CS344: Operating System Lab
Assignment-3

Group 24:

Name: **Dhruv Tarsadiya** Roll No: **200101123**

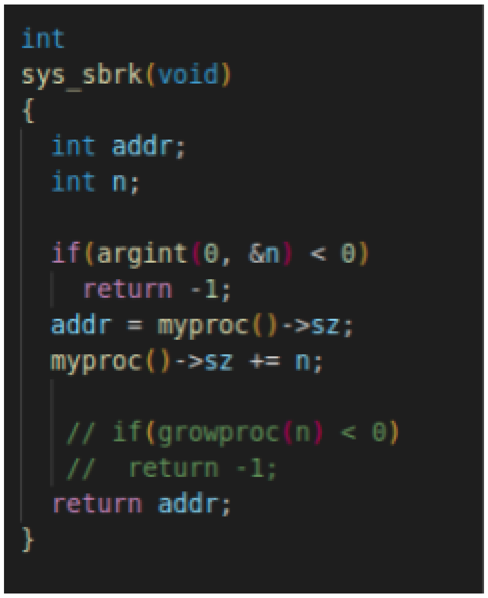
Name: **Vikram Singla** Roll No: **200101117**

Name: **Harsh Bhakkad** Roll No: **200101040**

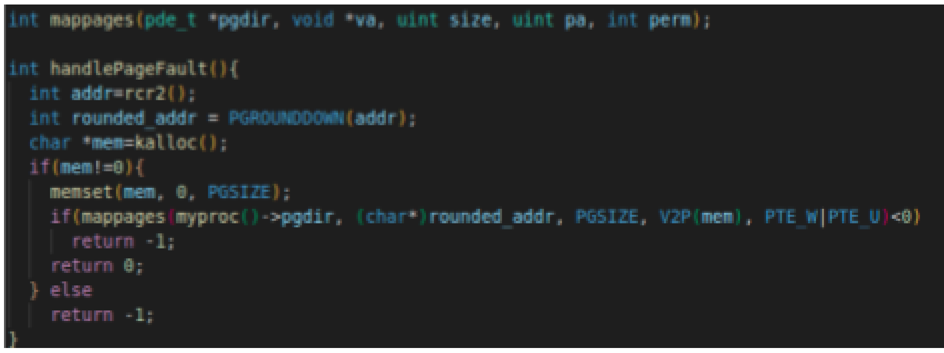
Part A

sbrk() system call is used when a process needs extra memory than its allocated value. It in turn calls growproc() which further calls allocuvm() which allocates extra pages and maps virtual addresses to physical addresses inside page tables. If we comment out the growproc() , we generate an error upon running a command say: '$ echo hi'. It occurs due the fact that in the sbrk() system call we have removed allocation code but the process thinks that it already has that page after making that system call, hence tries to access it resulting in a page fault which raises interrupt T_PGFLT.

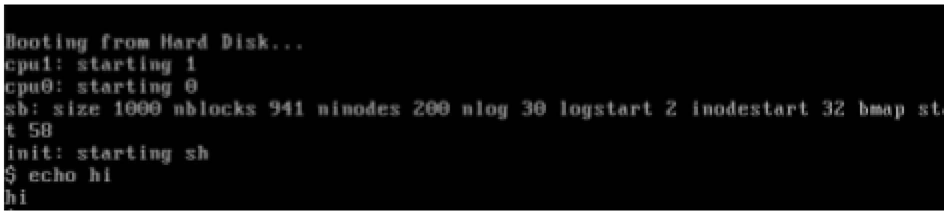
So now, we implement **Lazy Allocation** i.e. we do not give memory as soon as it is requested, but rather we give memory when it is accessed. The trap generated due to this as mentioned earlier is handled in trap.c by calling handlePageFault(); a function that we implemented. The code used to handle page fault is similar to that in allocuvm() function in vm.c. First, we allocate a frame to the process which causes a page fault, then we map the virtual address (we get that from rcr2()) where the page fault occurs, to the physical address of the allocated frame by using mappages() implemented in xv6. On successful mapping the process is restarted from the point where page fault occurred. Thus, now if we use the command '$ echo hi' there is no exception.



```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;
}
```



```c
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

int handlePageFault(){
  int addr=rcr2();
  int rounded_addr = PGROUNDDOWN(addr);
  char *mem=kalloc();
  if(mem!=0){
    memset(mem, 0, PGSIZE);
    if(mappages(myproc()->pgdir, (char*)rounded_addr, PGSIZE, V2P(mem), PTE_W|PTE_U)<0)
      return -1;
    return 0;
  } else
    return -1;
}
```

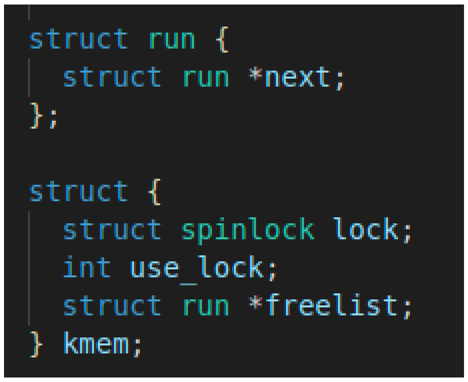


```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st
t 58
init: starting sh
$ echo hi
hi
```

Part B

The answers to the questions mentioned in the assignment are written below in order:

1. The unused pages are maintained as a linked list of free pages in struct kmem in kalloc.c



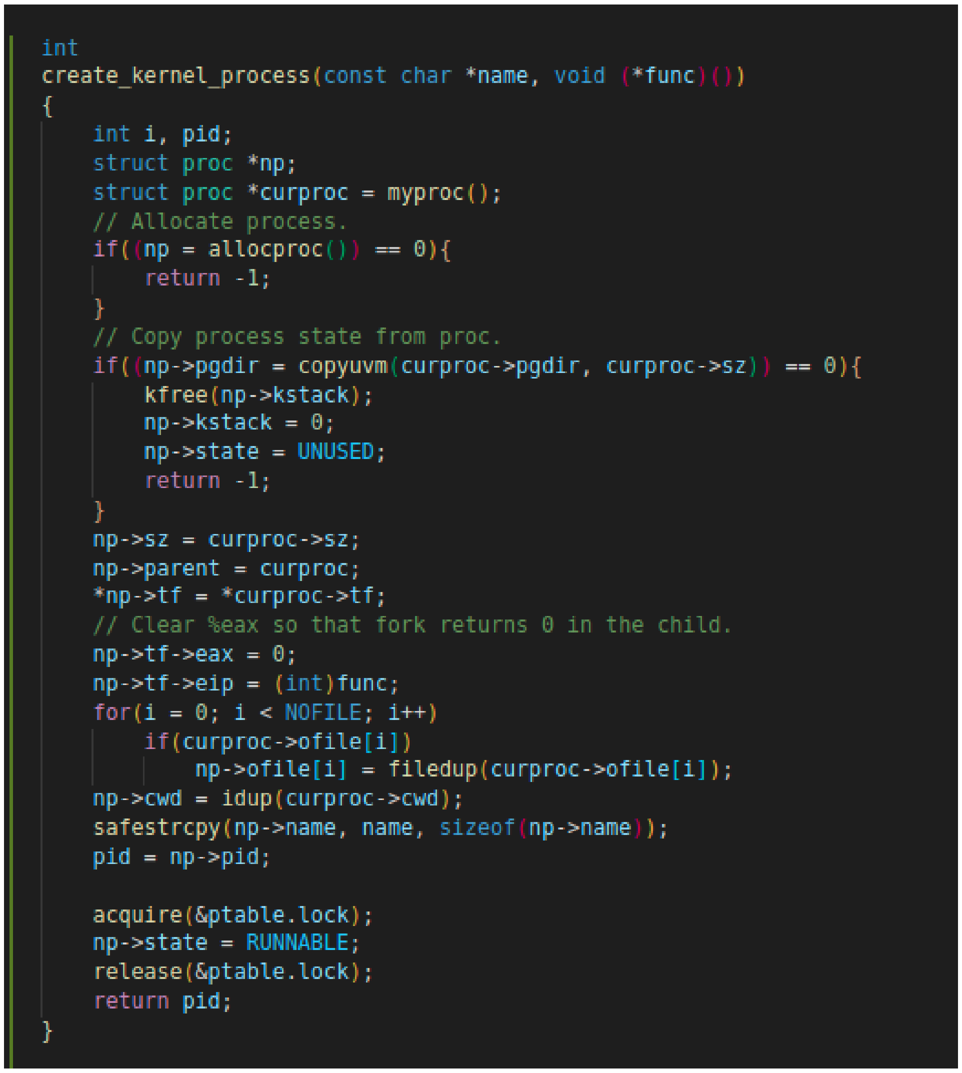
```c
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

2. A linked list struct run *freelist is used to maintain these.
3. These reside in struct kmem in kalloc.c which signifies that this list is in kernel memory.

4. Yes, xv6 memory mechanism restricts the number of user processes to NPROC which is defined as 64 in param.h.
5. After booting, the first process that runs is initproc and then sh is actually forked from it. Also, a single process can take up all the physical memory (PHYSTOP). Hence, the minimum number of processes xv6 can have is 1.

Task 1: Creating Kernel Processes:

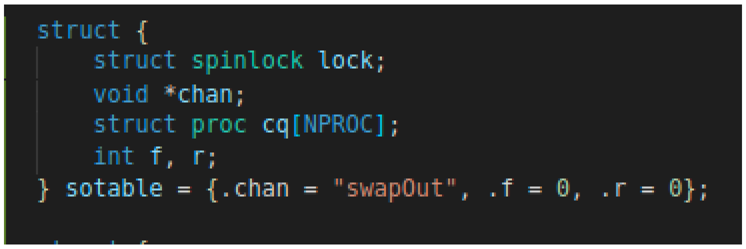
We defined the function int create kernel process(const char*, void (*entrypoint)()) in proc.c. We have reused the code from the fork() system call. First we allocate a process by using allocproc(). For this new process we copy the state of the calling process (initproc in this case) and copy the trap frame of the parent into this child process. We set the %eax register of tf to 0 so that the process receives 0 when this system call returns in the child process's space, and we also set the %eip register to the entrypoint passed in the function parameter. We do not need to set these trapframe values; since the process is to remain in the kernel.



```
int
create_kernel_process(const char *name, void (*func)())
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();
    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }
    // Copy process state from proc.
    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;
    // Clear %eax so that fork returns 0 in the child.
    np->tf->eax = 0;
    np->tf->eip = (int)func;
    for(i = 0; i < NOFILE; i++)
        if(curproc->ofile[i])
            np->ofile[i] = filedup(curproc->ofile[i]);
    np->cwd = idup(curproc->cwd);
    safestrcpy(np->name, name, sizeof(np->name));
    pid = np->pid;

