CS344:Operating Systems Lab
Assignment 3
Lab Report

Group Number : 28

Tanay Maheshwari	190101092
Shreyansh Meena	190101084
Vignesh Ravichandra Rao	190101109
Nikhil Kumar Pandey	190123040

Commands to reproduce changes:
The patch files PartA.patch and PartB.patch are included in the submission.
To apply the patch file, run the following command:-

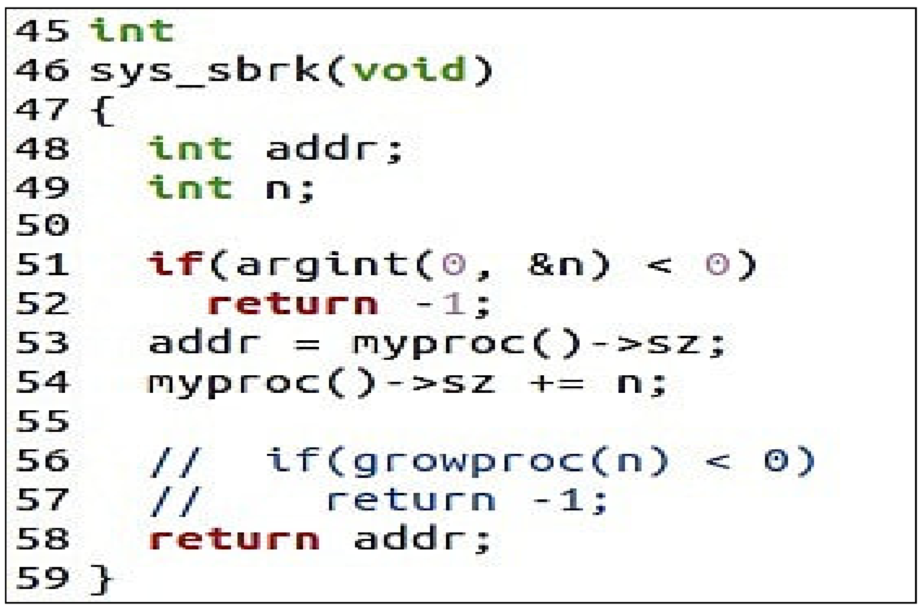
```
patch -uN -d xv6-public/ < PartA.patch
patch -uN -d xv6-public/ < PartB.patch
```

Part A) Lazy Memory Allocation

Implementation details

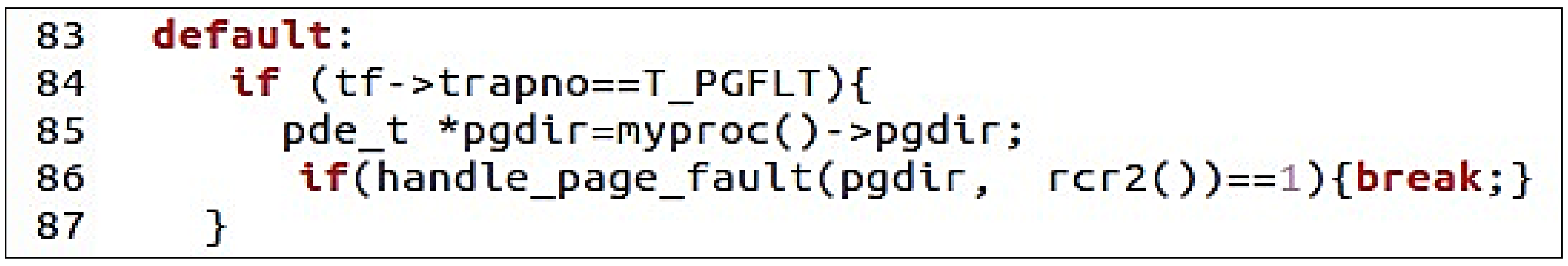
Following three files were modified for this part:

1. **Sysproc.c** : The **sbrk()** function was modified to prevent the allocation of physical memory Before it was needed. To implement lazy allocation, physical memory was allocated only when it was needed. The code in **sbrk()** which allocates the space to the process was commented.



```
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;
}
```

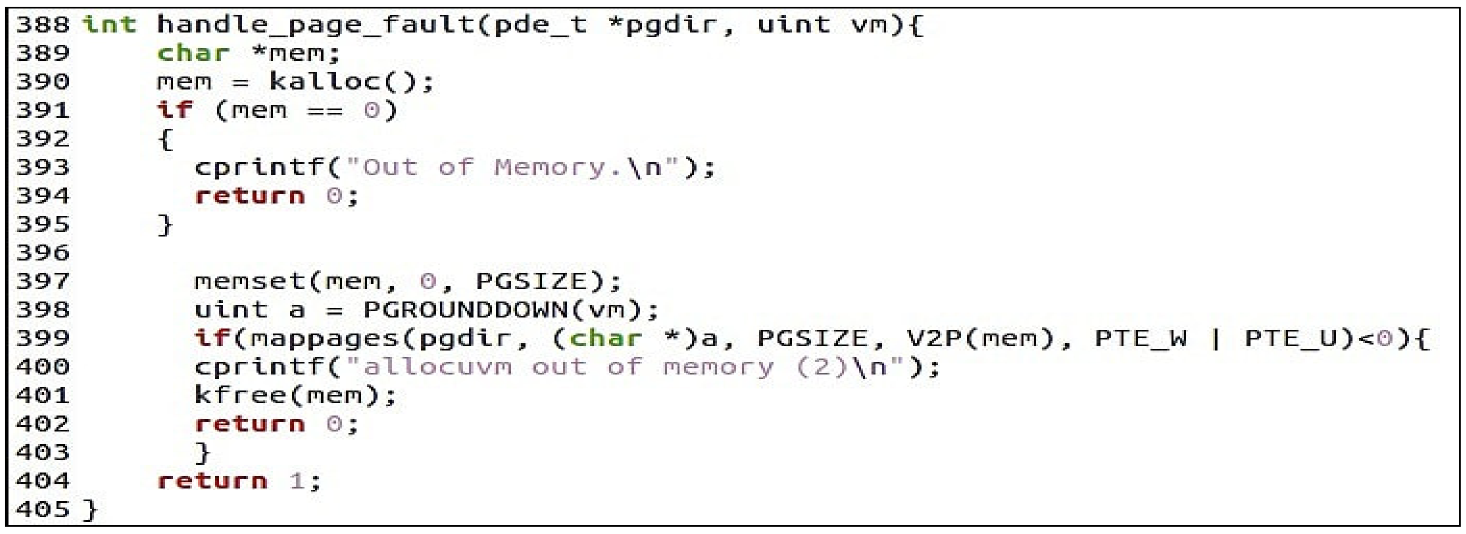
2. **Trap.c** : An if condition was added in the default case in the **trap()** function to check whether the error was caused by page fault. If we can handle the page fault successfully we break otherwise We kill the process instead of calling break.



```
  default:
    if (tf->trapno==T_PGFLT){
      pde_t *pgdir=myproc()->pgdir;
       if(handle_page_fault(pgdir,  rcr2())==1){break;}
   }
```

3. **Vm.c** : A new function handle_page_fault() was added. This function first checks if the memory is available or not if it is then it allocates one page to the process and maps the page to the virtual address.Upon successful allocation the function returns 1 otherwise it returns 0. **kalloc()** was called to Allocate the required space or to inform that no more memory is available.

