in the last iteration or not.

```
97 #define PTE_PS          0x080    // Page Size
98 #define PTE_A           0x020    // Accessed
```



```

231 void swap_out_process_function(){
232     acquire(&cq.lock);
233     while (cq.head != cq.tail){
234         struct proc *p = cq_pop();
235
236         pde_t *pd = p->pgdir;
237         for(int i = 0; i < NPENTRIES; i++){
238
239             // skip page table if accessed
240             if(pd[i] & PTE_A)
241                 continue;
242             pte_t *pt = (pte_t *) P2V(PTE_ADDR(pd[i]));
243             for(int j = 0; j < NPENTRIES; j++){
244                 // skip if found
245                 if((pt[j] & PTE_A) || !(pt[j] & PTE_P))
246                     continue;
247                 pte_t *pte = (pte_t *) P2V(PTE_ADDR(pt[j]));
248
249                 // for file name
250                 int pid = p->pid;
251                 int virt = ((1 << 22) * i) + ((1 << 12) * j);
252
253                 // file name
254                 char c[50];
255                 itoa(pid, c);
256                 int x = strlen(c);
257                 c[x] = '_';
258                 itoa(virt, c + x + 1);
259                 safestrcpy(c + strlen(c), ".swp", 5);
260
261                 // file management
262                 int fd = proc_open(c, O_CREATE | O_RDWR);
263                 if (fd < 0){
264                     cprintf("Error creating or opening file: %s\n", c);
265                     panic("swap out process");
266                 }
267
268                 if(proc_write(fd, (char *) pte, PGSIZE) != PGSIZE){
269                     cprintf("Error writing to file: %s\n", c);
270                     panic("swap out process");
271                 }
272                 proc_close(fd);
273
274                 kfree((char *) pte);
275                 memset(&pt[j], 0, sizeof(pt[j]));
276
277                 // mark this page as swapped out
278                 pt[j] = pt[j] ^ 0x008;
279
280                 break;
281             }
282         }
283     }
284     release(&cq.lock);
285
286     struct proc *p;
287     if ((p = myproc()) == 0)
288         panic("swap out process");
289
290     swap_out_process_exists = 0;
291     p->parent = 0;
292     p->name[0] = '*';
293     p->killed = 0;
294     p->state = UNUSED;
295     sched();
296 }

```

In the scheduler, we unset the accessed bit.

```
750     for(int i = 0; i < NPDETRIES; i++){
751         // if PDE were accessed
752
753         if(((p->pgdir)[i]) & PTE_P && ((p->pgdir)[i]) & PTE_A){
754
755             pte_t *pt = (pte_t *) P2V(PTE_ADDR((p->pgdir)[i]));
756
757             for(int j = 0; j < NPTENTRIES; j++){
758                 if(pt[j] & PTE_A){
759                     pt[j] ^= PTE_A;
760                 }
761             }
762             ((p->pgdir)[i]) ^= PTE_A;
763         }
764     }
```

Now the swapped out page needs to be stored(written) in secondary storage for that we have copied open, write, close, functions from proc.c to sysfile.c and named as proc_open, proc_write, proc_close

```
8 #include "spinlock.h"
9 #include "fcntl.h"
10 #include "stat.h"
11 #include "sleeplock.h"
12 #include "fs.h"
13 #include "file.h"           , etc.
```

```
--
20 int
21 proc_close(int fd)
22 {
23     struct file *f;
24
25     if(fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
26         return -1;
27     myproc()->ofile[fd] = 0;
28     fileclose(f);
29     return 0;
30 }
31
32 int
33 proc_write(int fd, char *p, int n)
34 {
35     struct file *f;
36
37     if(fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
38         return -1;
39     return filewrite(f, p, n);
40 }
```

We clear the stack of the kernel process while exiting from it.

```

737     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
738
739         // if the swapped out process has stopped running, free its stack
740         if(p->state == UNUSED && p->name[0] == '*'){
741             kfree(p->kstack);
742             p->kstack = 0;
743             p->name[0] = 0;
744             p->pid = 0;
745         }

