CS 344
ASSIGNMENT 3

Memory Management in xv6

PART A

Three files were modified for this part namely :

1. **sysproc.c :** The **sbrk( )** function was changed so that physical memory wasn't allocated before it was needed. To implement lazy allocation, physical memory was allocated to a process only when needed. The part in **sbrk( )** which allocates space to the process was commented.

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

2. **trap.c :** An additional condition was added in the switch-case section. A page fault occurs when **tf->trapno** equals **T_PGFLT**, so this case was added and was handled appropriately. **kalloc( )** and **mappages( )** were called to allocate the required space or to inform that no more memory is available (code reused from **allocuvm( )** in **vm.c**).

```
case T_PGFLT:
 {
   char *mem;
   mem = kalloc();
   if (mem == 0)
   {
     cprintf("Out of Memory.\n");
   }
   else
   {
     memset(mem, 0, PGSIZE);

     // creating page table entry
     uint a = PGROUNDDOWN(rcr2());
     mappages(myproc()->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W |
PTE_U);
   }
 }

 break;
```

3. **vm.c :** Return type of function **mappages( )** was changed from static int to int (so that it can be used in **trap.c**)

```
// static int
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

PART B

Task 1

create_kernel_process( ) function was created in **proc.c** which took the name of process and entrypoint function as arguments. The parent process was set to **initproc** and **p->context->eip** was set equal to the entrypoint argument and all other values default.

```
void create_kernel_process(const char *name, void (*entrypoint)()){
 struct proc *p;
 // Allocate process.
 if((p = allocproc()) == 0){
   return;
 }
 // Setup page table.
 if((p->pgdir = setupkvm()) == 0){
   kfree(p->kstack);
   p->kstack = 0;
   p->state = UNUSED;
   panic("kernel process: out of memory?");
 }
  // Specify other parameters
 p->sz = PGSIZE;
 p->parent = initproc;
  memset(p->tf, 0, sizeof(*p->tf));
 p->tf->cs = (SEG_KCODE << 3) | DPL_USER;
 p->tf->ds = (SEG_KDATA << 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;  // beginning of initcode.S
 p->tf->eax = 0;
 p->cwd = namei("/");

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

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

 return;
}
```

Task 2

- In swap out mechanism whenever a request to a page which isn't in the main memory is made and physical memory is full, we select a victim by using **Least Recently Used algorithm.**
- An additional parameter is maintained with all the page table entries which stores the timestamp of the instant when the page was last referenced.
- We iterate through the page tables of all the processes in the process table and select the page table entry having the least time stamp as the victim.
- The victim is then swapped out of the memory, page table entry for the victim is deleted and the contents of the page are copied to a file named '**<pid>_<VA[20:0].swap**' which is stored in the secondary storage so that we can fetch the memory back into the physical memory whenever required.

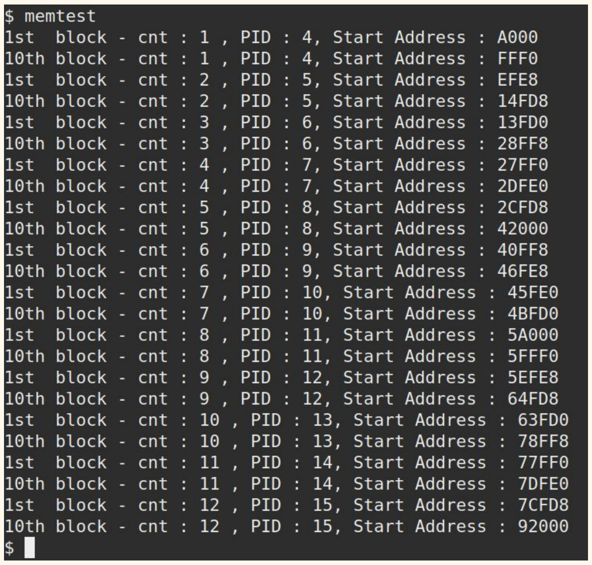
Task 3

- Whenever a page is requested which isn't in the page table of the process a call is made to the swap in mechanism.
- Swap in mechanism maintains a queue of all the requests and changes the process state to **SLEEPING** till the page is brought into memory.
- If there is already a free page in memory, swap_in simply loads the contents of file '**<pid>_<VA[20:0].swap**' into the free page in physical memory and creates the page table entry.
- If there is no free page, then a call to swap_out is made and as soon as a slot becomes free in memory, the required page is loaded in memory and page table entry is updated and the process is put back into **RUNNABLE** state.

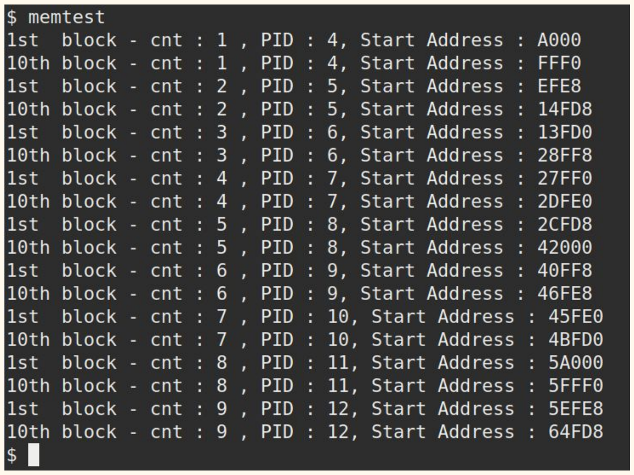
Task 4