    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);
    return pid;
}
```

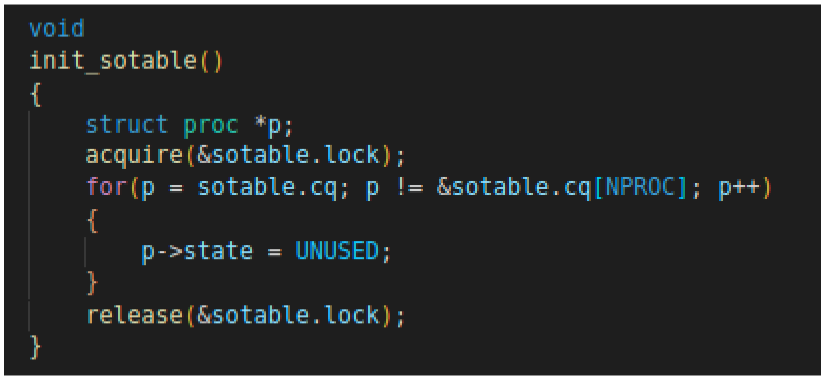
Task 2: Implementing Swapping Out Mechanism:

For implementing swapping out mechanism, we first maintain a circular queue called sotable with a spinlock and front and rear pointers f,r respectively with a channel as well for waiting processes to sleep on. Note that we have implemented the queue in such a way that one location will be empty in the queue when the queue gets full, i.e, the states f == r signifies empty queue, and (r+1)%size == f signifies full queue.



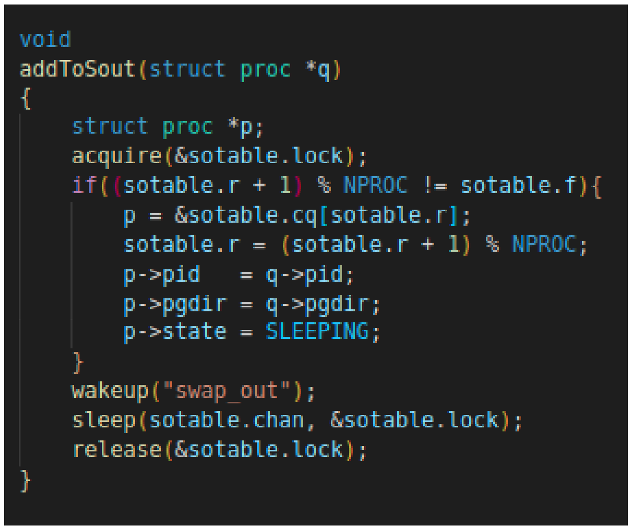
```c
struct {
    struct spinlock lock;
    void *chan;
    struct proc cq[NPROC];
    int f, r;
} sotable = {.chan = "swapOut", .f = 0, .r = 0};
```

Then we initialize sotable using init_sotable() and mark every process in queue as UNUSED.



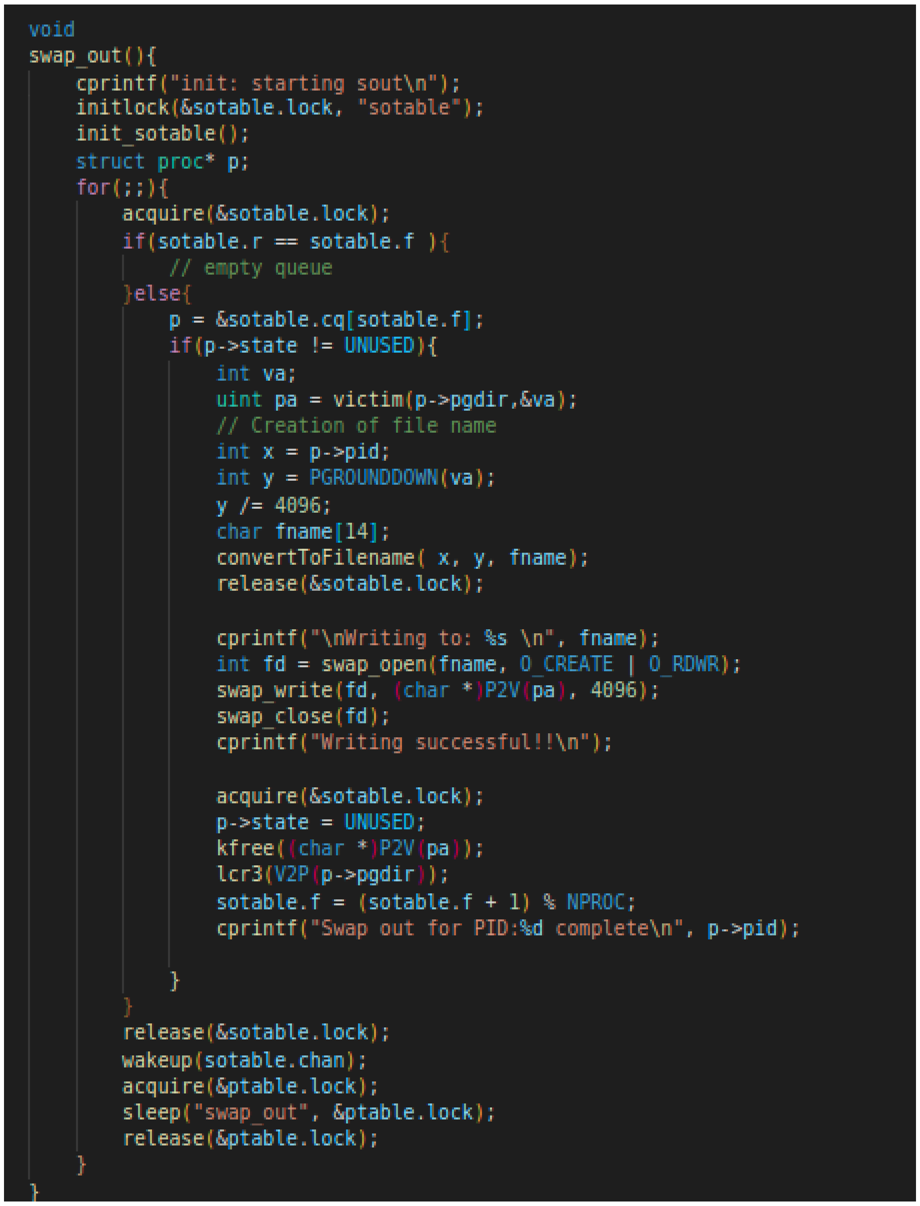
```c
void
init_sotable()
{
    struct proc *p;
    acquire(&sotable.lock);
    for(p = sotable.cq; p != &sotable.cq[NPROC]; p++)
    {
        p->state = UNUSED;
    }
    release(&sotable.lock);
}
```

We define the function addToSout() to add a swap out request. If the queue is not full we add the request and wake up the swapping out process which by implementation would be sleeping on the "swap out" channel. After making the request, process is made to sleep on the channel of sotable until it is woken up by the swapping out process, and it finally releases the lock on sotable.



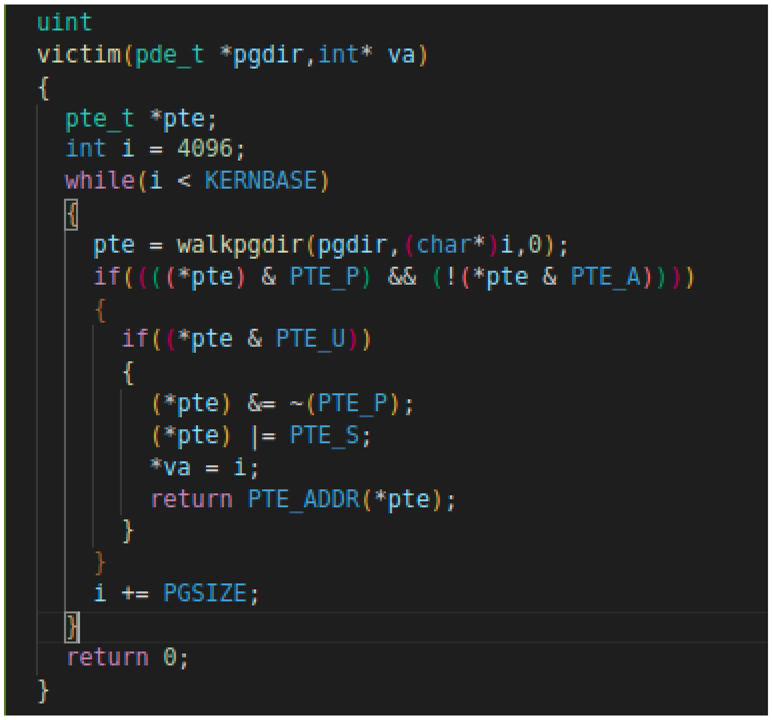
```c
void
addToSout(struct proc *q)
{
    struct proc *p;
    acquire(&sotable.lock);
    if((sotable.r + 1) % NPROC != sotable.f){
        p = &sotable.cq[sotable.r];
        sotable.r = (sotable.r + 1) % NPROC;
        p->pid   = q->pid;
        p->pgdir = q->pgdir;
        p->state = SLEEPING;
    }
    wakeup("swap_out");
    sleep(sotable.chan, &sotable.lock);
    release(&sotable.lock);
}
```

Then we define the function swap out() from where the swapping out process begins to execute. First, we initialise the spinlock on sotable, and then call init sotable() to initialise the queue. Since this process has to remain through out, we define the instructions within infinite loop. First, we acquire lock on sotable to remove a request from it and process it. If queue is empty then we make this process release the lock and go to sleep. Else, we remove a process from the position pointed to by the front pointer f and check if it is UNUSED which it would be, otherwise the queue would have been full.



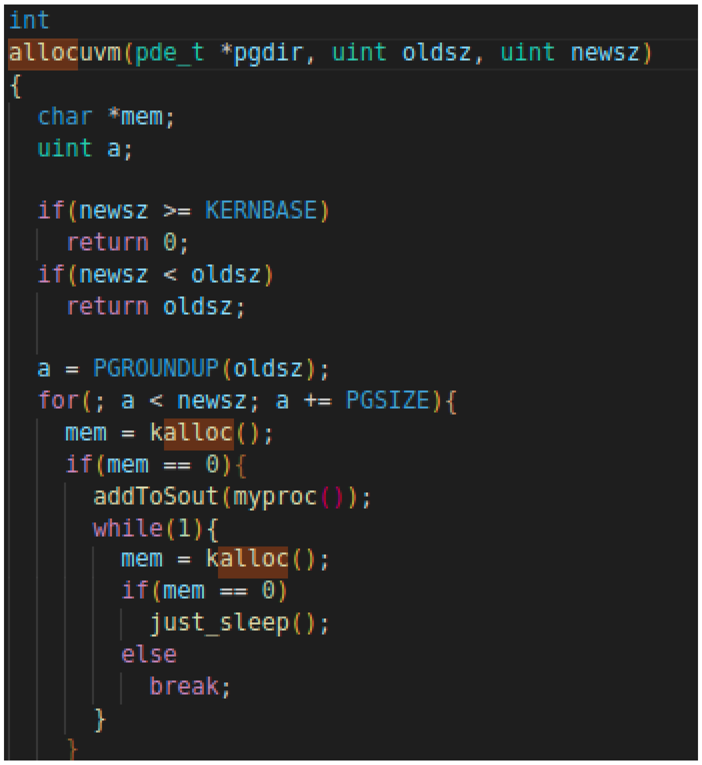
```
void
swap_out(){
    cprintf("init: starting sout\n");
    initlock(&sotable.lock, "sotable");
    init_sotable();
    struct proc* p;
    for(;;){
        acquire(&sotable.lock);
        if(sotable.r == sotable.f ){
            // empty queue
        }else{
            p = &sotable.cq[sotable.f];
            if(p->state != UNUSED){
                int va;
                uint pa = victim(p->pgdir,&va);
                // Creation of file name
                int x = p->pid;
                int y = PGROUNDDOWN(va);
                y /= 4096;
                char fname[14];
                convertToFilename( x, y, fname);
                release(&sotable.lock);