```

Task-3

When a process had requested a data which is not in main memory, i.e, in case of a page fault, we need to swap the page containing data into the main memory. First we need to create a container to satisfy the swap-in requests, just like we did in task-2. We create a circular queue and functions to enter and exit the container in proc.c.

```

177 struct circular_queue cq;
178
179 int cq_push(struct proc *p){
180     acquire(&cq.lock);
181     if ((cq.tail + 1) % NPROC == cq.head){
182         release(&cq.lock);
183         return 0;
184     }
185     cq.queue[cq.tail] = p;
186     cq.tail = (cq.tail + 1) % NPROC;
187     release(&cq.lock);
188
189     return 1;
190 }
191
192 struct proc* cq_pop(){
193     acquire(&cq.lock);
194     if(cq.head == cq.tail){
195         release(&cq.lock);
196         return 0;
197     }
198     struct proc *p = cq.queue[cq.head];
199     cq.head = (cq.head + 1) % NPROC;
200     release(&cq.lock);
201     return p;
202 }

```

We initialise the queue in userinit (user initialisation) function and lock for the queue in pinit function.

```

512 void
513 userinit(void)
514 {
515     acquire(&cq.lock);
516     cq.head = 0;
517     cq.tail = 0;
518     release(&cq.lock);
519
520     acquire(&cq1.lock);
521     cq1.head = 0;
522     cq1.tail = 0;
523     release(&cq1.lock);
370 void
371 pinit(void)
372 {
373     initlock(&ptable.lock, "ptable");
374     initlock(&cq.lock, "cq");
375     initlock(&sleeping_channel_lock, "sleeping_channel");
376     initlock(&cq1.lock, "cq1");
377 }

```

We want to use the circular queue globally therefore we give its definition in defs.h

```

127 extern struct circular_queue cq1;
128 int cq_push1(struct proc *p);
129 struct proc* cq_pop1();

```

We create an integer variable to store the virtual address where the page fault has occurred in proc.h .

```

51 char name[16];           // Process name (debugging)
52 int va;                  //Virtual Address of the process

```

Whenever a page fault occurs the process traps the os, therefore to handle the page fault we add the following in trap.c

```

97 case T_PGFLT:
98     if(PageFaultHandle()<0){
99         cprintf("Memory Not Allocated!!!");
100     }
101     break;

```

In pfhandling function, we set the process in sleeping state and obtain the virtual address where the page fault has occurred. Then we check whether the page was swapped out or not. If not then allow the default way of handling page fault else call **swap_in()** function.

Then we declare swap_in() in defs.h to use it globally.

```

130 void                swap_out_process_function();
131 void                swap_in();
132
133 extern int          swap_out_process_exists;
134 extern int          swap_in_process_exists;

```


We have already implemented other file management functions in task-2. We here implement read_process which is basically copy function.

```

299 int read_process(int fd, int n, char *p)
300 {
301     struct file *fl;
302     if(fd < 0 || fd >= NOFILE) return -1;
303     fl = myproc()->ofile[fd];
304     if(fl==0) return -1;
305     return fileread(fl, p, n);
306
307 }

```

We allocate a free page to the process in main memory and read the page from secondary storage to the allocated free page in the main memory.

```

309 void swap_in(){
310
311     acquire(&cq1.lock);
312     while(cq1.head!=cq1.tail){
313         struct proc *p=cq_pop1();
314
315         int process_id=p->pid;
316         int va=PTE_ADDR(p->va);
317
318         char pagename[50];
319         itoa(process_id,pagename);
320         int length = strlen(pagename);
321         pagename[length]='_';
322         itoa(va,pagename+length+1);
323         safestrcpy(pagename+strlen(pagename),".swp",5);
324
325         int fd=proc_open(pagename,O_RDONLY);
326         if(fd<0){
327             release(&cq1.lock);
328             cprintf("Page could not be found in memory: %s\n", pagename);
329             panic("swap in failed");
330         }
331         char *mem=kalloc();
332         read_process(fd,PGSIZE,mem);
333         int x = PTE_W|PTE_U;
334         int mp = mappages(p->pgdir, (void *)va, PGSIZE, V2P(mem),x );
335         if(mp<0){
336             release(&cq1.lock);
337             panic("page mapping");
338         }
339         wakeup(p);
340     }
341
342     release(&cq1.lock);
343     struct proc *p = myproc();
344     if(p==0)
345         panic("swap in failed");
346
347     swap_in_process_exists=0;
348     p->parent = 0;
349     p->name[0] = '*';
350     p->killed = 0;
351     p->state = UNUSED;
352     sched();
353
354 }