```
int main(int argc, char *argv[])
{
   int cnt = 0;
   int *address[10];
   int cnt_to_pid[100];

   for (int i = 0; i < 20; i++){
       if (fork() == 0){
           cnt++;
           for (int j = 0; j < 10; j++){
               int *ptr = (int *)malloc(4096);
               address[j] = ptr;

               if (ptr == NULL){
                   printf(1, "PID: %d\n", getpid());
                   break;
               }

               int pid = getpid();
               cnt_to_pid[cnt] = pid;

               for (int k = 0; k < 1024; k++){
                   int value = pid * 100000 + j * 10000 + k;
                   *(ptr + k) = value;
               }
               if (j == 0)
                   printf(1, "1st  block - cnt : %d , PID : %d, Start Address : %p\n", cnt, pid, address[j]);
               if (j == 9)
                   printf(1, "10th block - cnt : %d , PID : %d, Start Address : %p\n", cnt, pid, address[j]);
           }
       }
       else
           break;
   }
   while (wait() != -1); // Execute all the child process

   if (cnt == 0)
       exit();

   for (int i = 0; i < 10; i++){
       if (i == 0)
   printf(1, "PID : %d ,Start Address: %p, First value in 1st  block: %d \n", cnt_to_pid[cnt], address[i],
*address[i]);
       if (i == 9)
   printf(1, "PID : %d ,Start Address: %p, First value in 10th block: %d \n", cnt_to_pid[cnt], address[i],
*address[i]);
   }
   cnt--;
   exit();
}
```

- 20 child processes were created using the **fork( )** function .
- 10 blocks of memory, each of size 4KB, were allocated for each child process.
- 4KB memory -> 1024 integers could be stored in the given memory block as the size of an integer variable is 4 bytes.
- For a given **Process Id (pid) , block number (int j) and offset (int k)** the value stored in the that memory location is **Process Id *100000 + block number *10000 + offset.**
- We have also maintained the count of the number of child processes i.e variable cnt.