⇒

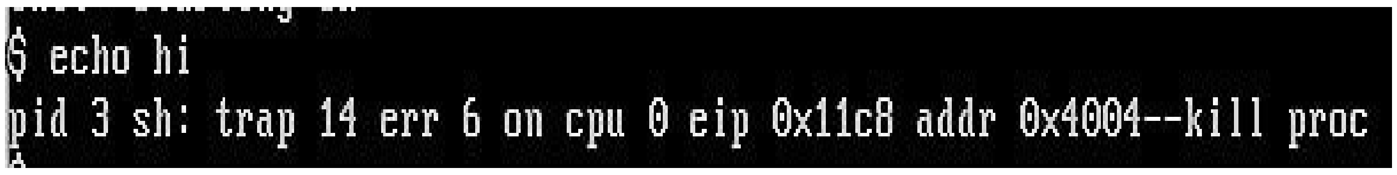


```c
int handle_page_fault(pde_t *pgdir, uint vm){
    char *mem;
    mem = kalloc();
    if (mem == 0)
    {
      cprintf("Out of Memory.\n");
      return 0;
    }

      memset(mem, 0, PGSIZE);
      uint a = PGROUNDDOWN(vm);
      if(mappages(pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W | PTE_U)<0){
      cprintf("allocuvm out of memory (2)\n");
      kfree(mem);
      return 0;
      }
    return 1;
}
```

Observations :

- Before lazy allocation was implemented, the command **"echo hi"** would cause a page fault. The page fault is caught by **trap.c** wherein it kills the process since it is accessing memory outside the process' limits.

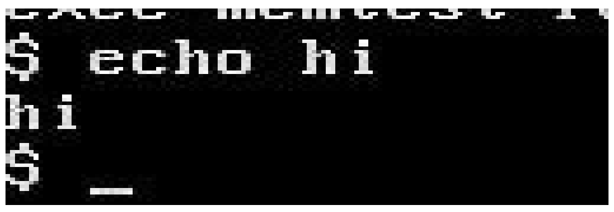
- This is because **sys_sbrk()** was manipulated only to increase the process size but not really allocate any memory Hence, **growproc()** , which makes a call to **sys_sbrk()**, assumes the size has increased without there actually being any increase in physical memory allocated to the process.



```
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x11c8 addr 0x4004--kill proc
```

fig 1 : sample output picture before implementation of handle_page_fault

- After the **handle_page_fault()** function was implemented for lazy allocation, the same **"echo hi"** command does not give a page fault trap. This is because **handle_page_fault()** makes a call to **kalloc()** to allocate a free page to the process and adds it to the page table by calling **mappages()**.



```
$ echo hi
hi
$ _
```

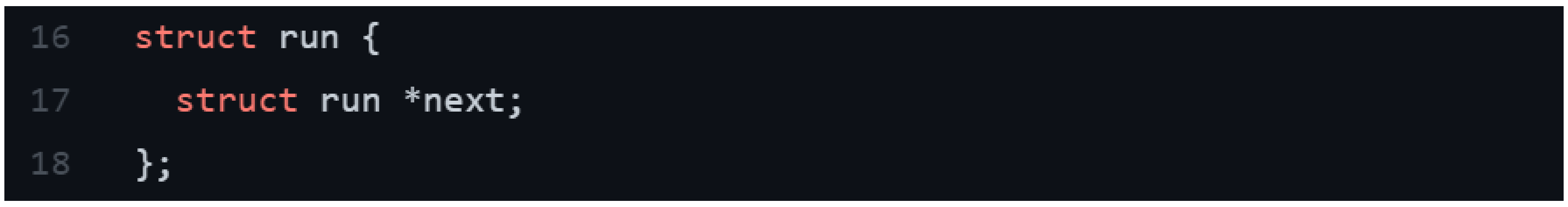
fig 2 : sample output picture after implementation of handle_page_fault

Part B)

Answers to the questions regarding memory management :

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

Ans. The kernel uses the struct run (line 16 of **kalloc.c**) data structure to point to a free page. This structure implements a linked list of free pages, where each page's next pointer points to The next available free page in memory.This structure is stored at the beginning of each free page while the rest of the page is filled with junk value. The kernel code always returns the virtual addresses in the kernel address space, and not the actual physical address. The **V2P** macro is used to translate from kernel's virtual address space to physical address space.



```
16  struct run {
17    struct run *next;
18  };
```

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

Ans. The struct run is a list of free pages maintained as a singly linked list. The pointer next is stored within the page itself. The kalloc function allocates a free page by updating the head of the linked list (called freelist) to **freelist->next;** Similarly kfree frees a page by adding the page to the front of the linked list.

Q3. Where do these reside?

Ans. Each free page's run structure is stored within the page itself and the no Additional space from within the kernel's address space is used to keep track of free pages.

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

Ans. The **NPROC** defined in **param.h** is the maximum number of user processes that can be in the process table of xv6. NPROC = 64 which means that a maximum of 64 process can be present in the memory of xv6 at a time. These processes can be in different states such as RUNNABLE, SLEEPING, ZOMBIE etc. At a given time, only one process is RUNNING.

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

Ans. The lowest number of processes xv6 can have at the same time is one process.