                cprintf("\nWriting to: %s \n", fname);
                int fd = swap_open(fname, O_CREATE | O_RDWR);
                swap_write(fd, (char *)P2V(pa), 4096);
                swap_close(fd);
                cprintf("Writing successful!!\n");

                acquire(&sotable.lock);
                p->state = UNUSED;
                kfree((char *)P2V(pa));
                lcr3(V2P(p->pgdir));
                sotable.f = (sotable.f + 1) % NPROC;
                cprintf("Swap out for PID:%d complete\n", p->pid);

            }
        }
        release(&sotable.lock);
        wakeup(sotable.chan);
        acquire(&ptable.lock);
        sleep("swap_out", &ptable.lock);
        release(&ptable.lock);
    }
}
```

Then we select a victim from present process's space (i.e. we are following fixed frame allocation) using victim() function defined in vm.c. This function takes the present process’s page directory and a pointer to store the virtual address of the page that is getting evicted. While traversing the table, in each entry of the directory we check if its PTE P flag (present bit) is set or not. Even if it is set we further check if its PTE A (0x020) flag (accessed) bit is set or not which signifies whether or not it was accessed after it was rescheduled or scheduled. x86 hardware resets PTE A flag for all the page table entries whenever control switches from scheduler to this process. Thus, choosing a page that is not accessed and is present is the best choice for LRU replacement policy as context switching is frequent. Then, we make this page as a victim and put down the PTE P flag and mark its table entry to be swapped (using PTE S flag). All these flags have been defined in mmu.h.



```
uint
victim(pde_t *pgdir,int* va)
{
  pte_t *pte;
  int i = 4096;
  while(i < KERNBASE)
  {
    pte = walkpgdir(pgdir,(char*)i,0);
    if((((*pte) & PTE_P) && (!(*pte & PTE_A))))
    {
      if((*pte & PTE_U))
      {
        (*pte) &= ~(PTE_P);
        (*pte) |= PTE_S;
        *va = i;
        return PTE_ADDR(*pte);
      }
    }
    i += PGSIZE;
  }
  return 0;
}
```

After finding the victim, we make a fname string, with pid and the virtual address of the page that is replaced, to create a file with this name to contain the contents of evicted block. While doing the file management we release the lock on sotable. After writing to the file is complete then we re-acquire the lock on sotable, make the entry in the queue to be UNUSED and increment the f pointer. When finally the queue is empty we release the lock on the sotable and make this swap out() process to sleep on channel "swap out" and wakeup other processes that might be sleeping on the channel of sotable.

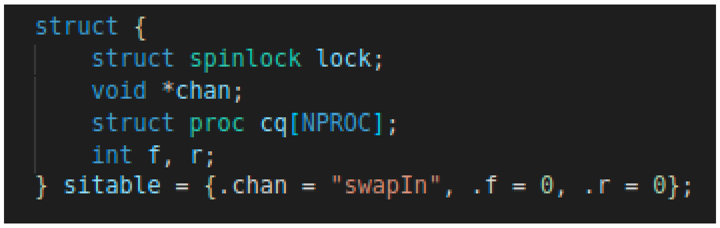


