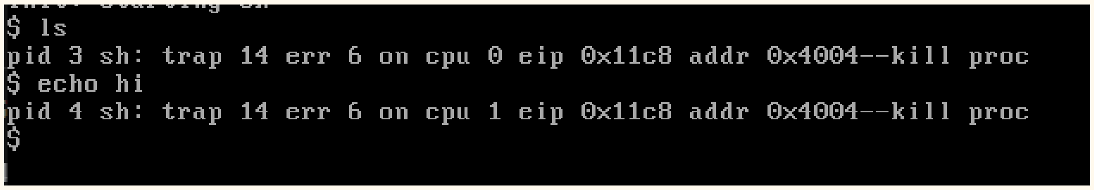
CS344

ASSIGNMENT-3

By Group-11

Part-A

Here we have to implement the lazy allocation for this first we have applied the patch file provided in the assignment , the sys_brk(n) function have been changed previously we have been allocating the memory whenever it is needed by calling growproc() function which in turn calls allocuvm() for assigning the space to the process but now we just increase the process size but we aren't allocating it so whenever a line of code is running and incurs a page fault then we will bring the page into the main memory this is known as lazy allocation. SO first let us run our code without implementing the lazy allocation and see the output.



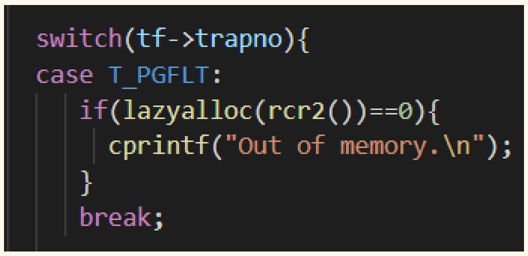
```
$ ls
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x11c8 addr 0x4004--kill proc
$ echo hi
pid 4 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$
```

So now let us implement the lazy allocation for that we have to modify the following three files.

1. **sysproc.c:**

   We have applied the patch provided in the assignment there we have commented out the growproc function and we have just increased the size of the process and this will incur a page fault at the time of running.

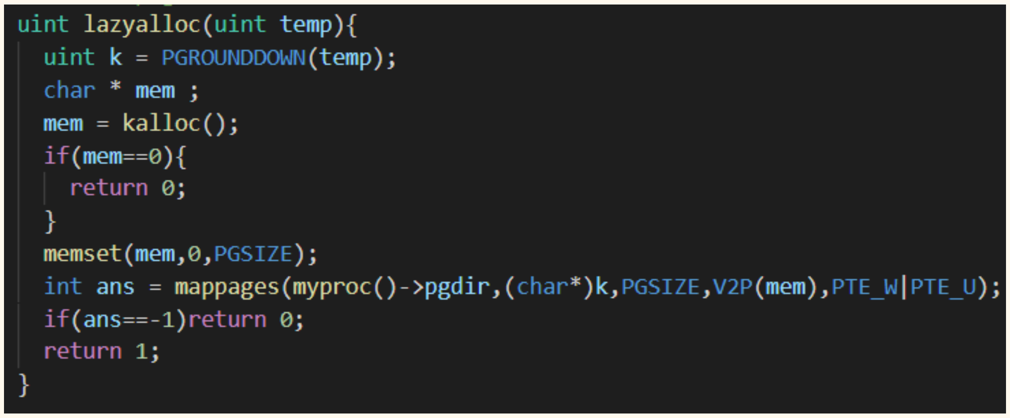
2. **trap.c:**



```
switch(tf->trapno){
case T_PGFLT:
    if(lazyalloc(rcr2())==0){
      cprintf("Out of memory.\n");
    }
    break;
```

When a page fault has occurred then the value of tf->trapno will be equal to T_PGFLT. So now we have to allocate a frame to the page at which the fault occurred . So to get the instruction at which the page fault occurred we would get it from rcr2() function. So we will implement the lazyalloc function which takes the rcr2() function as a parameter and it has been implemented in the vm.c file. We will use break statements to prevent extra printf statements and proc->killed = 1 statements.

