OS Assignment 3 Report

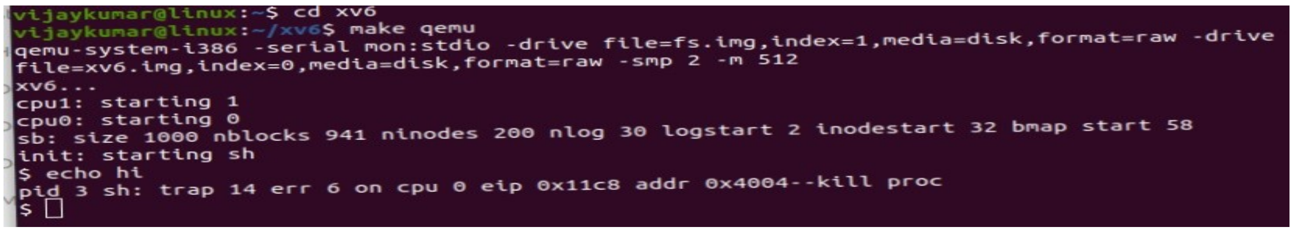
Group 44 : Kottakota Vijaykumar 190123031
Sankranthi Karthik 190123053
Mandala Yatish Ram Kumar 190123036
Lakavath Pavan Kalyan 190123033

Part A:

- Lazy Memory Allocation is ‘not’ allocating memory to a process unless and until it is really needed. By this, we can eliminate the allocation if we donot use the process. Xv6 applications ask the kernel for heap memory using the sbrk() system call. In the original xv6 kernel, sbrk() allocates physical memory and map it into the process virtual address space.

In this lab, we added support for this lazy allocation feature in xv6, by delaying the memory requested by sbrk() until the process actually uses it.

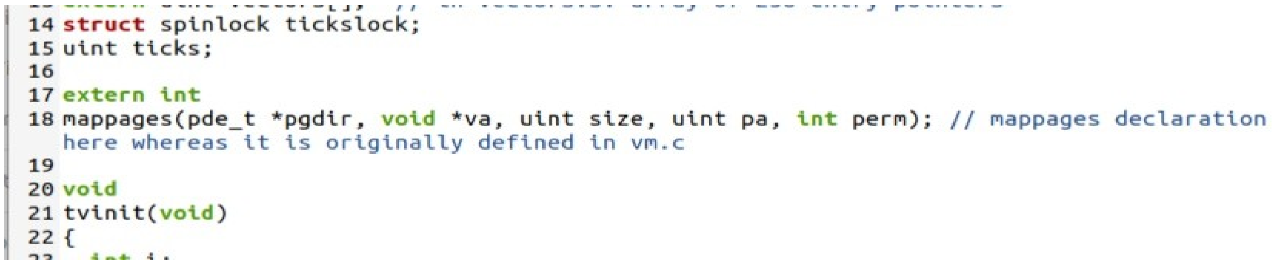
1) Eliminate allocation form sbrk():
We used the given patch file in which the call to function growproc is commented, to delete page allocation from sbrk() syscall implementation in sysproc.c . Our new sbrk() will just increment the process size by n bytes and return the old size. However, it still increases the process size(proc->sz) by n bytes to trick the process into believing that it has the memory requested even though memory is not available.



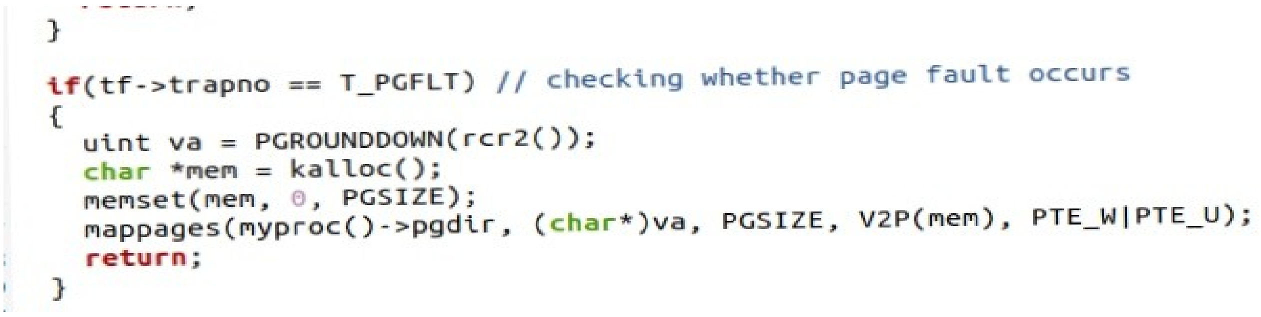
```
vijaykumar@linux:~$ cd xv6
vijaykumar@linux:~/xv6$ make qemu
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x11c8 addr 0x4004--kill proc
$ 
```

This address “0x4004” is the virtual address that caused page fault. After implementation of lazy allocation correctly, this page fayult will disapper and echo hi will reply “hi”.