```c
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      addToSout(myproc());
      while(1){
        mem = kalloc();
        if(mem == 0)
          just_sleep();
        else
          break;
      }
    }
```

A request to swap out is made when allocuvm() in vm.c runs out of frames in main memory. This is because a process requesting to increase its memory size would call srbk() system call, which in turn calls growproc() in proc.c which calls allocuvm(). When a process sends a swap out request from allocuvm() and still can not get a frame, then we make it sleep on sotable.chan until swap out() makes a wakeup call on sotable.chan and this sleeping makes the request again.

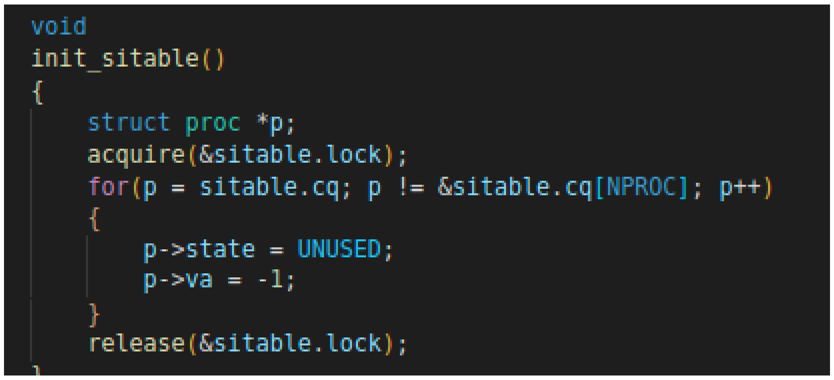
Task 3: Implementing Swapping In Mechanism:

The initial part of this task is very similar to Task 2, and the request is maintained using a circular queue sitable.



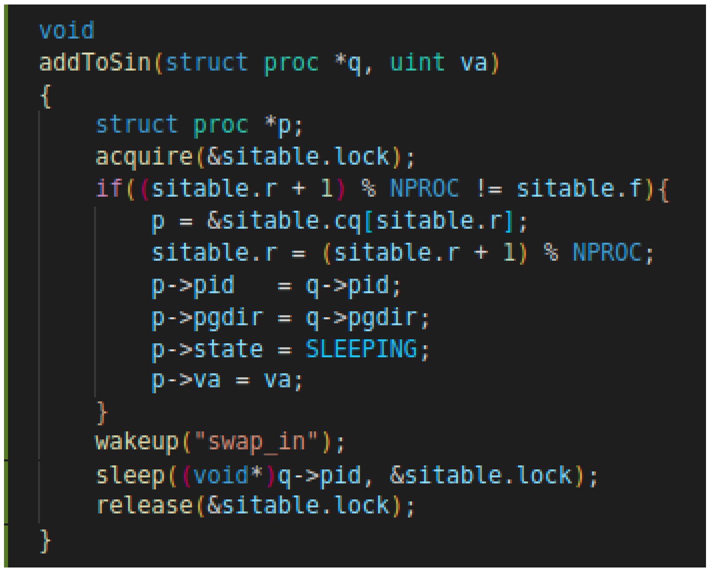
```c
struct {
    struct spinlock lock;
    void *chan;
    struct proc cq[NPROC];
    int f, r;
} sitable = {.chan = "swapIn", .f = 0, .r = 0};
```

A thing to note here is that a swap in request will have two parts - the process pid and the virtual address of the page swapped, for this we add an attribute va to struct proc to maintain the virtual address of the page swapped. Initialization is done keeping this in mind.



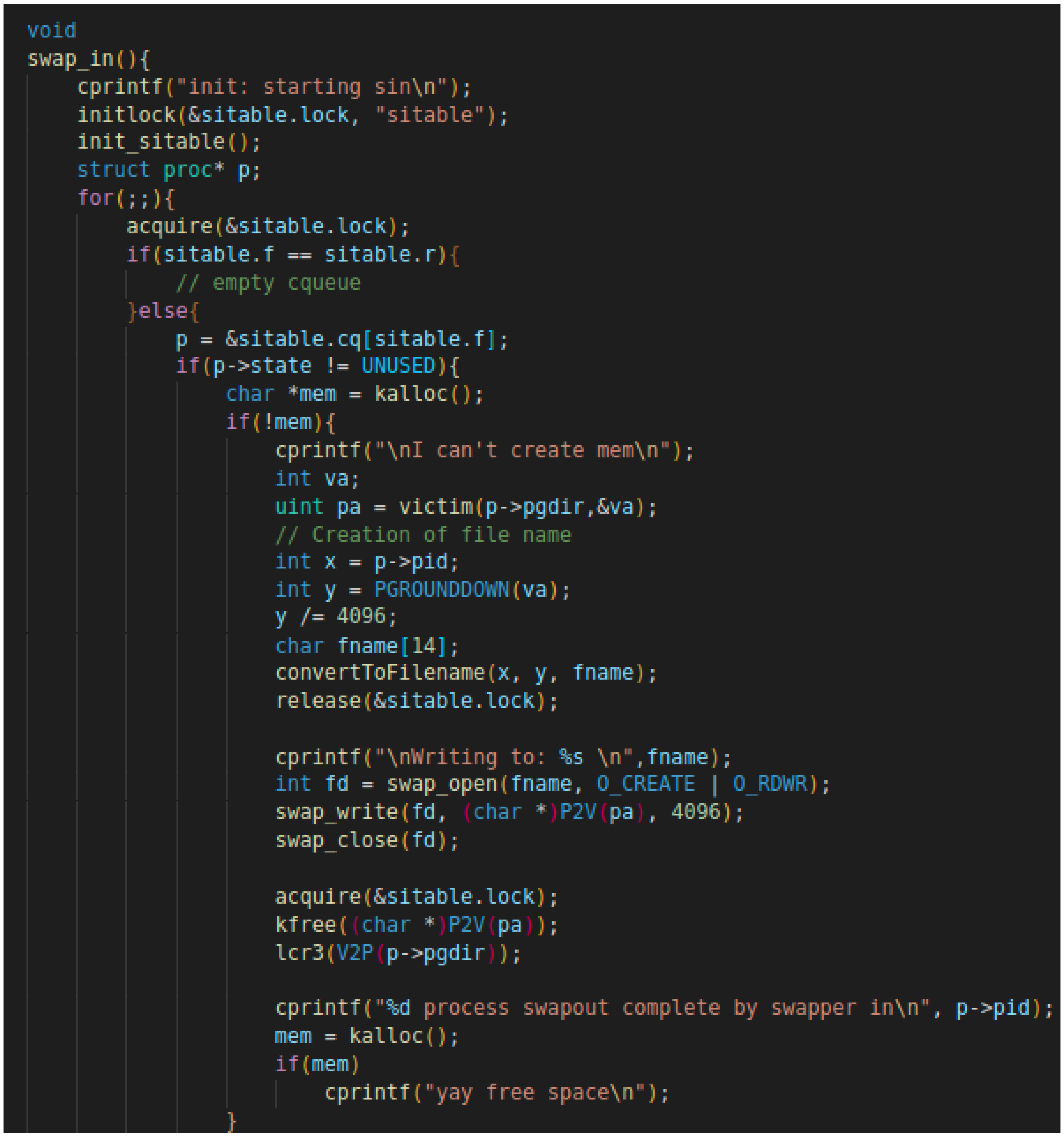
```
void
init_sitable()
{
    struct proc *p;
    acquire(&sitable.lock);
    for(p = sitable.cq; p != &sitable.cq[NPROC]; p++)
    {
        p->state = UNUSED;
        p->va = -1;
    }
    release(&sitable.lock);
```

To add a swap-in request, we handle it similar to addToSout(). See below:

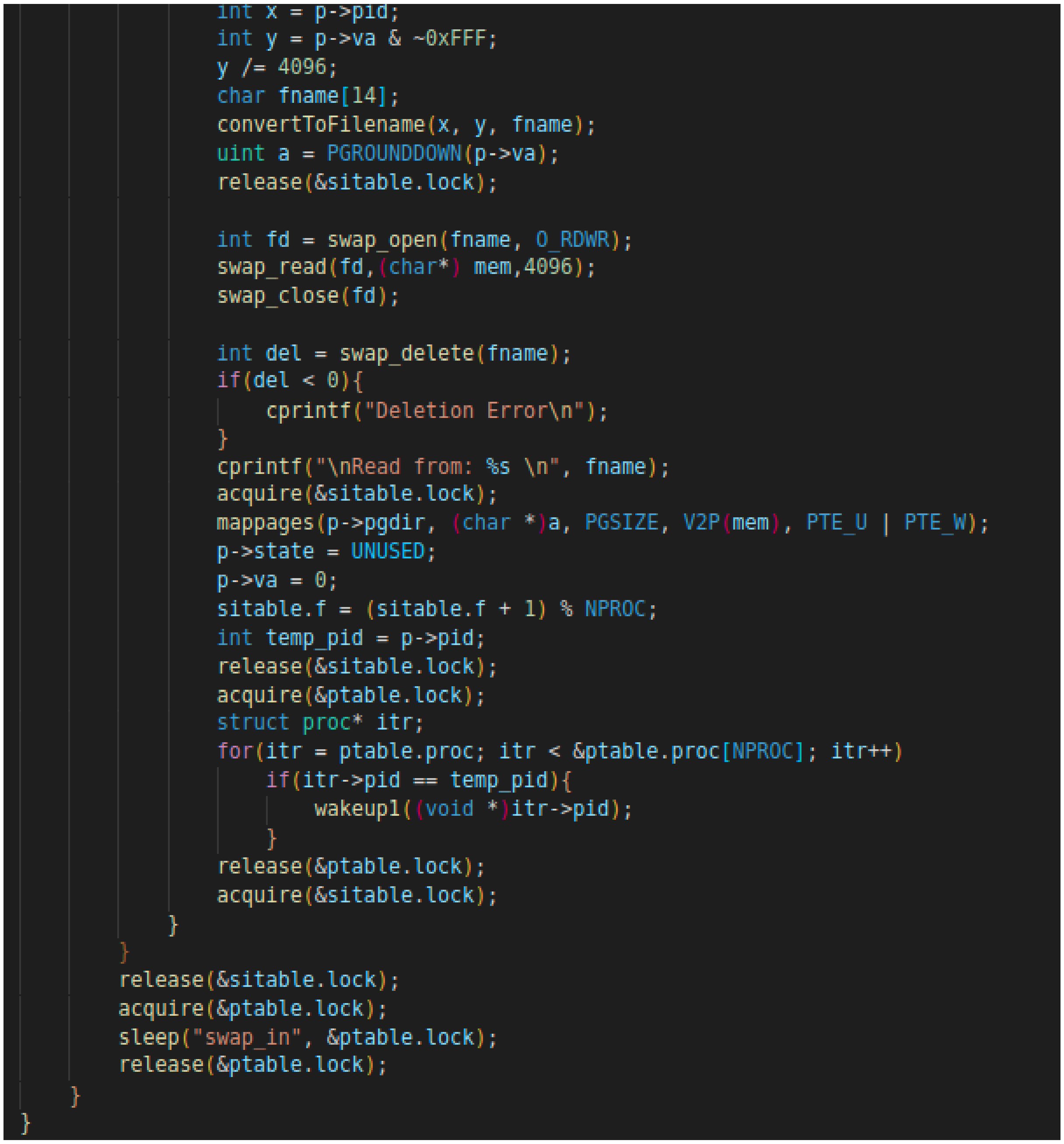