- Each cnt value is mapped to its corresponding Process Id.
- **while(wait( ) != -1)** is used to wait for completion of the execution of all the child processes.
- To save the amount of space consumed by the output on the console only the **1st and the 10th blocks details are printed on the console.**
- After all the child processes are executed the contents of the memory locations are checked.
- It is interesting to note that when we change the value of **PHYSTOP** in **memlayout.h** file, the number of processes for which the memory is allocated changes. Higher value of **PHYSTOP** implies more number of processes are allotted the memory.



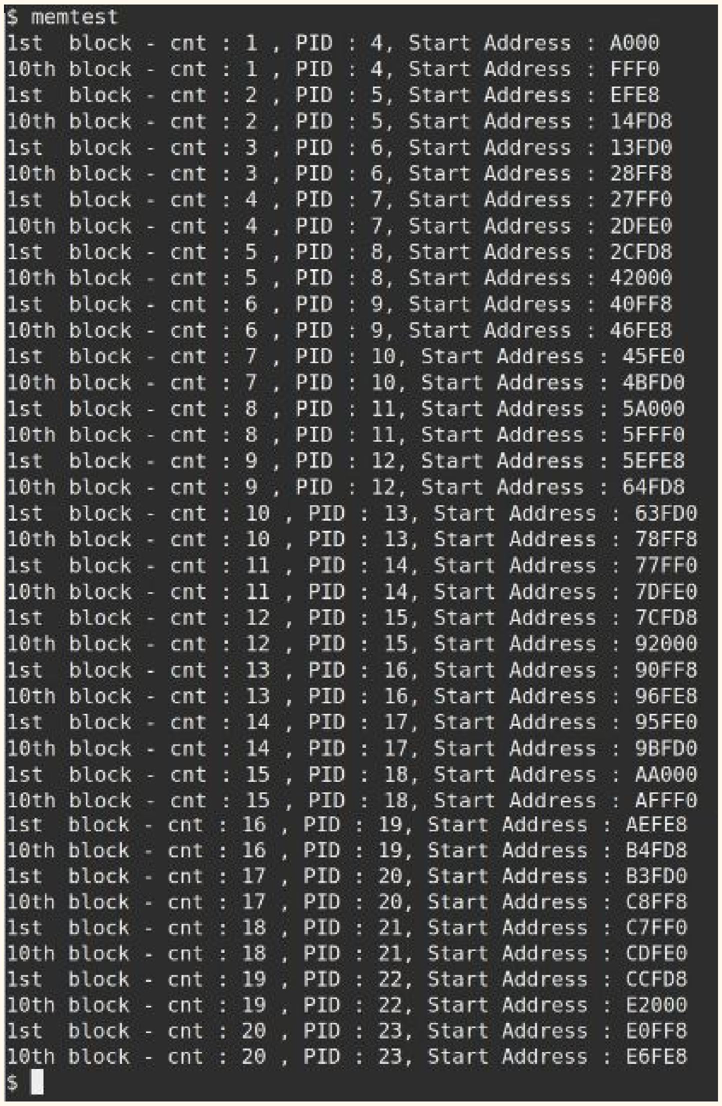
```
$ memtest
1st  block - cnt : 1 , PID : 4, Start Address : A000
10th block - cnt : 1 , PID : 4, Start Address : FFF0
1st  block - cnt : 2 , PID : 5, Start Address : EFE8
10th block - cnt : 2 , PID : 5, Start Address : 14FD8
1st  block - cnt : 3 , PID : 6, Start Address : 13FD0
10th block - cnt : 3 , PID : 6, Start Address : 28FF8
1st  block - cnt : 4 , PID : 7, Start Address : 27FF0
10th block - cnt : 4 , PID : 7, Start Address : 2DFE0
1st  block - cnt : 5 , PID : 8, Start Address : 2CFD8
10th block - cnt : 5 , PID : 8, Start Address : 42000
1st  block - cnt : 6 , PID : 9, Start Address : 40FF8
10th block - cnt : 6 , PID : 9, Start Address : 46FE8
1st  block - cnt : 7 , PID : 10, Start Address : 45FE0
10th block - cnt : 7 , PID : 10, Start Address : 4BFD0
1st  block - cnt : 8 , PID : 11, Start Address : 5A000
10th block - cnt : 8 , PID : 11, Start Address : 5FFF0
1st  block - cnt : 9 , PID : 12, Start Address : 5EFE8
10th block - cnt : 9 , PID : 12, Start Address : 64FD8
1st  block - cnt : 10 , PID : 13, Start Address : 63FD0
10th block - cnt : 10 , PID : 13, Start Address : 78FF8
1st  block - cnt : 11 , PID : 14, Start Address : 77FF0
10th block - cnt : 11 , PID : 14, Start Address : 7DFE0
1st  block - cnt : 12 , PID : 15, Start Address : 7CFD8
10th block - cnt : 12 , PID : 15, Start Address : 92000
$ 
```