2) Lazy Allocation: Now our modified function trap() will map a newly allocated page of physical memory memory at the page fault address to the page fault, then return to user space and again the process will be continuing execution. New memory page is allocated and added suitable page table entries. So that the process can avoid page fault next time.
Here are some segments of code we have added in trap.c:



```
struct spinlock tickslock;
uint ticks;

extern int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm); // mappages declaration
here whereas it is originally defined in vm.c

void
tvinit(void)
{
```



```
}

if(tf->trapno == T_PGFLT) // checking whether page fault occurs
{
  uint va = PGROUNDDOWN(rcr2());
  char *mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(myproc()->pgdir, (char*)va, PGSIZE, V2P(mem), PTE_W|PTE_U);
  return;
}
```

The above lines of code are actually as follows:

- if tf->trapno == T_PGFLT is to check whether a fault is a page fault.
- PGROUNDUP(rcr2()) is to round the faulting virtual address down to the start of a page boundary.
- Kalloc() allocates one page of 4kB from physical memory and returns a pointer to that page.
- Memset(, 0,); makes the whole page null when it is allocated because a newly allocated should be empty.
- Mappages(...) maps the page to our process page dir by converting the given virtual address to physical address by using V2P(mem). It creates a new page table entry PTE for that particular virtual address. We use this in lazy allocation implementation is by passing myproc() which contains the present process dir and rcr2() which gives the address at which page fault occurs and the rest of the implementation is done in the lazy allocation. The page table flags used in mappages() are PTE_W or PTE_U which declares the page is user accessible and writeable.
- So, our function takes the faulty address, rounds up and makes it page allocated, allocates a fresh page from physical memory to process using mappages() and process starts to execute normally if allocation is done perfectly.

Part B:

XV6 Memory Management

Answers to the questions given in the assignment:

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

- xv6 keeps a record of the phyical memory which is available, to be used by the processes that are to be run. Xv6 allocates the physical memory using pages. It maintains a linked list of all physical pages currently available and deletes the newly allocated pages from the list and adds freed pages to the list.

Q2. What data structures are used to answer this question?

- We used ‘struct run’ for maintaining the free page. Datastructure used here is a singly linked list with no data and a pointer to the next node and this linked list is protected by a lock . So both are enclosed in a structure to emphasize that the lock protects the list.

Q3. Where do these reside?

- These (allocator implementation and the data structures)reside in the file kalloc.c.

Q4. Does xv6 memory mechanism limit the number of user proceses?

- Yes xv6 memory mechanism limit the number of user processes because xv6 doesnt have paging technique to disk by default, so when it allocates memory for the user process it keeps it in the physical memory till the process gets terminated so, Since physical memory is bounded only certain amount of processes can reside in the physical memory at the given time.

Q5. If so, what is the lowest number of processes xv6 can ‘have’ at the same time(assuming the kernel requires no memory whatsoever)?

- In xv6 os, the minimum number of processses running at the same time will be **1** process named intiproc (this process forks the sh process which forks other user processes).

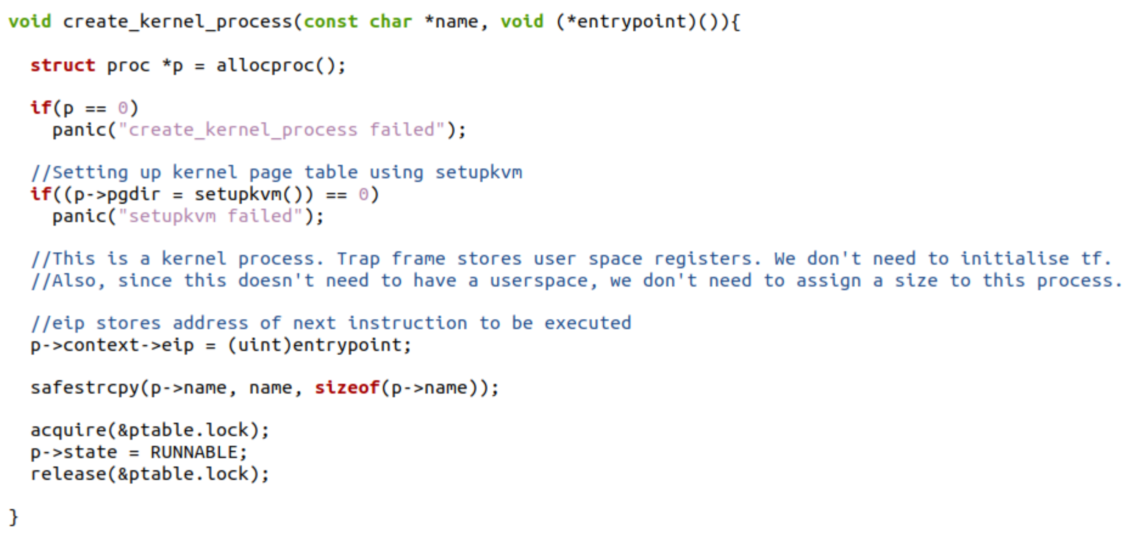
Task 1:

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

We added this function in the file **proc.c**. The kernel process will remain in kernel mode the whole time. Thus, we do not need to initialise its trapframe, userspace and the user section of the page table.The eip register of the process context stores address of the next instruction so since entrypoint is the address of the function where we want to start executing

so we have written p->context->eip=(unit)entrypoint;

allocproc assigns the process a spot in process table. Setupkvm sets up kernel part of the process page table that maps virtual address above KERNBASE to physical addresses between 0 and PHYSTOP.

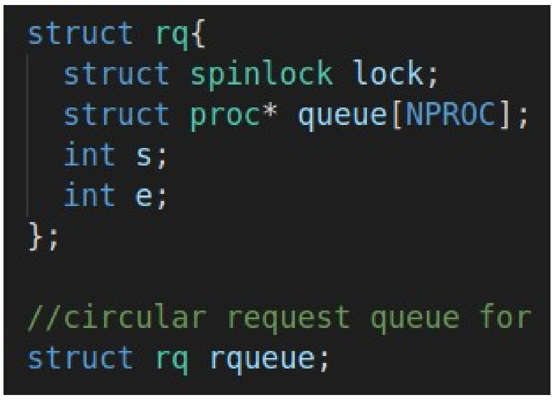


```
void create_kernel_process(const char *name, void (*entrypoint)()){

  struct proc *p = allocproc();

  if(p == 0)
    panic("create_kernel_process failed");

  //Setting up kernel page table using setupkvm
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");

  //This is a kernel process. Trap frame stores user space registers. We don't need to initialise tf.
  //Also, since this doesn't need to have a userspace, we don't need to assign a size to this process.

  //eip stores address of next instruction to be executed
  p->context->eip = (uint)entrypoint;

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

  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);

}
```