```
void
addToSin(struct proc *q, uint va)
{
    struct proc *p;
    acquire(&sitable.lock);
    if((sitable.r + 1) % NPROC != sitable.f){
        p = &sitable.cq[sitable.r];
        sitable.r = (sitable.r + 1) % NPROC;
        p->pid   = q->pid;
        p->pgdir = q->pgdir;
        p->state = SLEEPING;
        p->va = va;
    }
    wakeup("swap_in");
    sleep((void*)q->pid, &sitable.lock);
    release(&sitable.lock);
}
```

Now, we define the function swap_in() from where the process for swapping-in mechanism starts execution. Initial steps are similar to swap out().

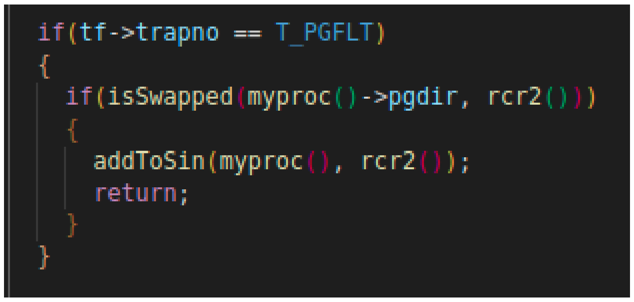


```
void
swap_in(){
    cprintf("init: starting sin\n");
    initlock(&sitable.lock, "sitable");
    init_sitable();
    struct proc* p;
    for(;;){
        acquire(&sitable.lock);
        if(sitable.f == sitable.r){
            // empty cqueue
        }else{
            p = &sitable.cq[sitable.f];
            if(p->state != UNUSED){
                char *mem = kalloc();
                if(!mem){
                    cprintf("\nI can't create mem\n");
                    int va;
                    uint pa = victim(p->pgdir,&va);
                    // Creation of file name
                    int x = p->pid;
                    int y = PGROUNDDOWN(va);
                    y /= 4096;
                    char fname[14];
                    convertToFilename(x, y, fname);
                    release(&sitable.lock);

                    cprintf("\nWriting to: %s \n",fname);
                    int fd = swap_open(fname, O_CREATE | O_RDWR);
                    swap_write(fd, (char *)P2V(pa), 4096);
                    swap_close(fd);

                    acquire(&sitable.lock);
                    kfree((char *)P2V(pa));
                    lcr3(V2P(p->pgdir));

                    cprintf("%d process swapout complete by swapper in\n", p->pid);
                    mem = kalloc();
                    if(mem)
                        cprintf("yay free space\n");
                }
```



```
                int x = p->pid;
                int y = p->va & ~0xFFF;
                y /= 4096;
                char fname[14];
                convertToFilename(x, y, fname);
                uint a = PGROUNDDOWN(p->va);
                release(&sitable.lock);

                int fd = swap_open(fname, O_RDWR);
                swap_read(fd,(char*) mem,4096);
                swap_close(fd);