Implementation details

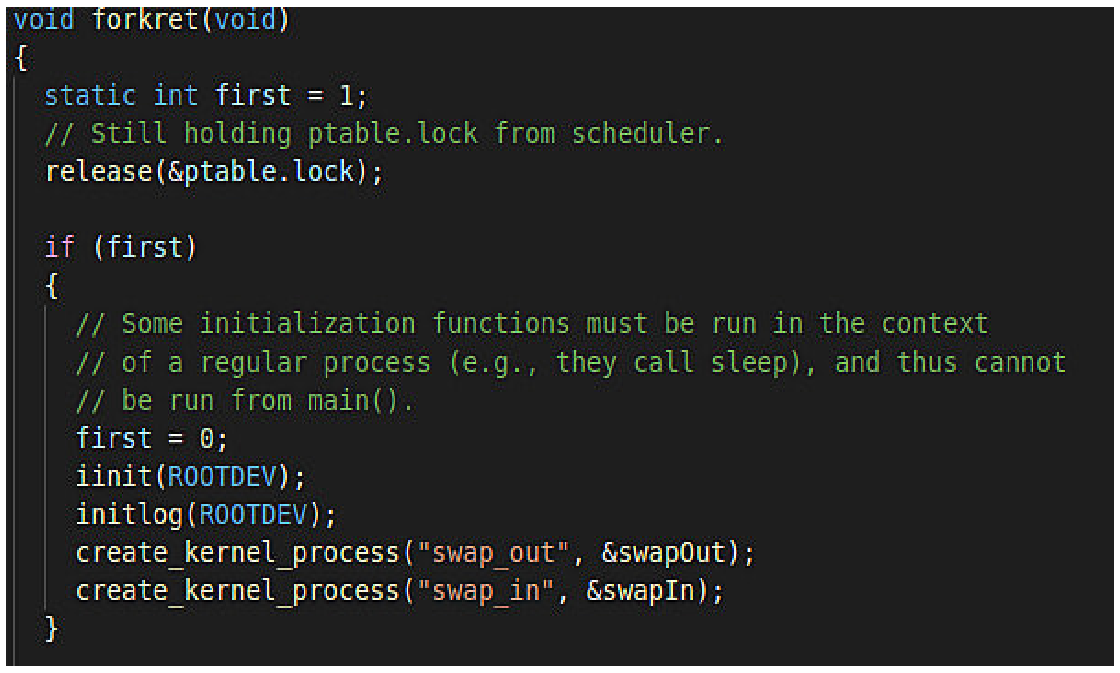
Task 1:Kernel Processes

Following files were modified for this task :

1. Proc.c : A function **create_kernel_process()** is added which is a combination of **initproc()**, **fork()** and **allocproc()**.

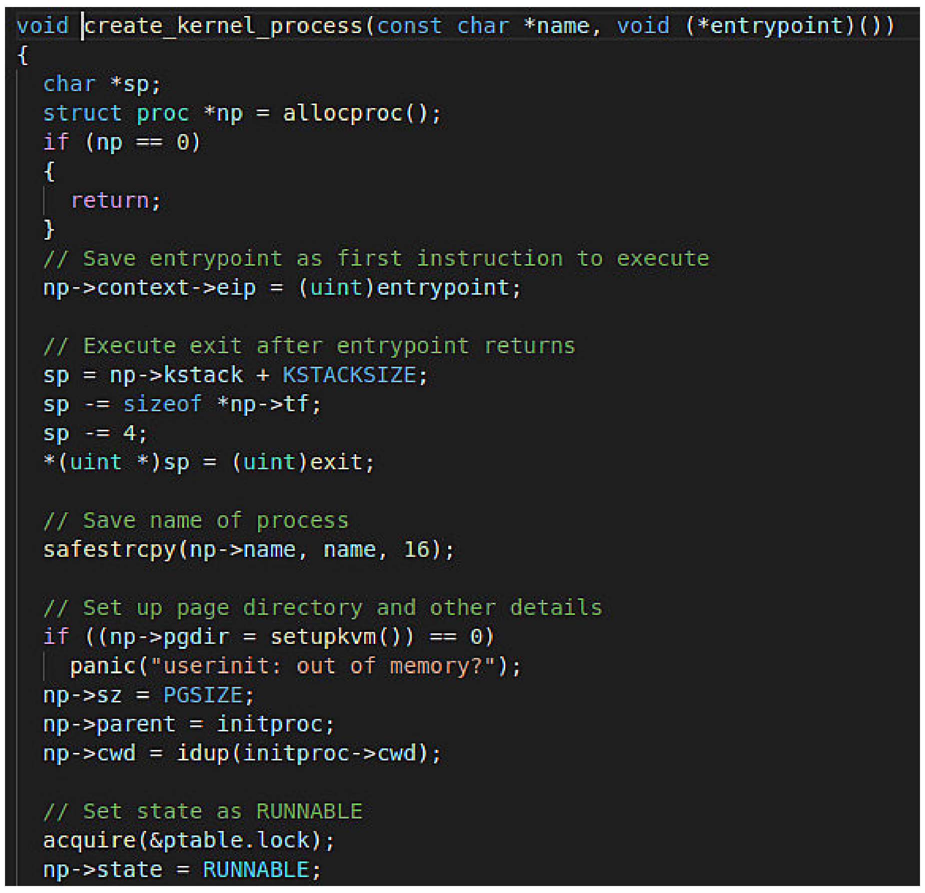
- Firstly allocproc is called to assign pid and get process in the ptable then kernel stack is created for process using kalloc.
- In kstack instruction pointer context->eip is set to entrypoint so that it executes from the first instruction of entry point.
- In kstack **trapret** is set to exit so that process exits after entrypoint returns.

- **swapOut()** and **swapIn()** are the two kernel processes we need They were created in the **forkret** function.
- When **forkret** is called from initproc and when it runs for the first time, it creates the kernel processes.



```c
void forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first)
  {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
    create_kernel_process("swap_out", &swapOut);
    create_kernel_process("swap_in", &swapIn);
  }
```

fig 2 : forkret()



```c
void create_kernel_process(const char *name, void (*entrypoint)())
{
  char *sp;
  struct proc *np = allocproc();
  if (np == 0)
  {
    return;
  }
  // Save entrypoint as first instruction to execute
  np->context->eip = (uint)entrypoint;

  // Execute exit after entrypoint returns
  sp = np->kstack + KSTACKSIZE;
  sp -= sizeof *np->tf;
  sp -= 4;
  *(uint *)sp = (uint)exit;

  // Save name of process
  safestrcpy(np->name, name, 16);

  // Set up page directory and other details
  if ((np->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  np->sz = PGSIZE;
  np->parent = initproc;
  np->cwd = idup(initproc->cwd);

  // Set state as RUNNABLE
  acquire(&ptable.lock);
  np->state = RUNNABLE;
```