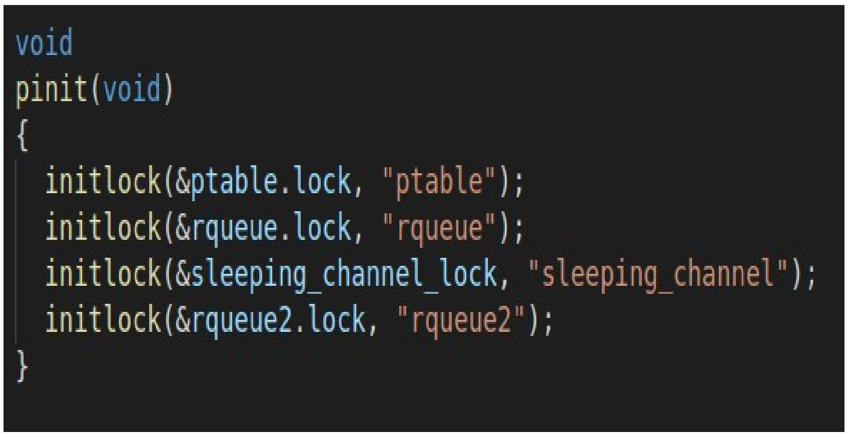
Task 2:

Swapping out mechanism:

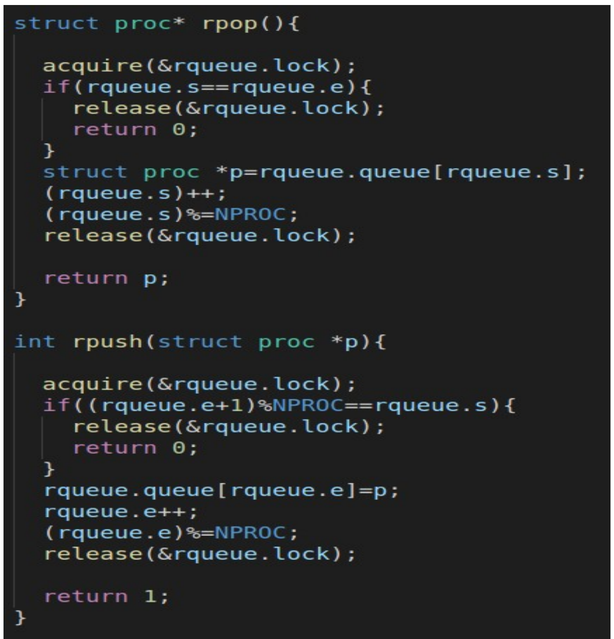
- First we need a process queue that keeps track of the processes that were refused additional memory since there were no free pages available. We created a circular queue struct called rq. And queue for swap out requests is rqueue. We also implemented the fuctions rpush(), rpop() for queue rq. The queue needs to be accessed with a lock that we have initialized in pinit. We have also initialised values of s,e to zero in userinit. We also added prototypes in defs.h for using them in other files.



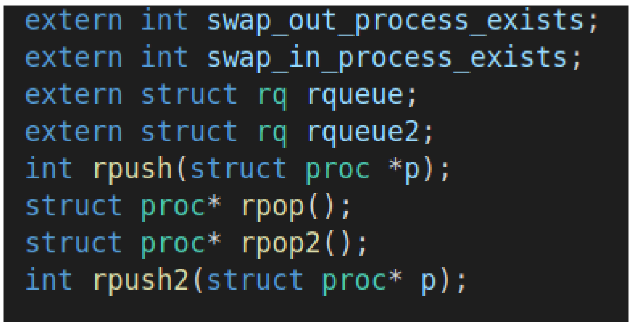
```
struct rq{
  struct spinlock lock;
  struct proc* queue[NPROC];
  int s;
  int e;
};

//circular request queue for
struct rq rqueue;
```



```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&rqueue.lock, "rqueue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&rqueue2.lock, "rqueue2");
}
```

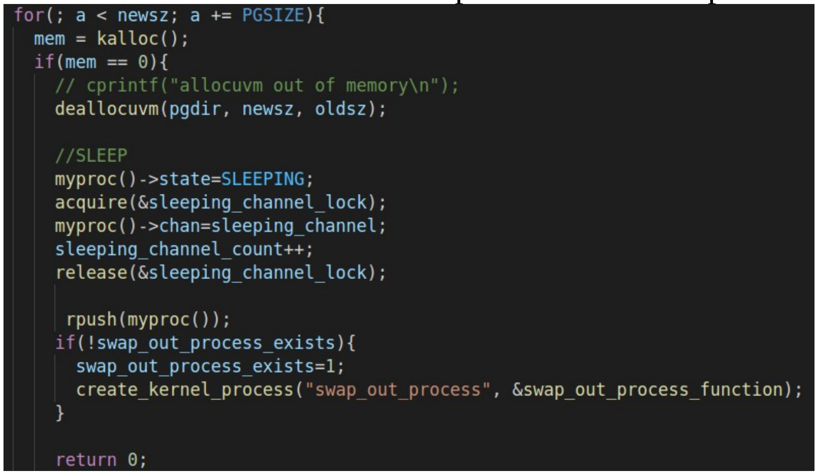


```
struct proc* rpop(){

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

  return p;
}

int rpush(struct proc *p){

  acquire(&rqueue.lock);
  if((rqueue.e+1)%NPROC==rqueue.s){
    release(&rqueue.lock);
    return 0;
  }
  rqueue.queue[rqueue.e]=p;
  rqueue.e++;
  (rqueue.e)%=NPROC;
  release(&rqueue.lock);

  return 1;
}
```



```
extern int swap_out_process_exists;
extern int swap_in_process_exists;
extern struct rq rqueue;
extern struct rq rqueue2;
int rpush(struct proc *p);
struct proc* rpop();
struct proc* rpop2();
int rpush2(struct proc* p);
```