                int del = swap_delete(fname);
                if(del < 0){
                    cprintf("Deletion Error\n");
                }
                cprintf("\nRead from: %s \n", fname);
                acquire(&sitable.lock);
                mappages(p->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_U | PTE_W);
                p->state = UNUSED;
                p->va = 0;
                sitable.f = (sitable.f + 1) % NPROC;
                int temp_pid = p->pid;
                release(&sitable.lock);
                acquire(&ptable.lock);
                struct proc* itr;
                for(itr = ptable.proc; itr < &ptable.proc[NPROC]; itr++)
                    if(itr->pid == temp_pid){
                        wakeup1((void *)itr->pid);
                    }
                release(&ptable.lock);
                acquire(&sitable.lock);
            }
        }
        release(&sitable.lock);
        acquire(&ptable.lock);
        sleep("swap_in", &ptable.lock);
        release(&ptable.lock);
    }
}
```

After initialization is done, within the infinite loop we remove the request pointed to by the front pointer if the queue is not empty. If it is empty we release the lock and wakeup the process that might be sleeping on the sitable.chan and make this process sleep. Else the extracted process requests a memory frame to write the content of the files to. If it is not available then we swap out a page from requesting process's address space similar to swap out(). Then we read from the file and delete the file so that the other files can be accommodated in its block. Then we map the frame to which we had written the contents of the file to the virtual address which is used to open the corresponding file. Then we wakeup this process and continue dealing with the other requests. Once the request queue gets empty we make this kernel process sleep on channel "swap in" and wakeup the processes sleeping on sitable.chan.

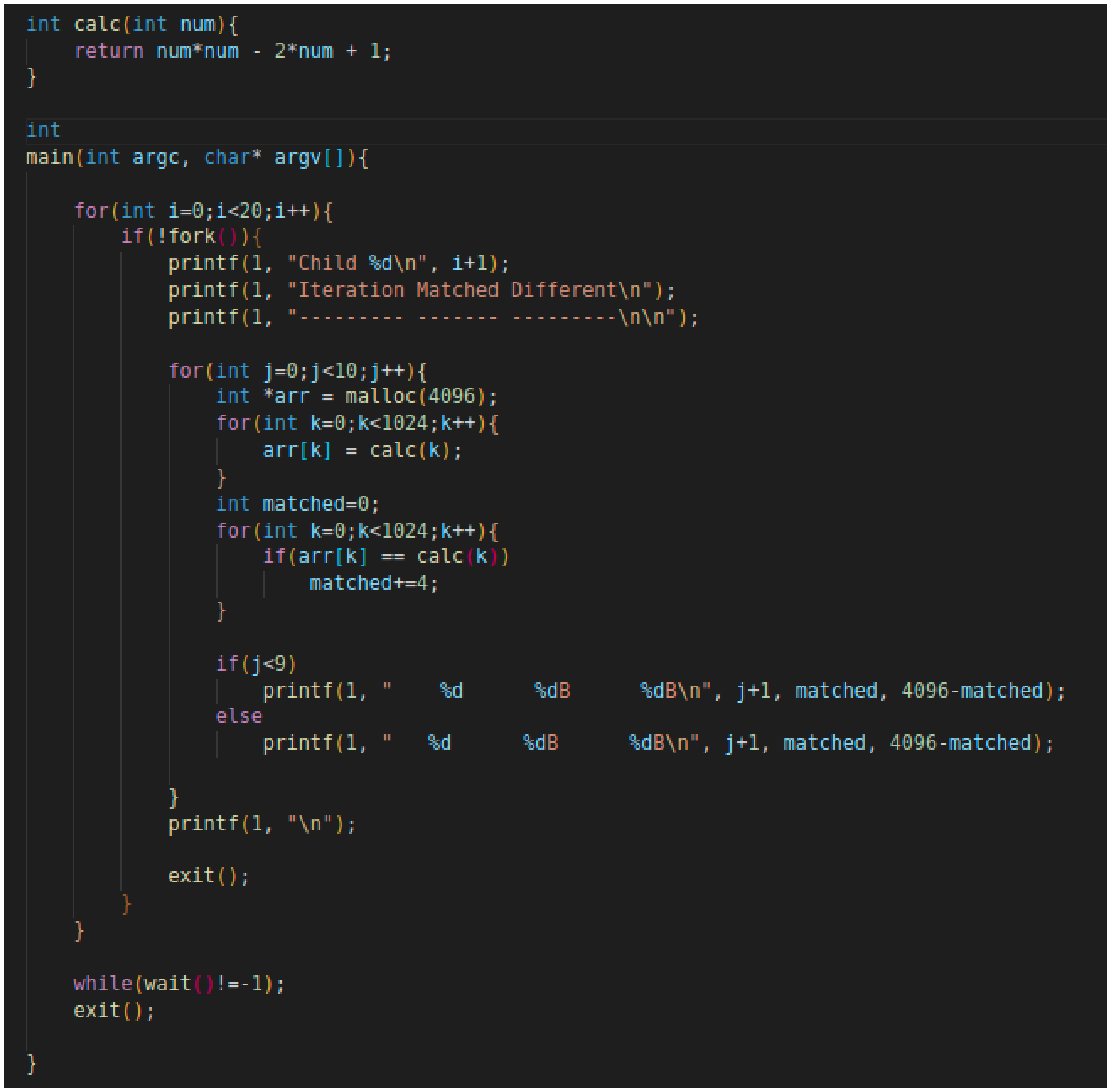