fig 1:create_kernel_process

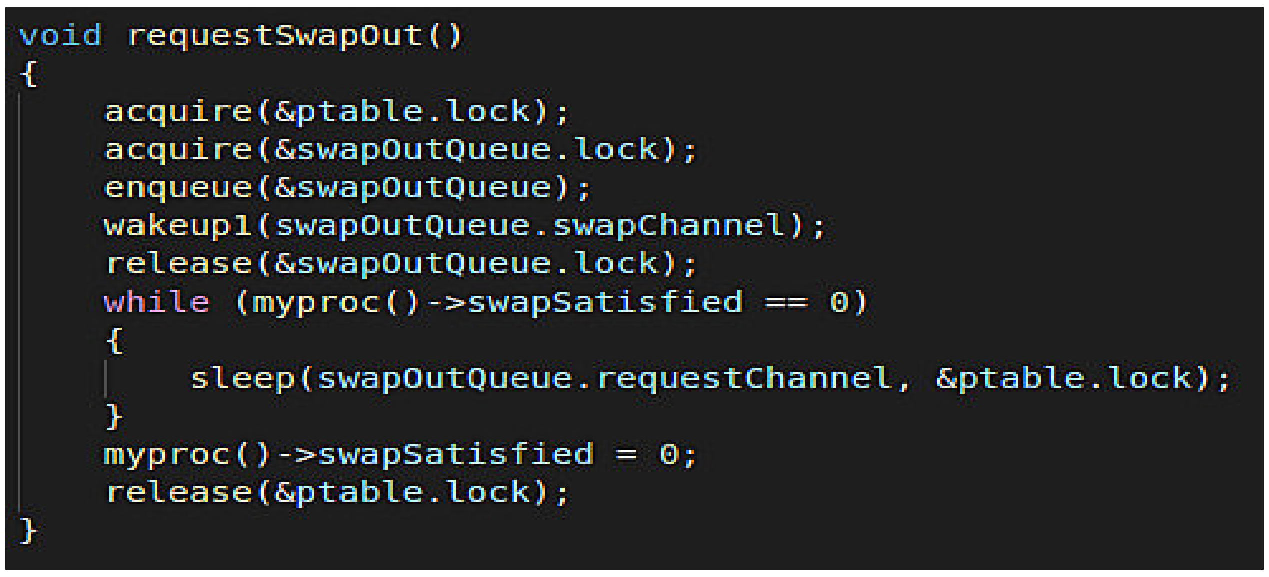
Task 2:Swapping Out

- **Paging.c** : It contains the functions required for paging and swapping.
- In struct proc two new fields were added :
  1. swapSatisfied : It is equal to 1 when swapOut request was satisfied otherwise 0.
  2. pageFaultaddress : It is used by swapIn to fetch appropriate page.

- Struct **swapRequests** was created in **paging.h** which implements circular queue for swapIn and swapOut requests.It is lock protected for mutual exclusion.
  1. enqueue() : Adds the requesting process to the head of the queue.
  2. dequeue() : Removes the requesting process from the head of the queue.
  3. gethead() : Returns the head of the queue.
  4. isEmpty() : Checks whether the queue is empty or not.

- **requestSwapOut()** : It is Called from kalloc() when running is unable to find a free page.
  When requestswapout() is called it does the following:
    1. Adds the request for memory into the swapOutQueue.
    2. Wakes up the swapOut() process.

  Since all process that sleep while waiting for swap out share a common channel, the swapSatisfied field in struct proc is used for the selective wake up of the process whose swapout request is satisfied.
  Hence process continues to sleep until swapsatisfied !=0.

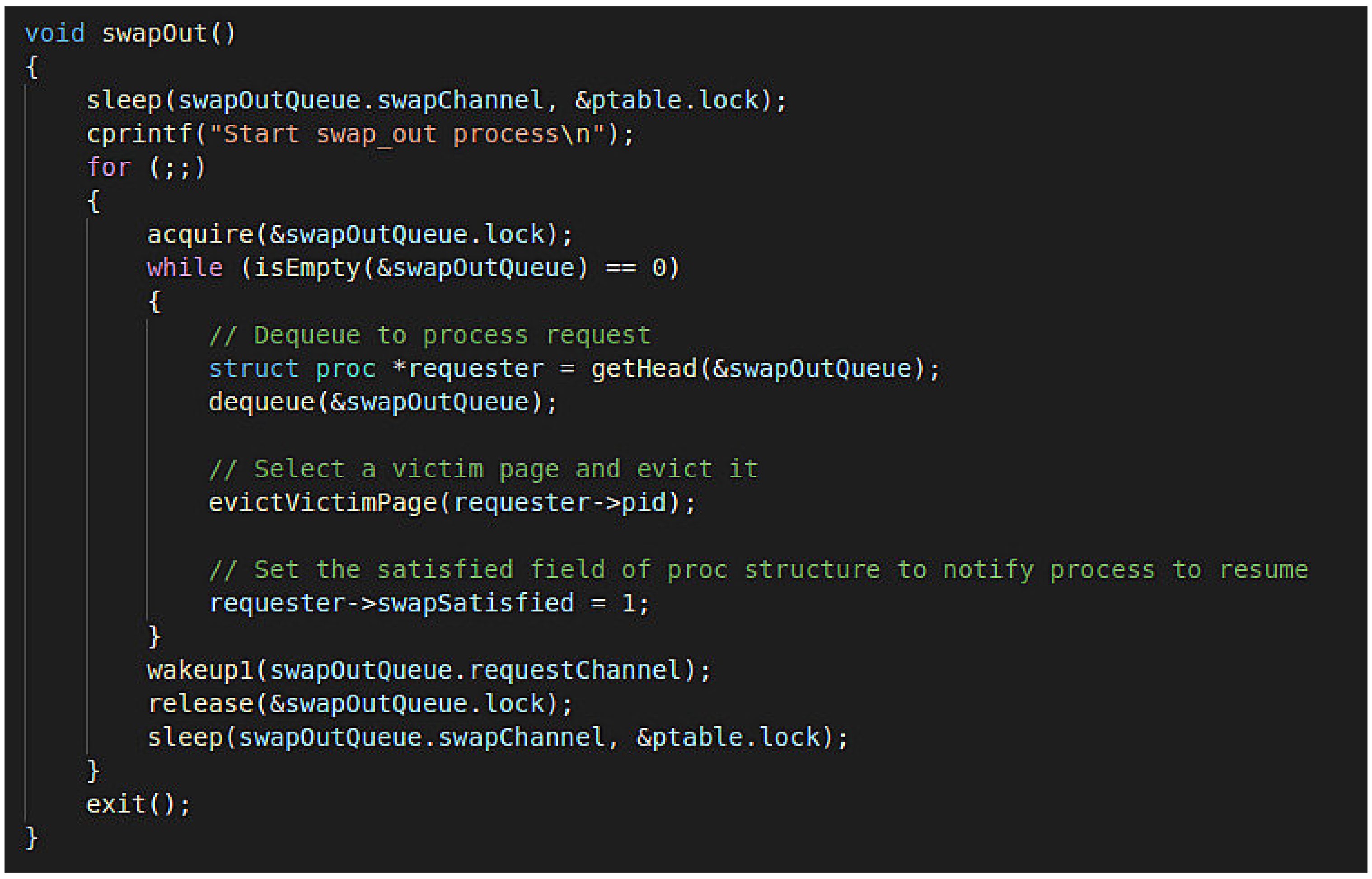


```c
void requestSwapOut()
{
    acquire(&ptable.lock);
    acquire(&swapOutQueue.lock);
    enqueue(&swapOutQueue);
    wakeup1(swapOutQueue.swapChannel);
    release(&swapOutQueue.lock);
    while (myproc()->swapSatisfied == 0)
    {
        sleep(swapOutQueue.requestChannel, &ptable.lock);
    }
    myproc()->swapSatisfied = 0;
    release(&ptable.lock);
}
```

fig 3: requestSwapout()

⇒ **swapOut** :

- The kernel process that deals with victim page selection and swapping out.
- This process sleeps whenever swapOutQueue is empty.
- When there is a new request to process following steps are involved :
  1. getHead of queue and dequeue.
  2. choose a victim page.
  3. Create a file and write contents of victim page in that.
  4. Update page table entry to indicate not present.
  5. set swapSatisfied of requesting process and wake it up.