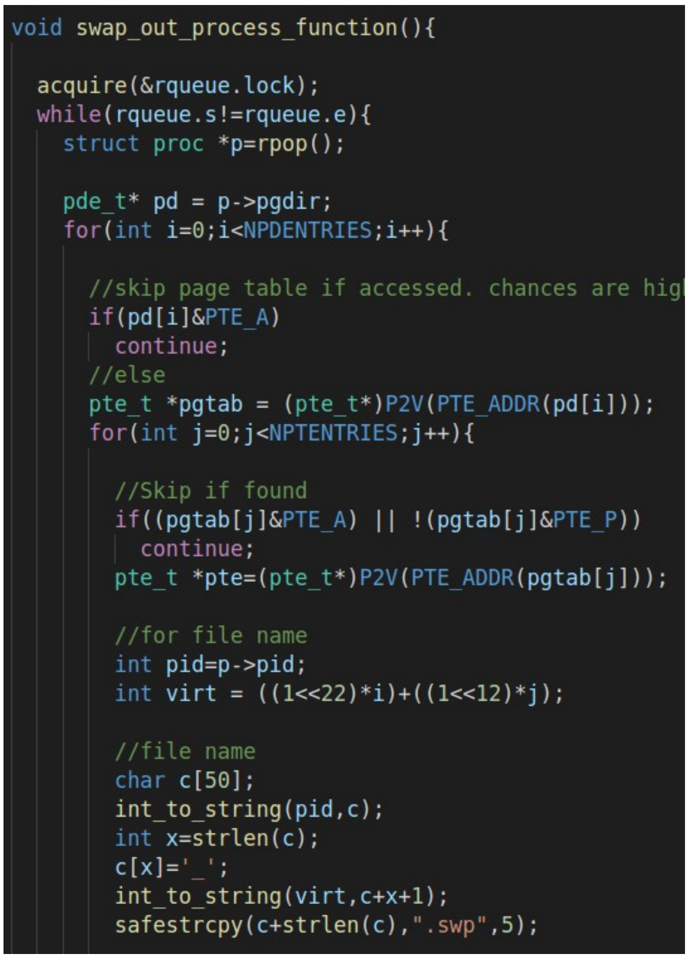
- **Kalloc** returns zero when it is not able to allocate pages to process. Then this notifies **allocuvm** that requested memory was not alloted (mem=0). Then first we need to change the process state to sleeping and then we need to add process to the swap out request to the rqueue.



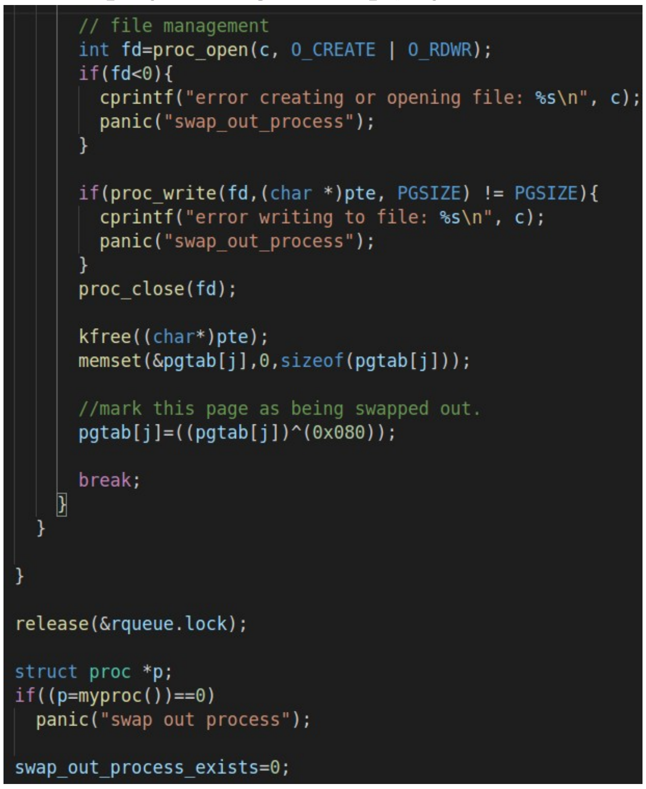
```c
for(; a < newsz; a += PGSIZE){
  mem = kalloc();
  if(mem == 0){
    // cprintf("allocuvm out of memory\n");
    deallocuvm(pgdir, newsz, oldsz);

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

     rpush(myproc());
    if(!swap_out_process_exists){
      swap_out_process_exists=1;
      create_kernel_process("swap_out_process", &swap_out_process_function);
    }

    return 0;
```

Note: create_kernel_process here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. When the swap out process ends, the swap_out_process_exists(initialised in proc.c to 0) variable is set to 0. when it is created, it is set to 1 . This is done so multiple swap out processes are not created.

- Next, we create a mechanism by which whenever free pages are available, all the processes sleeping on sleeping_channel are woken up by calling wakeup() system call.