3. vm.c:

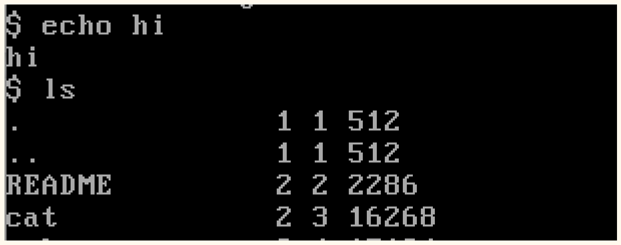


```c
uint lazyalloc(uint temp){
  uint k = PGROUNDDOWN(temp);
  char * mem ;
  mem = kalloc();
  if(mem==0){
    return 0;
  }
  memset(mem,0,PGSIZE);
  int ans = mappages(myproc()->pgdir,(char*)k,PGSIZE,V2P(mem),PTE_W|PTE_U);
  if(ans==-1)return 0;
  return 1;
}
```

Here we will get the parameter rcr2() which will give the instruction at which the page fault occurred and we will use the PGROUNDDOWN function to find the starting position of page and we will call kalloc to find a free frame from the set of free frames which are stored in the linked list and then we will set the memory to zero and call the mappages function which takes the arguments myproc()->pgdir which gives the processes page directory and the page number and page size, physical address of the memory to which this page has been allotted and permissions to be set on the page and here we have to change the return type of mappages function from static int to int.

4. defs.h:

Here we will include the prototype or define the lazyalloc() function so that we can use it in trap.c.



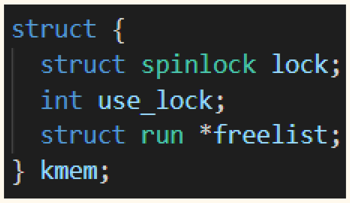
```
$ echo hi
hi
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16268
```

We can see that now all the functions have been working properly so this tests us that our implementation is correct.

Part-B

Questions:

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



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

Xv6 maintains a list of free pages in the struct kmem as freelist which is implemented in the kalloc.c file as shown and it also calls the kinit1 at the starting as the list is empty it will allot 4MB of free pages initially.

2. **What data structures are used to answer this question?**

   Free frames are stored in a linked list named free list as shown which the entries are of the data structure struct run which is defined in kalloc.c file and whenever a frame is freed we will call the kfree function which inserts into freelist by typecasting as (struct run*).

3. **Where do these reside?**

   The linked list freelist is defined in kmem structure which is of type struct run which is also defined in the kalloc.c file.

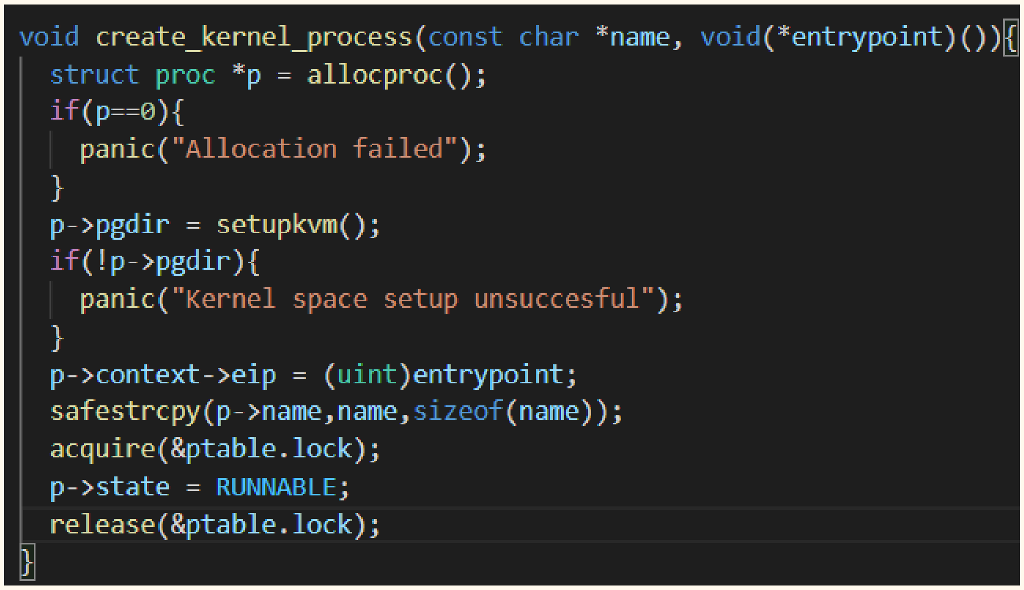
4. **Does xv6 memory mechanism limit the number of user processes**