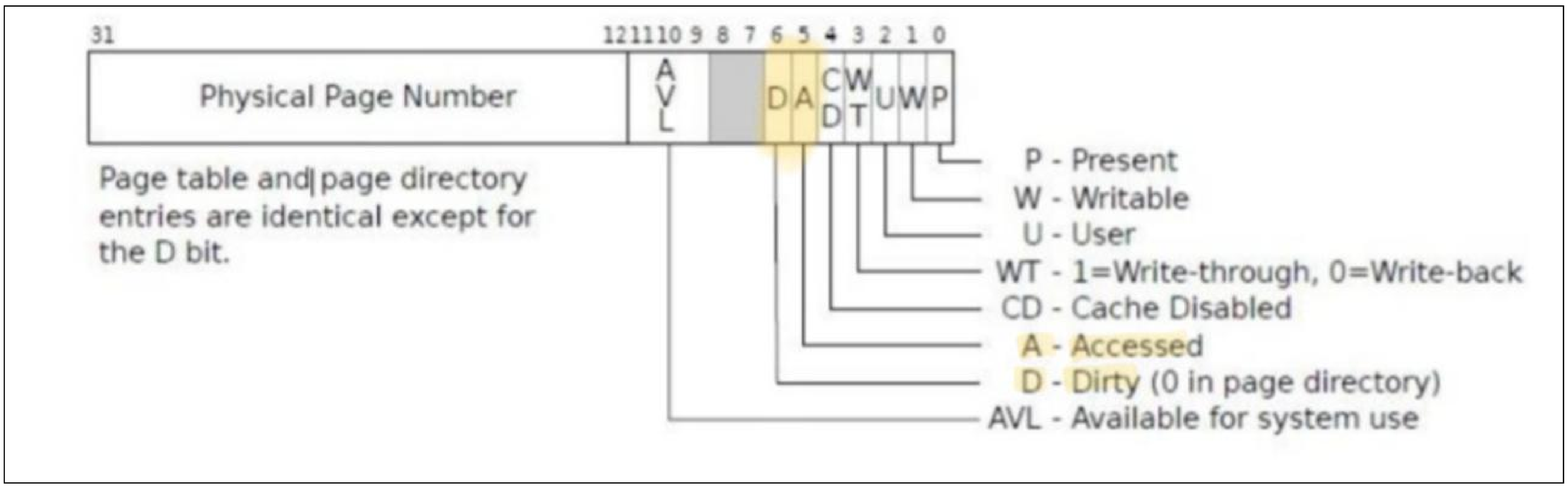


```
void swapOut()
{
    sleep(swapOutQueue.swapChannel, &ptable.lock);
    cprintf("Start swap_out process\n");
    for (;;)
    {
        acquire(&swapOutQueue.lock);
        while (isEmpty(&swapOutQueue) == 0)
        {
            // Dequeue to process request
            struct proc *requester = getHead(&swapOutQueue);
            dequeue(&swapOutQueue);

            // Select a victim page and evict it
            evictVictimPage(requester->pid);

            // Set the satisfied field of proc structure to notify process to resume
            requester->swapSatisfied = 1;
        }
        wakeup1(swapOutQueue.requestChannel);
        release(&swapOutQueue.lock);
        sleep(swapOutQueue.swapChannel, &ptable.lock);
    }
    exit();
}
```

fig 4: swapout()

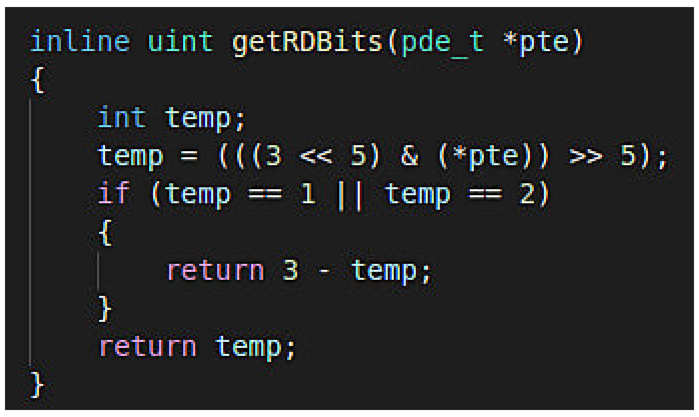
⇒ **evictVictimPage()** :

- It is called by **swapOut()**. It selects victim page then updates its page table entry and writes out page to a file.
- In order to implement **LRU** the function uses the **"Enhanced-Second change Algorithm"** discussed in class.
- xv6 maintains an accessed bit and a modify bit for each PTE as shown below in the figure:


31
12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Page Number
AVL
D
A
CD
WT
U
W
P
Page table and page directory entries are identical except for the D bit.
P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

- The function **getRDbits()** gets the value of (access,modify) bits using bitwise operators.
- For the (access,modify) bits we have the following four possible cases :

1. (0,0) - neither recently used nor modified -> best page to replace.
2. (0,1) - not recently used but modified -> not as good because we need to swap out a page, but still better than used pages.
3. (1,0) - recently used but unmodified.
4. (1, 1) - recently used and modified -> the worst page to replace.