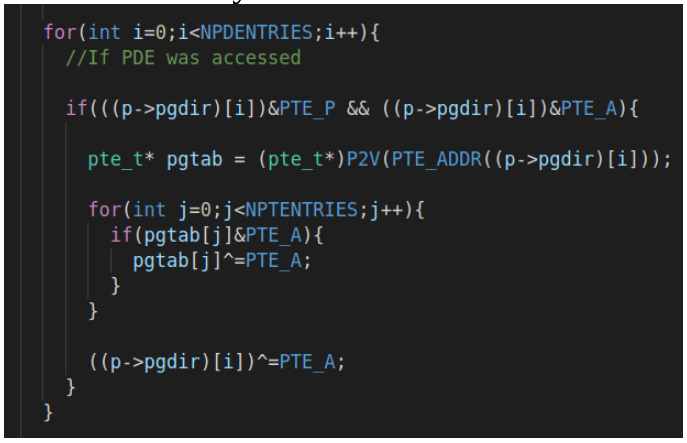
```c
void swap_out_process_function(){

  acquire(&rqueue.lock);
  while(rqueue.s!=rqueue.e){
    struct proc *p=rpop();

    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){

      //skip page table if accessed. chances are hig
      if(pd[i]&PTE_A)
        continue;
      //else
      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
      for(int j=0;j<NPTENTRIES;j++){

        //Skip if found
        if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P))
          continue;
        pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));

        //for file name
        int pid=p->pid;
        int virt = ((1<<22)*i)+((1<<12)*j);

        //file name
        char c[50];
        int_to_string(pid,c);
        int x=strlen(c);
        c[x]='_';
        int_to_string(virt,c+x+1);
        safestrcpy(c+strlen(c),".swp",5);
```



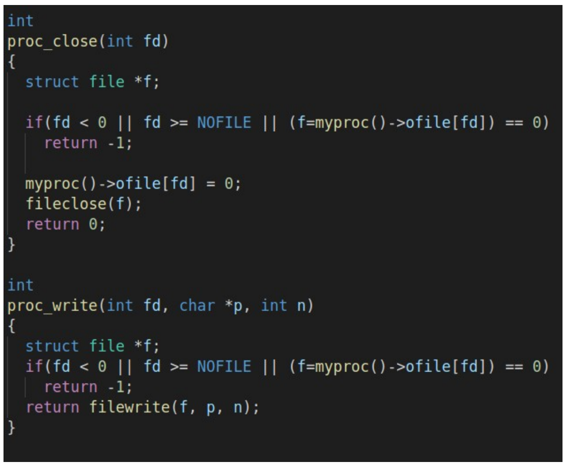
```c
        // file management
        int fd=proc_open(c, O_CREATE | O_RDWR);
        if(fd<0){
          cprintf("error creating or opening file: %s\n", c);
          panic("swap_out_process");
        }

        if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
          cprintf("error writing to file: %s\n", c);
          panic("swap_out_process");
        }
        proc_close(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(&rqueue.lock);

  struct proc *p;
  if((p=myproc())==0)
    panic("swap out process");

  swap_out_process_exists=0;
```

- • Now, The entry point for the swapping out process in swap_out_process_function. The process runs a loop until the rqueue is non empty. When the queue is empty, a set of instructions are executed for the termination of swap_out_process. The loop starts by popping the first process from rqueue and uses LRU policy to determine victim page in its page table. We iterate through pgdir and extracts the physical address for each secondary page table.For each secondary page table we iterate and look at accessed bit(A) and check if it is set by checking bitwise & of the entry and PTE_A.

Note: Whenever the process is being context switched into by scheduler, all accessed bits are unset. So accessed bit seen by the swap out function will indicate whether entry was accessed in the last iteration of the process or not.



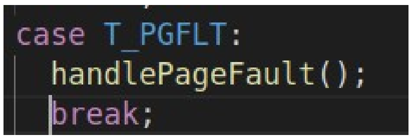
```c
for(int i=0;i<NPDENTRIES;i++){
  //If PDE was accessed

  if(((p->pgdir)[i])&PTE_P && ((p->pgdir)[i])&PTE_A){

    pte_t* pgtab = (pte_t*)P2V(PTE_ADDR((p->pgdir)[i]));

    for(int j=0;j<NPTENTRIES;j++){
      if(pgtab[j]&PTE_A){
        pgtab[j]^=PTE_A;
      }
    }

    ((p->pgdir)[i])^=PTE_A;
  }
}
```



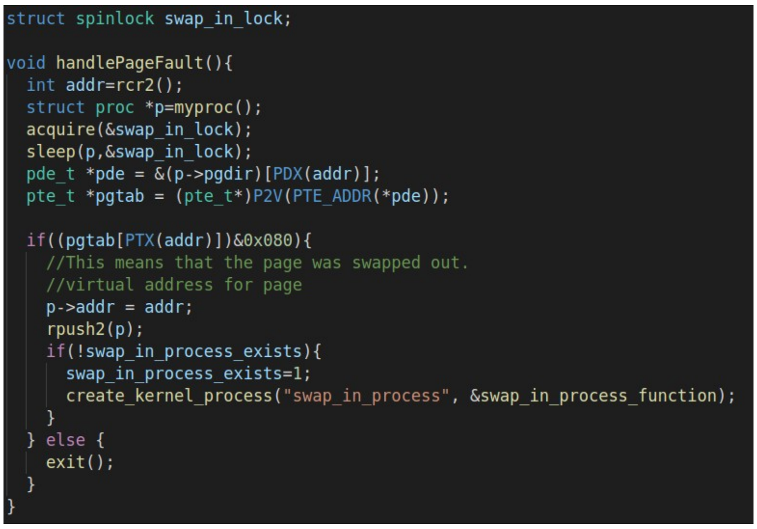
```c
int
proc_close(int fd)
{
  struct file *f;

  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;

  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}

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

- Now, back to swap_out_process_function. As soon as the function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number as the victim page. This page is then swapped out and stored to drive.
- We need to write the contents of the victim page to the file with the name <pid>_<virt>.swp. But we encounter a problem here we store the filename in a string called c. File system calls cannot be called from proc.c. The solution was that we copied the open,write, read, close etc. Functions from sysfile.c to proc.c, modifed them since the sysfile.c functions used a different way to take arguments and then renamed them to proc_open, proc_read, proc_write, proc_close etc. So we can use them in proc.c.

Task 3:



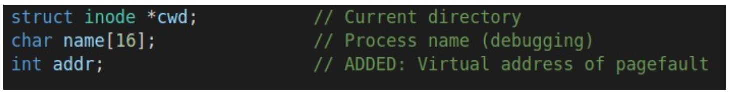
```c
case T_PGFLT:
  handlePageFault();
  break;
```