   In xv6 the maximum number of processes has been limited by the value NPROC which is defined as 64 in param.h as default due to the limitations of the process table . So by default the maximum number of user processes are 64.

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

   When xv6 is booted there is only one process which is initproc . As we know that a process can have a max of 2GB (KERNBASE) and maximum physical address of 240MB(PHYSTOP) , so one process can take all the process available so lowest number could be 1 and it is not also zero because there must be at least one user process to interact which is forked from either initproc or sh .

TASK-01:

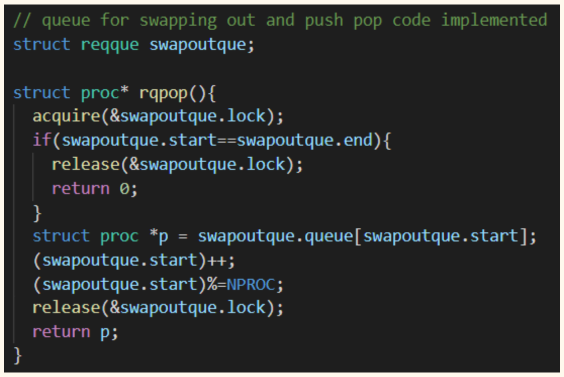


```c
void create_kernel_process(const char *name, void(*entrypoint)()){
  struct proc *p = allocproc();
  if(p==0){
    panic("Allocation failed");
  }
  p->pgdir = setupkvm();
  if(!p->pgdir){
    panic("Kernel space setup unsuccesful");
  }
  p->context->eip = (uint)entrypoint;
  safestrcpy(p->name,name,sizeof(name));
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
}
```

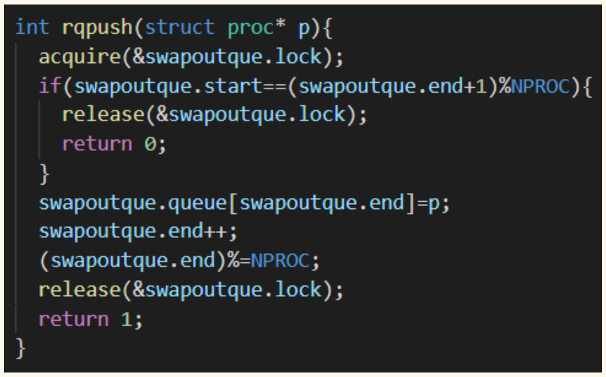
We have implemented the create_kernel_process function in the proc.c file. As the kernel process always remains in kernel mode the whole time we need not to initialise its trap frame as the trap frame contains userspace registers and user section of the page table . allocproc has been used to assign the process a place in the process table and after that we have used setupkvm() function to setup the kernel part of the process page table which is used to map the address above KERNBASE to the physical address between 0 & PHYSTOP . And finally we have set the eip register to entrypoint as this is where the information about the next instruction to be run is present and we have assigned the given name to the process using safestrcpy() function and acquired a lock for page table and changed the state of the process to runnable.