```
$ memtest
1st  block - cnt : 1 , PID : 4, Start Address : A000
10th block - cnt : 1 , PID : 4, Start Address : FFF0
1st  block - cnt : 2 , PID : 5, Start Address : EFE8
10th block - cnt : 2 , PID : 5, Start Address : 14FD8
1st  block - cnt : 3 , PID : 6, Start Address : 13FD0
10th block - cnt : 3 , PID : 6, Start Address : 28FF8
1st  block - cnt : 4 , PID : 7, Start Address : 27FF0
10th block - cnt : 4 , PID : 7, Start Address : 2DFE0
1st  block - cnt : 5 , PID : 8, Start Address : 2CFD8
10th block - cnt : 5 , PID : 8, Start Address : 42000
1st  block - cnt : 6 , PID : 9, Start Address : 40FF8
10th block - cnt : 6 , PID : 9, Start Address : 46FE8
1st  block - cnt : 7 , PID : 10, Start Address : 45FE0
10th block - cnt : 7 , PID : 10, Start Address : 4BFD0
1st  block - cnt : 8 , PID : 11, Start Address : 5A000
10th block - cnt : 8 , PID : 11, Start Address : 5FFF0
1st  block - cnt : 9 , PID : 12, Start Address : 5EFE8
10th block - cnt : 9 , PID : 12, Start Address : 64FD8
$ 
```

PHYSTOP : 0x600000

PHYSTOP : 0x400000



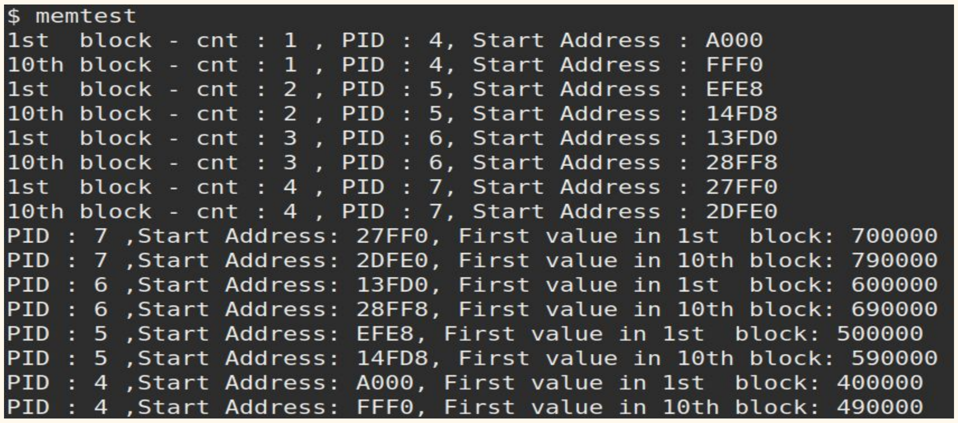
```
$ memtest
1st  block - cnt : 1 , PID : 4, Start Address : A000
10th block - cnt : 1 , PID : 4, Start Address : FFF0
1st  block - cnt : 2 , PID : 5, Start Address : EFE8
10th block - cnt : 2 , PID : 5, Start Address : 14FD8
1st  block - cnt : 3 , PID : 6, Start Address : 13FD0
10th block - cnt : 3 , PID : 6, Start Address : 28FF8
1st  block - cnt : 4 , PID : 7, Start Address : 27FF0
10th block - cnt : 4 , PID : 7, Start Address : 2DFE0
1st  block - cnt : 5 , PID : 8, Start Address : 2CFD8
10th block - cnt : 5 , PID : 8, Start Address : 42000
1st  block - cnt : 6 , PID : 9, Start Address : 40FF8
10th block - cnt : 6 , PID : 9, Start Address : 46FE8
1st  block - cnt : 7 , PID : 10, Start Address : 45FE0
10th block - cnt : 7 , PID : 10, Start Address : 4BFD0
1st  block - cnt : 8 , PID : 11, Start Address : 5A000
10th block - cnt : 8 , PID : 11, Start Address : 5FFF0
1st  block - cnt : 9 , PID : 12, Start Address : 5EFE8
10th block - cnt : 9 , PID : 12, Start Address : 64FD8
1st  block - cnt : 10 , PID : 13, Start Address : 63FD0
10th block - cnt : 10 , PID : 13, Start Address : 78FF8
1st  block - cnt : 11 , PID : 14, Start Address : 77FF0
10th block - cnt : 11 , PID : 14, Start Address : 7DFE0
1st  block - cnt : 12 , PID : 15, Start Address : 7CFD8
10th block - cnt : 12 , PID : 15, Start Address : 92000