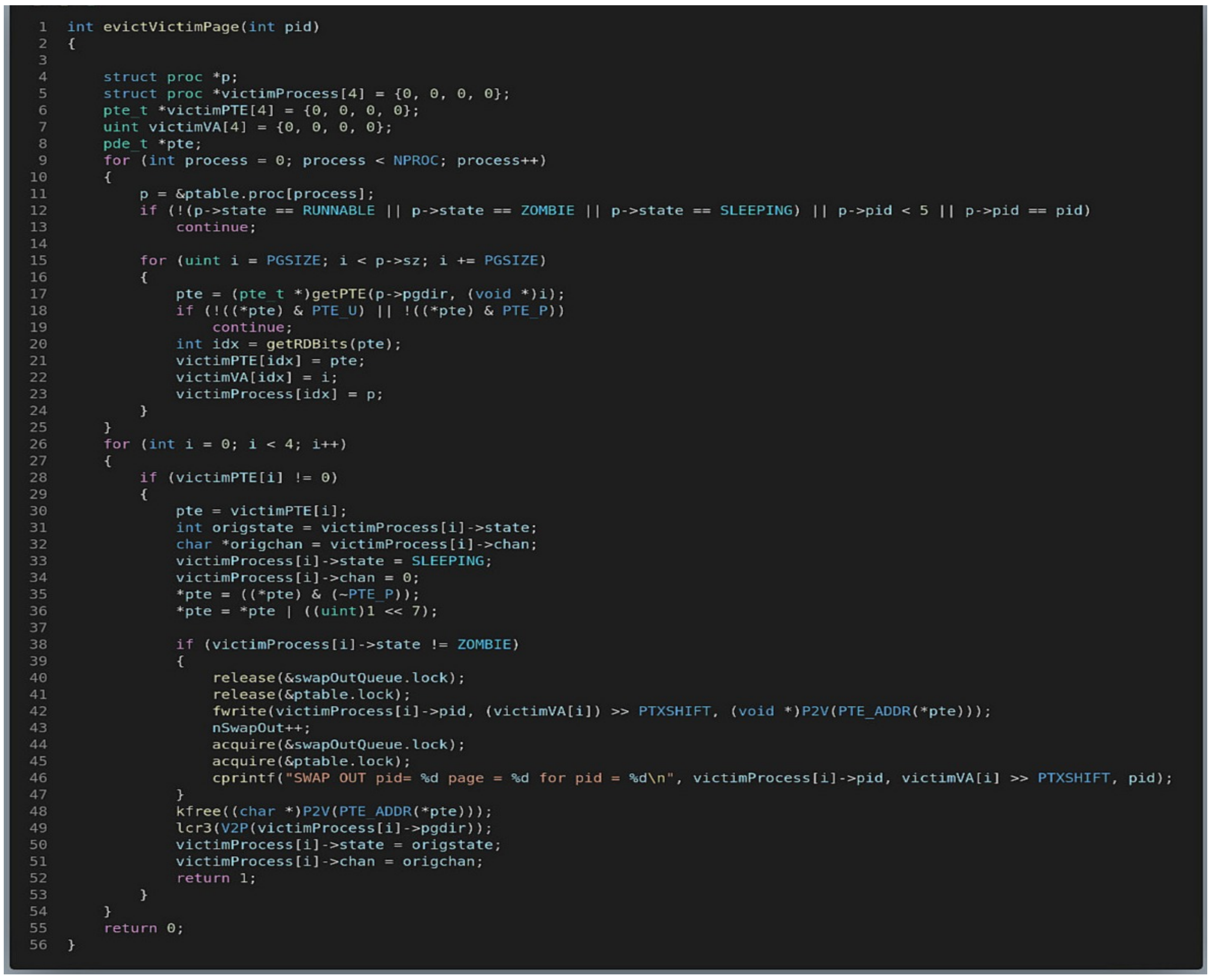
```c
inline uint getRDBits(pde_t *pte)
{
    int temp;
    temp = (((3 << 5) & (*pte)) >> 5);
    if (temp == 1 || temp == 2)
    {
        return 3 - temp;
    }
    return temp;
}
```

fig 5: getRDbits()

⇒ Selection of Victim is done as follows :

- Iterate through ptable and For every process, examine each page.
- Save the required details in **victimPTE**, **victimProcess**, **victimVA** arrays indexed by the value of (access,modify) bits.
- Evict the lowest possible (access,modify) pair value based on the rules described above.
  1. First change state of victim process to **SLEEPING** so that it doesn't run while swap is being performed.
  2. Unset the present bit in **PTE** using **PTE_P**.
  3. Set swapped out bit to indicate page has been swapped out and It will be used by swapIn to Handle page faults.
  4. Create a new file, assign a file descriptor to it and copy contents of page into file.The **getSwapFileName()** gives a unique name for file based on the format **<PID>_<VIRTUAL_PAGE_NUMBER>.swp**

  6. Make a call to kfree to deallocate victim page and refer TLB by calling lcr3()
  7. Set victim process state back to what it was.



```c
int evictVictimPage(int pid)
{

    struct proc *p;
    struct proc *victimProcess[4] = {0, 0, 0, 0};
    pte_t *victimPTE[4] = {0, 0, 0, 0};
    uint victimVA[4] = {0, 0, 0, 0};
    pde_t *pte;
    for (int process = 0; process < NPROC; process++)
    {
        p = &ptable.proc[process];
        if (!(p->state == RUNNABLE || p->state == ZOMBIE || p->state == SLEEPING) || p->pid < 5 || p->pid == pid)
            continue;

        for (uint i = PGSIZE; i < p->sz; i += PGSIZE)
        {
            pte = (pte_t *)getPTE(p->pgdir, (void *)i);
            if (!((*pte) & PTE_U) || !((*pte) & PTE_P))
                continue;
            int idx = getRDBits(pte);
            victimPTE[idx] = pte;
            victimVA[idx] = i;
            victimProcess[idx] = p;
        }
    }
    for (int i = 0; i < 4; i++)
    {
        if (victimPTE[i] != 0)
        {
            pte = victimPTE[i];
            int origstate = victimProcess[i]->state;
            char *origchan = victimProcess[i]->chan;
            victimProcess[i]->state = SLEEPING;
            victimProcess[i]->chan = 0;
            *pte = ((*pte) & (~PTE_P));
            *pte = *pte | ((uint)1 << 7);

            if (victimProcess[i]->state != ZOMBIE)
            {
                release(&swapOutQueue.lock);
                release(&ptable.lock);
                fwrite(victimProcess[i]->pid, (victimVA[i]) >> PTXSHIFT, (void *)P2V(PTE_ADDR(*pte)));
                nSwapOut++;
                acquire(&swapOutQueue.lock);
                acquire(&ptable.lock);
                cprintf("SWAP OUT pid= %d page = %d for pid = %d\n", victimProcess[i]->pid, victimVA[i] >> PTXSHIFT, pid);
            }
            kfree((char *)P2V(PTE_ADDR(*pte)));
            lcr3(V2P(victimProcess[i]->pgdir));
            victimProcess[i]->state = origstate;
            victimProcess[i]->chan = origchan;
            return 1;
        }
    }
    return 0;
}
```

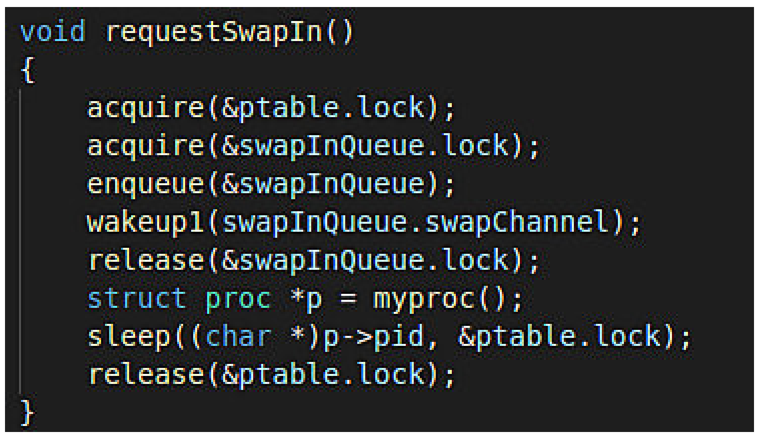
fig 6: evictVictimPage

Task 3:Swapping In

- In **trap.c**, we added the switch case to deal with **T_PGFLT** (page fault trap). Faulting VA present in the cr2 register which is accessed using **rcr2()** function. We fetch the PTE using getPTE() function.If the VA is within process size limit and has been swapped out then call **requestSwapIn()**.