```
if(tf->trapno == T_PGFLT)
{
  if(isSwapped(myproc()->pgdir, rcr2()))
  {
    addToSin(myproc(), rcr2());
    return;
  }
}
```

Now, whenever a page fault occurs we handle that page fault to check if it is a swapped page or not. If it is then submit a swap in request else capture the trap. For this we add the above in trap() in trap.c.

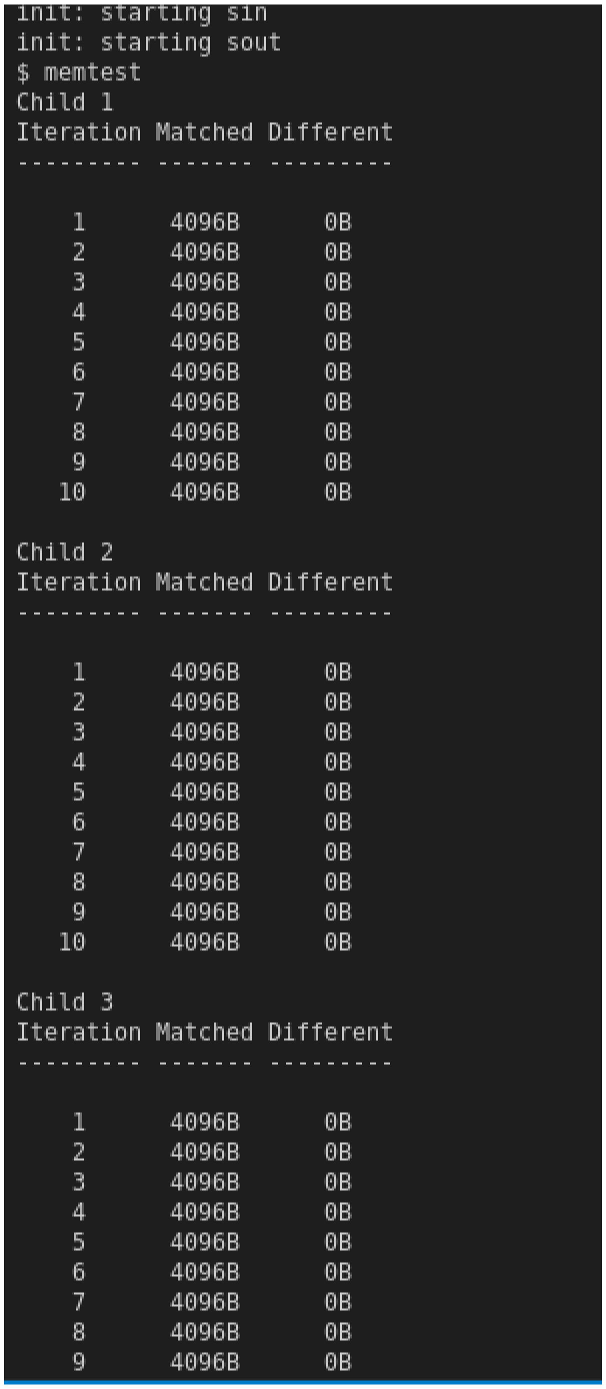
Task 4: Sanity test:

First, the methodology, we implemented the create kernel process(), swap in(), swap out() as system calls. In init.c we called the first system call twice, with addresses of the other two system call functions in user space, to create two kernel processes. So, after init process forks and execs into sh, it creates the other two kernel process (using a system call) that remain in memory as long as xv6 is running just like init/sh. We reduced the PHYSTOP from 0xE000000 (224 MB) to 0x0400000 (4 MB) which is the minimum memory required by xv6 to run kinit1() in kalloc.c.

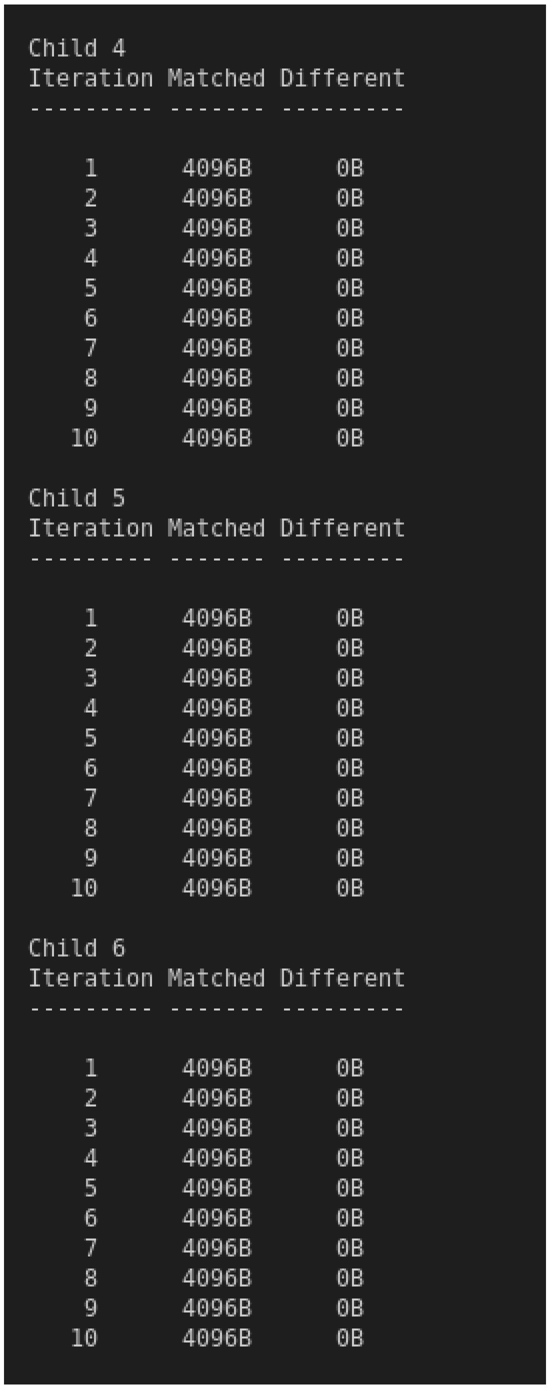


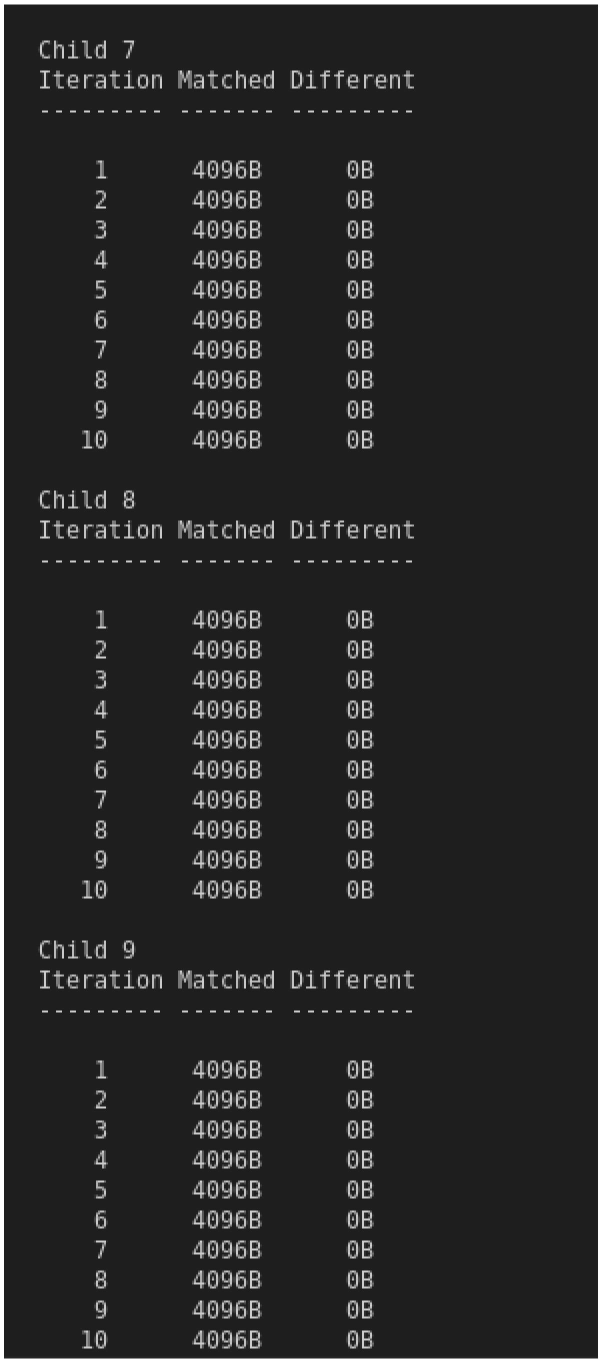
```c
int calc(int num){
    return num*num - 2*num + 1;
}

int
main(int argc, char* argv[]){

    for(int i=0;i<20;i++){
        if(!fork()){
            printf(1, "Child %d\n", i+1);
            printf(1, "Iteration Matched Different\n");
            printf(1, "--------- ------- ---------\n\n");

            for(int j=0;j<10;j++){
                int *arr = malloc(4096);
                for(int k=0;k<1024;k++){
                    arr[k] = calc(k);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    if(arr[k] == calc(k))
                        matched+=4;
                }

                if(j<9)
                    printf(1, "    %d      %dB      %dB\n", j+1, matched, 4096-matched);
                else
                    printf(1, "   %d      %dB      %dB\n", j+1, matched, 4096-matched);

            }
            printf(1, "\n");

            exit();
        }
    }

    while(wait()!=-1);
    exit();

}
```

The main process forks 20 child processes using the fork() system call. Each process executes a loop in 10 iterations. In each iteration, 4096 B memory is allocated using malloc. The value stored at index i of the array is given by the calc() function. A counter called matched is used to store the number of bytes which are stored correctly. Whereas the Different counter is used to store the values which are stored differently (should be none if our implementation is correct). We make relevant changes in the Makefile and thereby run memtest.



```
init: starting sin
init: starting sout
$ memtest
Child 1
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 2
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 3
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
```


```
Child 4
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 5
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 6
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B
```



```
Child 7
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 8
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 9
Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B
```

As we can see, our implementation passes the sanity test; as all memory allocated held the same information on reading, thus all bytes matched.