1st  block - cnt : 13 , PID : 16, Start Address : 90FF8
10th block - cnt : 13 , PID : 16, Start Address : 96FE8
1st  block - cnt : 14 , PID : 17, Start Address : 95FE0
10th block - cnt : 14 , PID : 17, Start Address : 9BFD0
1st  block - cnt : 15 , PID : 18, Start Address : AA000
10th block - cnt : 15 , PID : 18, Start Address : AFFF0
1st  block - cnt : 16 , PID : 19, Start Address : AEFE8
10th block - cnt : 16 , PID : 19, Start Address : B4FD8
1st  block - cnt : 17 , PID : 20, Start Address : B3FD0
10th block - cnt : 17 , PID : 20, Start Address : C8FF8
1st  block - cnt : 18 , PID : 21, Start Address : C7FF0
10th block - cnt : 18 , PID : 21, Start Address : CDFE0
1st  block - cnt : 19 , PID : 22, Start Address : CCFD8
10th block - cnt : 19 , PID : 22, Start Address : E2000
1st  block - cnt : 20 , PID : 23, Start Address : E0FF8
10th block - cnt : 20 , PID : 23, Start Address : E6FE8
$ 
```

PHYSTOP : 0xE000000

Observe that due to the large value of PHYSTOP all the child processes are allocated the requested amount of memory.

The second image is a dummy image wherein only 4 child processes were created to check the sanity i.e to check the values stored in the memory locations after all the child processes are done with their execution.

We observe that the retrieved values in the memory locations match with values which were stored in the corresponding memory location.



```
$ memtest
1st  block - cnt : 1 , PID : 4, Start Address : A000
10th block - cnt : 1 , PID : 4, Start Address : FFF0
1st  block - cnt : 2 , PID : 5, Start Address : EFE8
10th block - cnt : 2 , PID : 5, Start Address : 14FD8
1st  block - cnt : 3 , PID : 6, Start Address : 13FD0
10th block - cnt : 3 , PID : 6, Start Address : 28FF8
1st  block - cnt : 4 , PID : 7, Start Address : 27FF0
10th block - cnt : 4 , PID : 7, Start Address : 2DFE0
PID : 7 ,Start Address: 27FF0, First value in 1st  block: 700000
PID : 7 ,Start Address: 2DFE0, First value in 10th block: 790000
PID : 6 ,Start Address: 13FD0, First value in 1st  block: 600000
PID : 6 ,Start Address: 28FF8, First value in 10th block: 690000
PID : 5 ,Start Address: EFE8, First value in 1st  block: 500000
PID : 5 ,Start Address: 14FD8, First value in 10th block: 590000
PID : 4 ,Start Address: A000, First value in 1st  block: 400000
PID : 4 ,Start Address: FFF0, First value in 10th block: 490000
```