```c
struct spinlock swap_in_lock;

void handlePageFault(){
  int addr=rcr2();
  struct proc *p=myproc();
  acquire(&swap_in_lock);
  sleep(p,&swap_in_lock);
  pde_t *pde = &(p->pgdir)[PDX(addr)];
  pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

  if((pgtab[PTX(addr)])&0x080){
    //This means that the page was swapped out.
    //virtual address for page
    p->addr = addr;
    rpush2(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

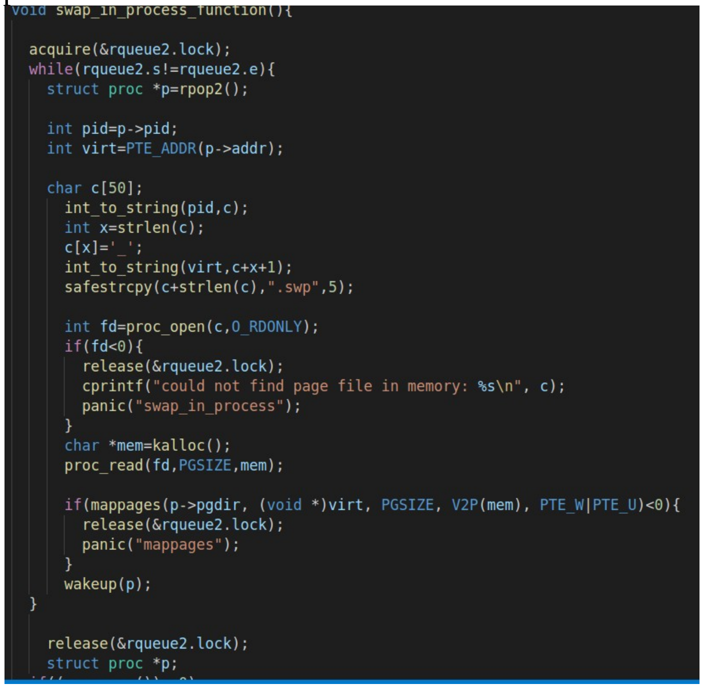
- We need to handle pagefault in the handlePageFault we need to find the virtual address at which the page fault occured by using rcr2(). We then put the current process to sleep with a new lock called swap_in_lock. We then obtain page table entry corresponding to this address . Now we need to check whether this page is swapped out . In task 2, whenever we swapped out a page, we set its page table entry's bit of 7th order. Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using bitwise & with 0x080.If it is set, we initiate swap_in_process (if it doesn't already exist - check using swap_in_process_exists). Otherwise, we safely suspend the process using exit() .

- Now we need to create a swap in request queue.We used the same struct rq as in Task 2 to create a swap in request queue called rqueue2 in proc.c. We also declare an extern prototype for rqueue2 in defs.h. Along with declaring the queue, we also created the corresponding functions for rqueue2(rpop2() and rpush2()) in proc.c and declared their prototype in defs.h. We also initialised its lock in pinit. We also initialised its s,e variables in userinit.



```
struct inode *cwd;           // Current directory
char name[16];               // Process name (debugging)
int addr;                    // ADDED: Virtual address of pagefault
```

- Now we add an additional entry to the struct proc in proc.h called addr(int). This entry will tell the swapping in function at which virtual address the page fault occured.
- Now, we go through the swapping in process.The entry point for the swapping out process is swap_in_process_function (declared in proc.c) as you can see in handlePageFault. The function runs a loop until rqueue2 is not empty. In the loop, it pops a process from the queue and extracts its pid and addr value to get the file name. Then, it creates the filename in a string called "c" using int_to_string.Then, it used proc_open to open this file in read only mode with file descriptor fd. We then allocate a free frame (mem) to this process using kalloc. We read from the file with the fd file descriptor into this free frame using proc_read. We then make mappages available to proc.c and then declaring a prototype in proc.c. We then use mappages to map the page corresponding to addr with the physical page that got using kalloc and read into (mem). Then we wake up, the process for which we allocated a new page to fix the page fault using wakeup. Once the loop is completed, we run the kernel process termination instructions.