TASK-02:

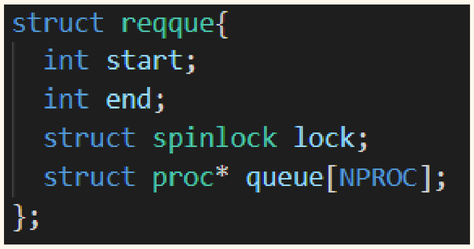
Here we have to implement the swapping out mechanism and for that it should also support multiple requests , so for that first we will create a request queue for that we define a data structure struct reqque and it contains a queue of size NPROC as at a time the max number of process are NPROC and lock to maintain synchronisation and two pointers start and end to maintain the front and rear pointer of the circular queue. And then we have defined a swapoutque , further we have implemented rqpush() and rqpop() function for implementing push and popping into the queue and we have initialised the locks in the pinit() function and in userinit() we have initialised the start and end pointer all the modifications are done in proc.c file and shown in the below screenshots.



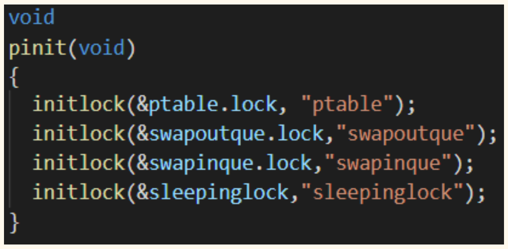
```
// queue for swapping out and push pop code implemented
struct reqque swapoutque;

struct proc* rqpop(){
  acquire(&swapoutque.lock);
  if(swapoutque.start==swapoutque.end){
    release(&swapoutque.lock);
    return 0;
  }
  struct proc *p = swapoutque.queue[swapoutque.start];
  (swapoutque.start)++;
  (swapoutque.start)%=NPROC;
  release(&swapoutque.lock);
  return p;
}
```



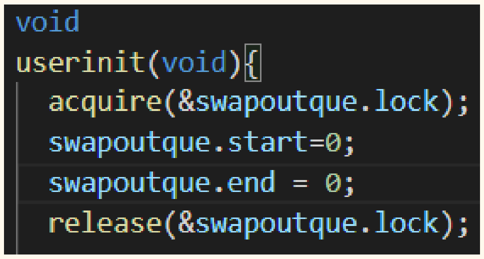
```
int rqpush(struct proc* p){
  acquire(&swapoutque.lock);
  if(swapoutque.start==(swapoutque.end+1)%NPROC){
    release(&swapoutque.lock);
    return 0;
  }
  swapoutque.queue[swapoutque.end]=p;
  swapoutque.end++;
  (swapoutque.end)%=NPROC;
  release(&swapoutque.lock);
  return 1;
}
```



```
struct reqque{
  int start;
  int end;
  struct spinlock lock;
  struct proc* queue[NPROC];
};
```

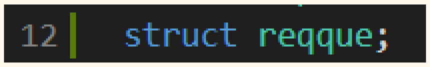


```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&swapoutque.lock,"swapoutque");
  initlock(&swapinque.lock,"swapinque");
  initlock(&sleepinglock,"sleepinglock");
}
```

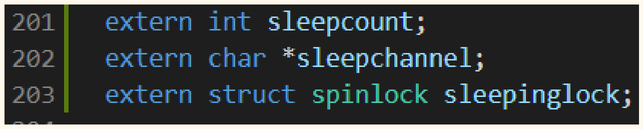


```
void
userinit(void){
  acquire(&swapoutque.lock);
  swapoutque.start=0;
  swapoutque.end = 0;
  release(&swapoutque.lock);
```