⇒ **requestSwapIn()** : Upon getting called it performs following operations :

- Enqueues current process into swapInQueue.
- wakes up swapIn() kernel process.
- sleeps until it's request is satisfied by swapIn().

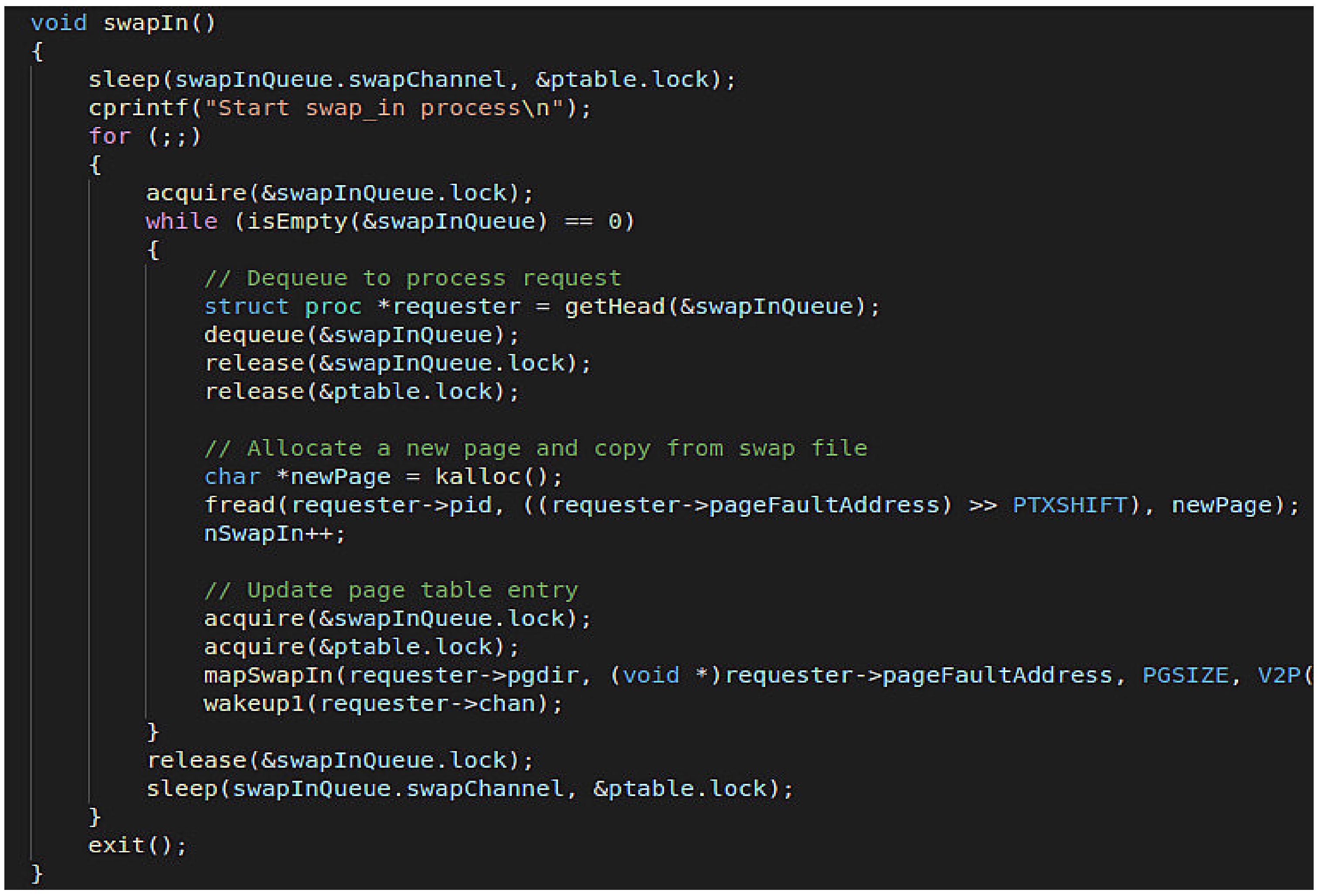


```
void requestSwapIn()
{
    acquire(&ptable.lock);
    acquire(&swapInQueue.lock);
    enqueue(&swapInQueue);
    wakeup1(swapInQueue.swapChannel);
    release(&swapInQueue.lock);
    struct proc *p = myproc();
    sleep((char *)p->pid, &ptable.lock);
    release(&ptable.lock);
}
```

fig 7 : requestSwapIn()

⇒ **swapIn()** :

- This kernel process deals with swapping in pages that were previously swapped out.
- It sleeps when there are no requests i.e swapInQueue is empty.
- It performs the following steps :
  1. dequeue process from the queue.
  2. calls **kalloc()** to get a new free page.
  3. calls **fread()** to copy contents from swapped out page into free page assigned by **kalloc()**.
     - Within fread **getswapFilename()** is called to get swap page name based o page fault address.
     - An inbuilt function **fileread()** is called which copies contents in the free page.
     - Deletes the swapped out page since its no longer required by calling **fdelete()**.
  4. updates PTE entry with new PHYSICAL address by calling **mapSwapIn()**.
  5. Wakes up requesting process.



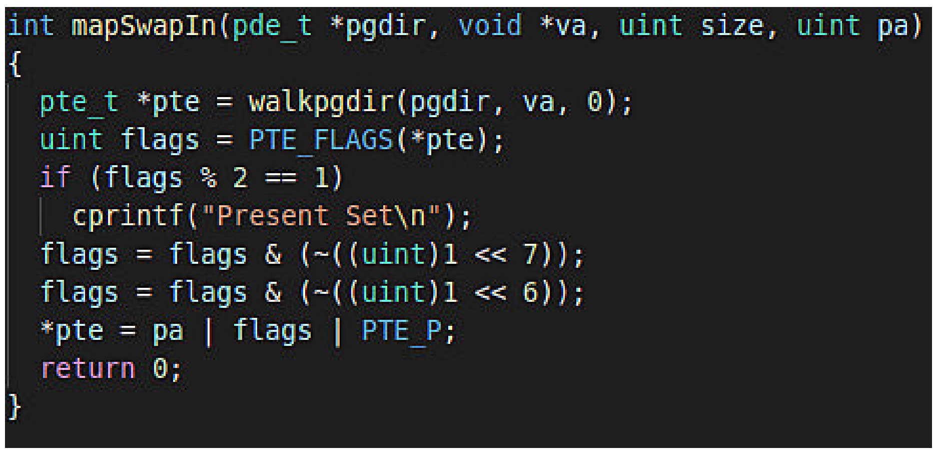
```
void swapIn()
{
    sleep(swapInQueue.swapChannel, &ptable.lock);
    cprintf("Start swap_in process\n");
    for (;;)
    {
        acquire(&swapInQueue.lock);
        while (isEmpty(&swapInQueue) == 0)
        {
            // Dequeue to process request
            struct proc *requester = getHead(&swapInQueue);
            dequeue(&swapInQueue);
            release(&swapInQueue.lock);
            release(&ptable.lock);

            // Allocate a new page and copy from swap file
            char *newPage = kalloc();
            fread(requester->pid, ((requester->pageFaultAddress) >> PTXSHIFT), newPage);
            nSwapIn++;

            // Update page table entry
            acquire(&swapInQueue.lock);
            acquire(&ptable.lock);
            mapSwapIn(requester->pgdir, (void *)requester->pageFaultAddress, PGSIZE, V2P(
            wakeup1(requester->chan);
        }
        release(&swapInQueue.lock);
        sleep(swapInQueue.swapChannel, &ptable.lock);
    }
    exit();
}
```

fig 8 : swapIn()

⇒ **mapSwapIn()** : It was implemented in **vm.c**. It is used to update physical address in PTE during swap in.