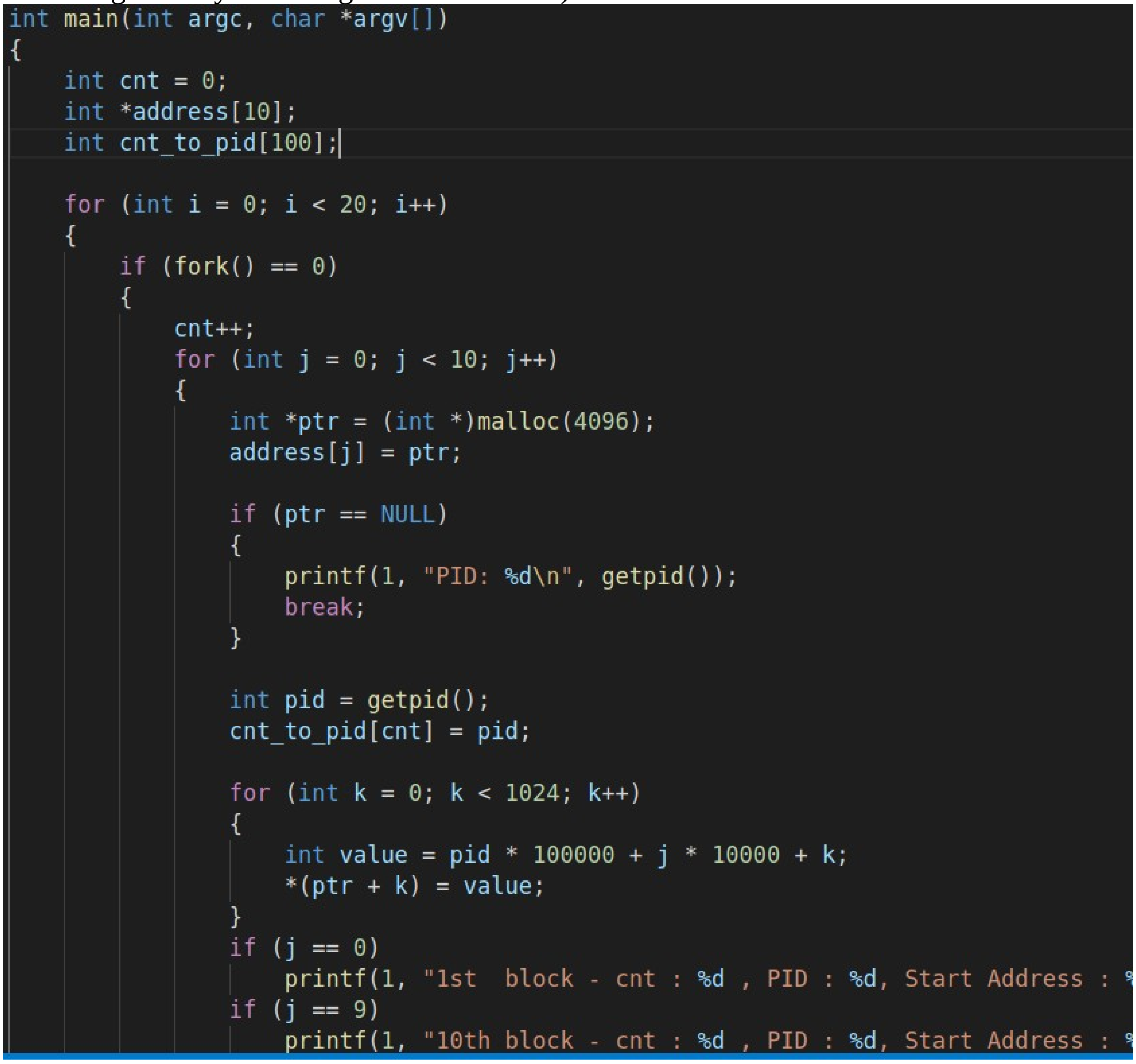
```
void swap_in_process_function(){

  acquire(&rqueue2.lock);
  while(rqueue2.s!=rqueue2.e){
    struct proc *p=rpop2();

    int pid=p->pid;
    int virt=PTE_ADDR(p->addr);

    char c[50];
      int_to_string(pid,c);
      int x=strlen(c);
      c[x]='_';
      int_to_string(virt,c+x+1);
      safestrcpy(c+strlen(c),".swp",5);

      int fd=proc_open(c,O_RDONLY);
      if(fd<0){
        release(&rqueue2.lock);
        cprintf("could not find page file in memory: %s\n", c);
        panic("swap_in_process");
      }
      char *mem=kalloc();
      proc_read(fd,PGSIZE,mem);

      if(mappages(p->pgdir, (void *)virt, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
        release(&rqueue2.lock);
        panic("mappages");
      }
      wakeup(p);
  }

    release(&rqueue2.lock);
    struct proc *p;
```

Task 4:

The essence of this task is to write a userprogram to the memory management techniques we implemented in above tasks.

The userprogram is in memtest.c file and we have been instructed that the main program should create 20 child processes(using fork()) and each child will be need to allocated 4KB