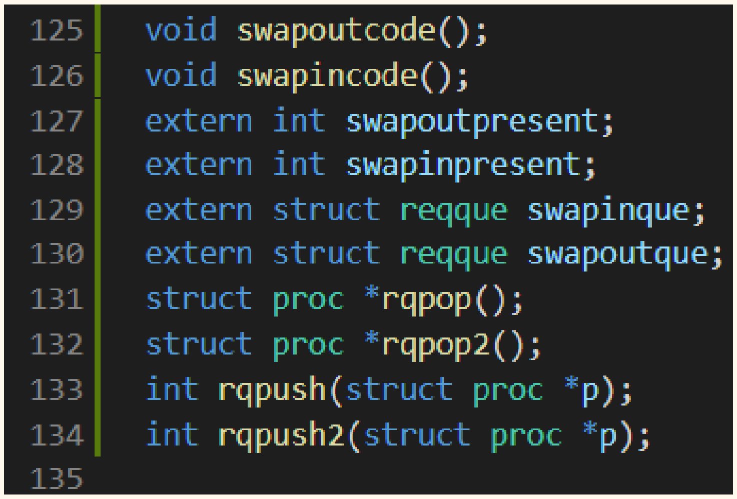
Similarly we have also defined the prototypes in defs.h as shown in the below



```
struct reqque;
```



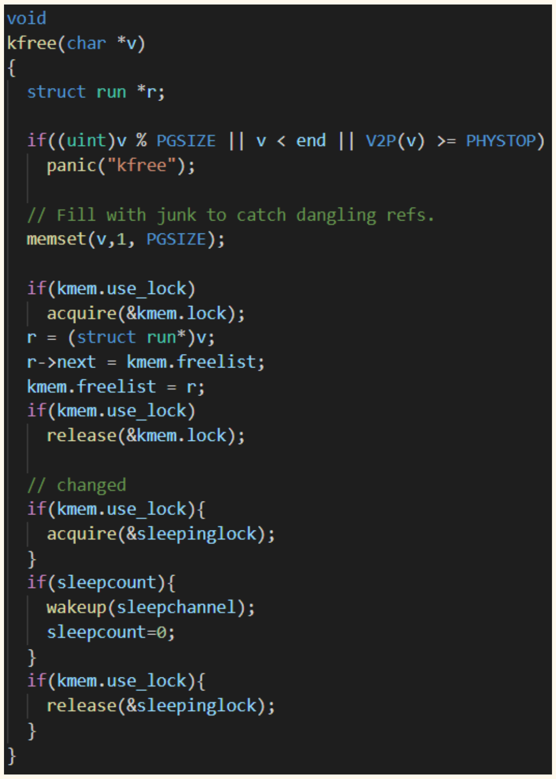
```
extern int sleepcount;
extern char *sleepchannel;
extern struct spinlock sleepinglock;
```



```
void swapoutcode();
void swapincode();
extern int swapoutpresent;
extern int swapinpresent;
extern struct reqque swapinque;
extern struct reqque swapoutque;
struct proc *rqpop();
struct proc *rqpop2();
int rqpush(struct proc *p);
int rqpush2(struct proc *p);
```

Now we know that whenever kalloc is not able to allocate pages then it returns zero and this notifies allocuvm that the requested memory is not available so it will go to mem==0 case . So here what we will do is we will define a sleeping channel so that all the processes which are made into sleeping because of unavailability of memory and we will also maintain a sleep count to maintain the count of the number of suspended processes sleeping in the sleeping channel . So in the allocuvm when mem==0 we will acquire a lock and change the state to sleeping and keep the channel to the sleeping channel and add the request to the swapoutque.