It performs the following steps :

1. Page table of process is traversed to get PTE using **walkpgdir().**
2. Sets the PTE_P flag.
3. Unsets the modify bit.
4. Unsets the swapped out bit.
5. Updates the physical address along with other permission bits.



```
int mapSwapIn(pde_t *pgdir, void *va, uint size, uint pa)
{
  pte_t *pte = walkpgdir(pgdir, va, 0);
  uint flags = PTE_FLAGS(*pte);
  if (flags % 2 == 1)
    cprintf("Present Set\n");
  flags = flags & (~((uint)1 << 7));
  flags = flags & (~((uint)1 << 6));
  *pte = pa | flags | PTE_P;
  return 0;
}
```

fig 9 : mapSwapIn()

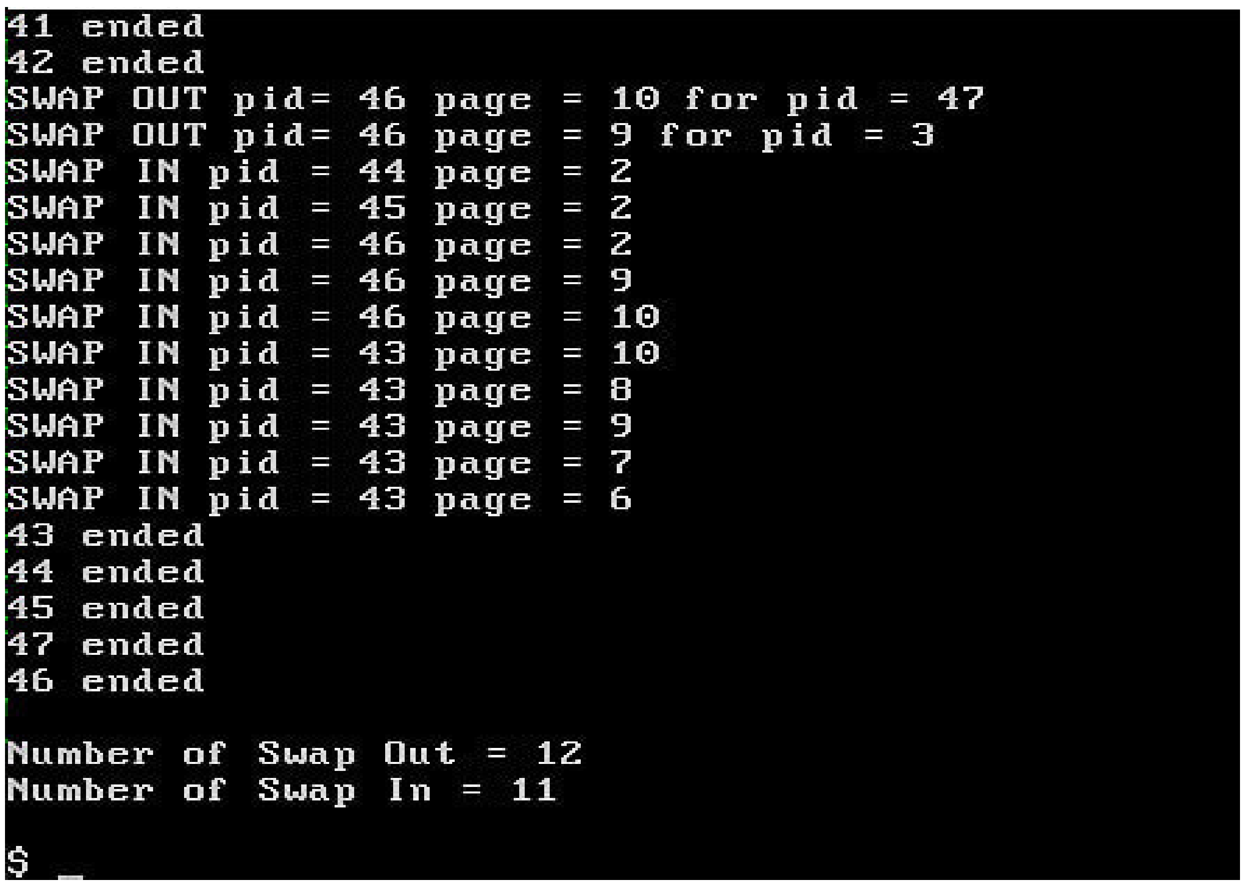
⇒ **deleteSwapPages() :**

- It iterates through the open files list of the swapout process, and if the file is not already deleted then it deletes it.
- It is Called in proc.c whenever a child of the shell process exits.

<u>Task 4:Sanity Test</u>

⇒ **memtest.c :**

- This is our user program, it forks 20 child processes. Each child process iterates 20 times and calls **malloc(PGSIZE)**.
- The new page assigned is interpreted as character array Each element of the array i.e each byte is assigned the value (j*k)%128 , where:
  j = iteration number
  k = byte number within newly allocated free page
- Then the child runs the same loop again and examines if the values at each byte are as expected.



```
41 ended
42 ended
SWAP OUT pid= 46 page = 10 for pid = 47
SWAP OUT pid= 46 page = 9 for pid = 3
SWAP IN pid = 44 page = 2
SWAP IN pid = 45 page = 2
SWAP IN pid = 46 page = 2
SWAP IN pid = 46 page = 9
SWAP IN pid = 46 page = 10
SWAP IN pid = 43 page = 10
SWAP IN pid = 43 page = 8
SWAP IN pid = 43 page = 9
SWAP IN pid = 43 page = 7
SWAP IN pid = 43 page = 6
43 ended
44 ended
45 ended
47 ended
46 ended

Number of Swap Out = 12
Number of Swap In = 11

$ _
```

fig 10 : sample output picture

⇒ **Observations** :

- Since there is no **"Error"** output, it implies the page table entries of swapped out and swapped in pages were handled correctly and user processes are able to continue execution without page faults due to incorrect PTEs.
- The number of swapped out pages = 12 and number of swapped in pages = 11. The number of swapped in pages is less than that swapped out because some process that was swapped out must have completed execution before accessing the swapped out page. As a result, this process did not create a page fault and did not request for swapIn.