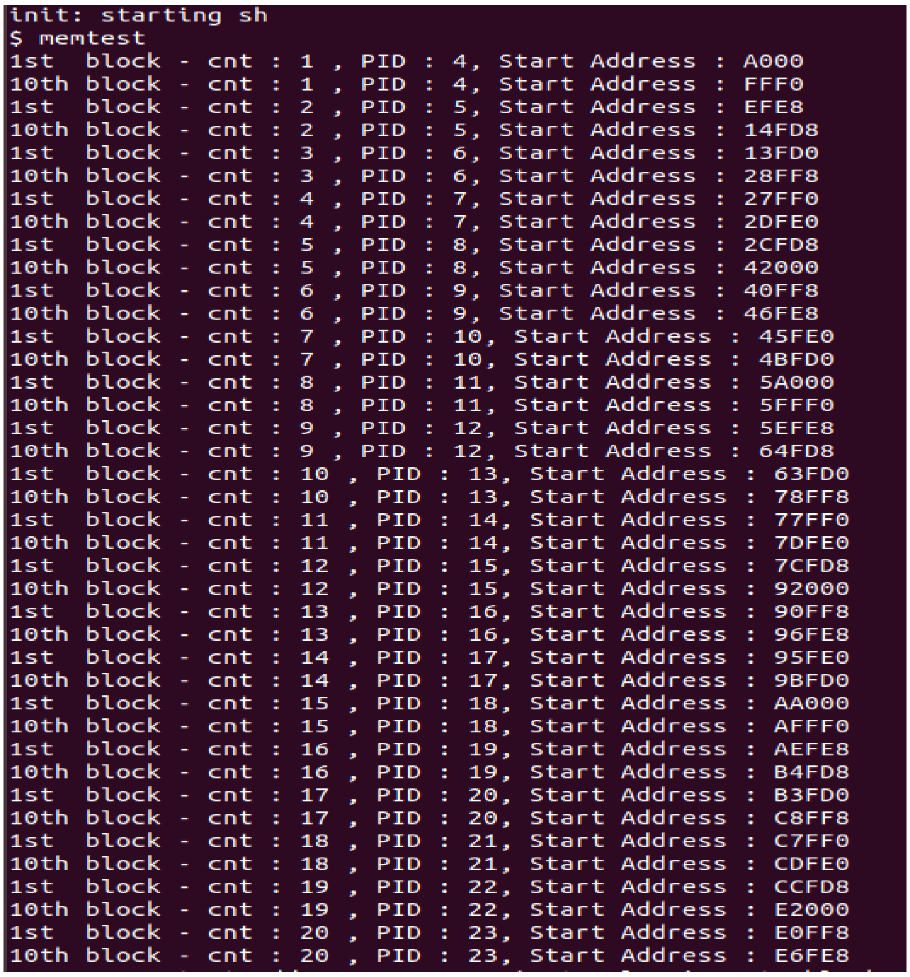
for each of 10 iterations on a whole and the memory location is filled with the expression(pid*100000+10000*j + k).
we store the number of child processes created in the variable cnt. The main process will wait for the child processes to terminate using the wait function in the while loop.
Output format is that we print the details of the first and tenth block for each process(you can change this by removing the if statements).



```
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 : %
                if (j == 9)
                    printf(1, "10th block - cnt : %d , PID : %d, Start Address : %
```

Checking the paging mechanism by changing value of PHYSTOP :

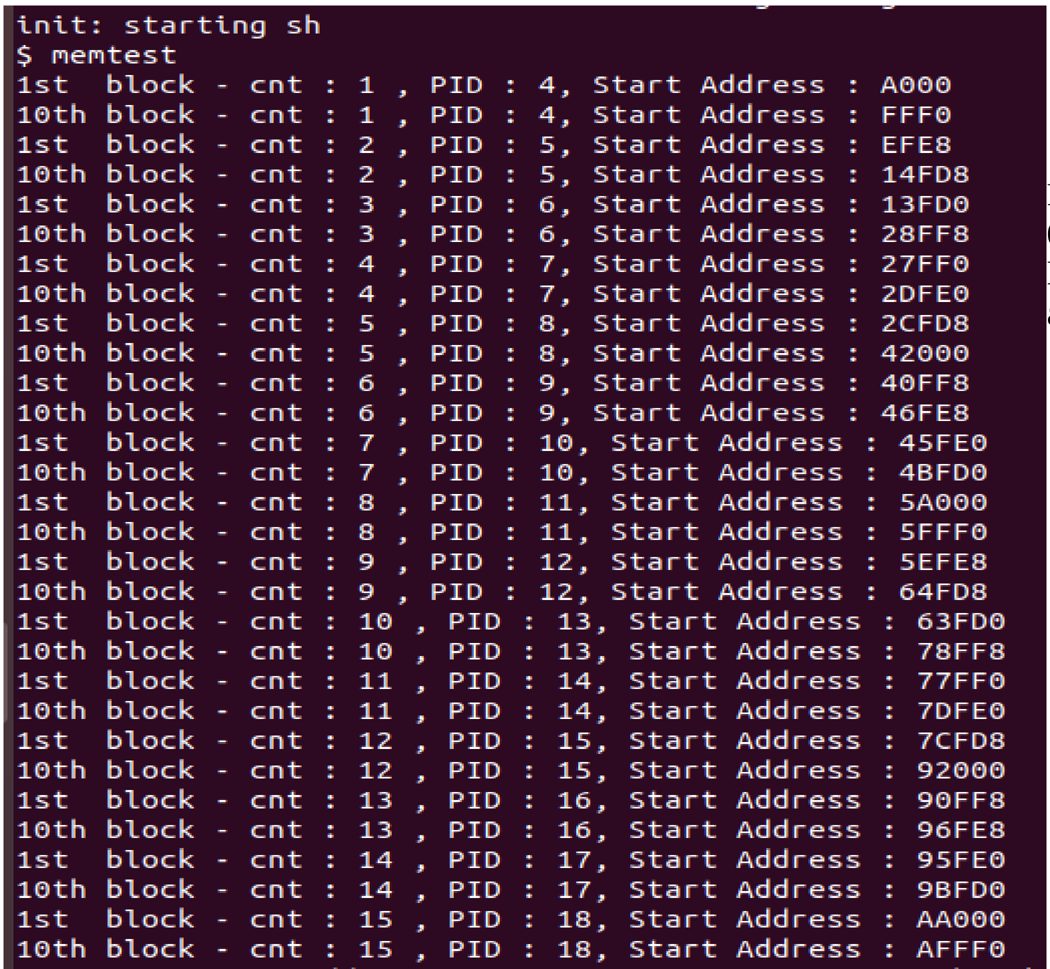
we know that as we reduce the value of PHYSTOP in memlayout.h the no of processes for which the memory is allocated will decrease which we check for our implementation.



```
init: starting sh
$ 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 is 0xE000000

No of child processes are 20



```
init: starting sh
$ 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
```

PHYSTOP is set to 0x800000
No of child processes are 15