```

Task-4:Sanity Test

We will create a user-space program to test our swapping mechanism.

20 child processes are created using fork().

10 4Kb blocks of memory is allocated for each process.

For a given process_id <pid>, block number <j> and offset <k>, the memory location stored in address field is $pid*100000+j*10000+k$.

```

1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "fs.h"
5 #include <stddef.h>
6
7 int main(int argc, char *argv[])
8 {
9
10     int *add_list[10];
11     int id_list[100];
12     int counter = 0;
13     for (int i = 0; i < 20; i++)
14     {
15         if (fork() == 0)
16         {
17             counter = counter + 1;
18             for (int j = 0; j < 10; j++)
19             {
20                 int *addr = (int *)malloc(4096);
21                 int p_id;
22                 if ((add_list[j] = addr) == NULL)
23                 {
24                     p_id = getpid();
25                     printf(1, "the process ID is: %d\n", p_id);
26                     break;
27                 }
28                 p_id = getpid();
29                 id_list[counter] = p_id;
30                 for (int k = 0; k < 1024; k++)
31                 {
32                     *(addr + k) = p_id * 100000 + j * 10000 + k;
33                 }
34                 if (j == 0)
35                     printf(1, "block 1 index : %d, Beginning Address : %p ,",
36 process ID : %d\n", counter, add_list[j], p_id);
37                 if (j == 4)
38                     printf(1, "block 5 index : %d, Beginning Address : %p ,",
39 process ID : %d\n", counter, add_list[j], p_id);
40                 if (j == 9)
41                     printf(1, "block 10 index : %d, Beginning Address : %p ,",
42 process ID : %d\n", counter, add_list[j], p_id);
43             }
44             else
45                 break;
46         }
47         while (wait() != -1); // Execute all the child process
48         if (counter == 0)
49             exit();
50         for (int i = 0; i < 10; i++)
51         {
52             if (i == 0)
53                 printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 1st",
54 block: %d \n", add_list[i], id_list[counter], *(add_list[i]+100));
55             if (i == 4)
56                 printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 5th",
57 block: %d \n", add_list[i], id_list[counter], *(add_list[i]+100));
58             if (i == 9)
59                 printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 10th",
60 block: %d \n", add_list[i], id_list[counter], *(add_list[i]+100));
61         }
62         counter--;
63     }
64     exit();
65 }
66 }

```

Output:

- 1) Details of block 1, 5, 10 are displayed on the console.
- 2) After all child processes stop executing the contents of memory locations are checked.


```

$ sanity
block 1 index : 1, Beginning Address : A000 , process ID : 4
block 5 index : 1, Beginning Address : 5FE0 , process ID : 4
block 10 index : 1, Beginning Address : FFF0 , process ID : 4
block 1 index : 2, Beginning Address : EFE8 , process ID : 5
blockA failed : 2, Beginning Address : 1A000 , process ID : 5
$ ock 10 index : 2, Beginning Address : 14FD8 , process ID : 5
block 1 index : 3, Beginning Address : 13FD0 , process ID : 6
block 5 index : 3, Beginning Address : 1EFE8 , process ID : 6
block 10 index : 3, Beginning Address : 28FF8 , process ID : 6
block 1 index : 4, Beginning Address : 27FF0 , process ID : 7
block 5 index : 4, Beginning Address : 23FD0 , process ID : 7
blexec: failex : 4, Beginning Address : 2DFE0 , process ID : 7
exec 1 index : 5, Beginning Address : 2CFD8 , process ID : 8
block 5 index : 5, Beginning Address : 37FF0 , process ID : 8
block 10 index : 5, Beginning Address : 42000 , process ID : 8
block 1 index : 6, Beginning Address : 40FF8 , process ID : 9
block 5 index : 6, Beginning Address : 3CFD8 , process ID : 9
block 10 index : 6, Beginning Address : 46FE8 , process ID : 9
block 1 index : 7, Beginning Address : 45FE0 , process ID : 10
block 5 index : 7, Beginning Address : 50FF8 , process ID : 10
block 10 index : 7, Beginning Address : 4BFD0 , process ID : 10
block 1 index : 8, Beginning Address : 5A000 , process ID : 11
block 5 index : 8, Beginning Address : 55FE0 , process ID : 11
block 10 index : 8, Beginning Address : 5FFF0 , process ID : 11
block 1 index : 9, Beginning Address : 5EFE8 , process ID : 12
block 5 index : 9, Beginning Address : 6A000 , process ID : 12
block 10 index : 9, Beginning Address : 64FD8 , process ID : 12
Beginning Address : 5EFE8 ,Process ID: 12, 100th value of the 1st block: 120010
Beginning Address : 6A000 ,Process ID: 12, 100th value of the 5th block: 124010
Beginning Address : 64FD8 ,Process ID: 12, 100th value of the 10th block: 12901
Beginning Address : 5A000 ,Process ID: 11, 100th value of the 1st block: 110010
Beginning Address : 55FE0 ,Process ID: 11, 100th value of the 5th block: 114010
Beginning Address : 5FFF0 ,Process ID: 11, 100th value of the 10th block: 11901
Beginning Address : 45FE0 ,Process ID: 10, 100th value of the 1st block: 100010
Beginning Address : 50FF8 ,Process ID: 10, 100th value of the 5th block: 104010
Beginning Address : 4BFD0 ,Process ID: 10, 100th value of the 10th block: 10901
Beginning Address : 40FF8 ,Process ID: 9, 100th value of the 1st block: 900100
Beginning Address : 3CFD8 ,Process ID: 9, 100th value of the 5th block: 940100
Beginning Address : 46FE8 ,Process ID: 9, 100th value of the 10th block: 990100
Beginning Address : 2CFD8 ,Process ID: 8, 100th value of the 1st block: 800100
Beginning Address : 37FF0 ,Process ID: 8, 100th value of the 5th block: 840100
Beginning Address : 42000 ,Process ID: 8, 100th value of the 10th block: 890100
Beginning Address : 27FF0 ,Process ID: 7, 100th value of the 1st block: 700100
Beginning Address : 23FD0 ,Process ID: 7, 100th value of the 5th block: 740100
Beginning Address : 2DFE0 ,Process ID: 7, 100th value of the 10th block: 790100
Beginning Address : 13FD0 ,Process ID: 6, 100th value of the 1st block: 600100
Beginning Address : 1EFE8 ,Process ID: 6, 100th value of the 5th block: 640100
Beginning Address : 28FF8 ,Process ID: 6, 100th value of the 10th block: 690100
Beginning Address : EFE8 ,Process ID: 5, 100th value of the 1st block: 500100
Beginning Address : 1A000 ,Process ID: 5, 100th value of the 5th block: 540100
Beginning Address : 14FD8 ,Process ID: 5, 100th value of the 10th block: 590100
Beginning Address : A000 ,Process ID: 4, 100th value of the 1st block: 400100
Beginning Address : 5FE0 ,Process ID: 4, 100th value of the 5th block: 440100
Beginning Address : FFF0 ,Process ID: 4, 100th value of the 10th block: 490100

```

0x0400000

Results:-

- 1) When phystop=0xE000000, then we have all 20 processes in the output.
- 2) When phystop=0x0400000, then we have only 9 out of 20 processes in the output.

Explanation:-When we reduce the phystop then, we don't have enough capacity to hold all the processes in the main memory.

NOTE:- Part-A and Part-B are separately implemented on two different xv-6 directories.