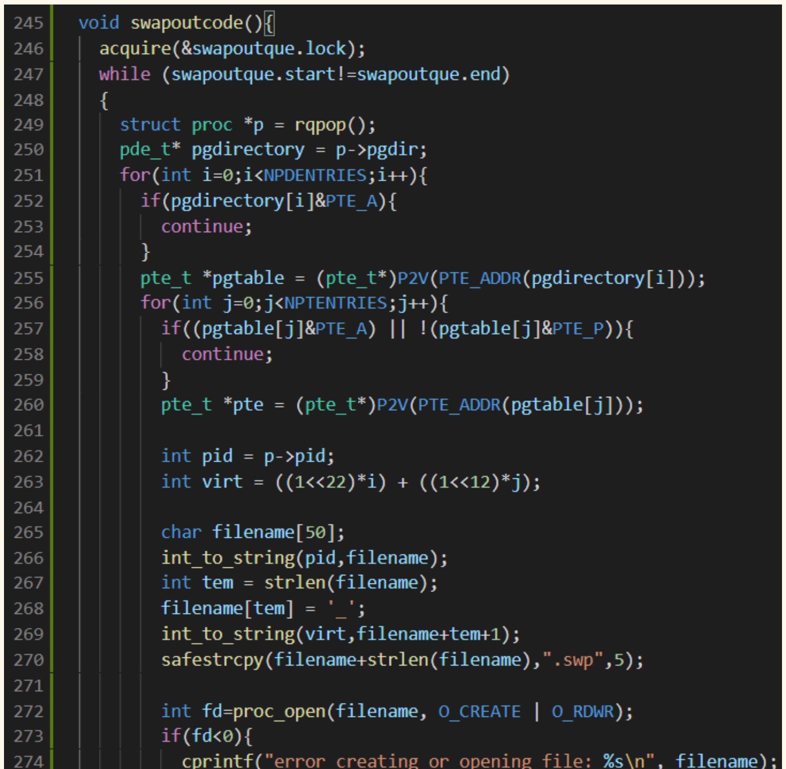
Then we will initialise a bool swapoutpresent to 0 and if not exists already we will create the swapout function as it is kernel process we will use the create_kernel_process implementation as in task1 and create it and also make the boolean to true we will get know what is use of this in later part of assignment.



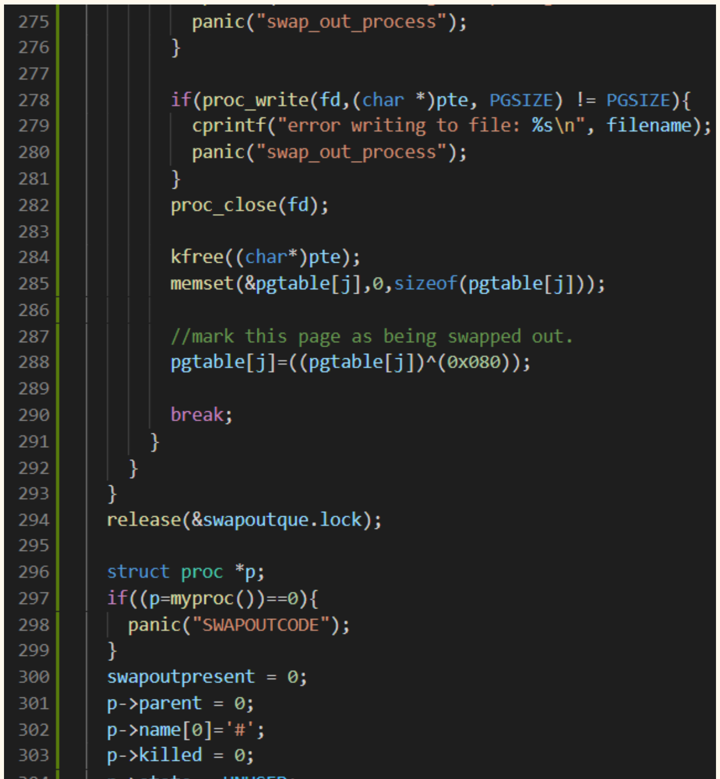
```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v,1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);

  // changed
  if(kmem.use_lock){
    acquire(&sleepinglock);
  }
  if(sleepcount){
    wakeup(sleepchannel);
    sleepcount=0;
  }
  if(kmem.use_lock){
    release(&sleepinglock);
  }
}
```

Next we will do is when a free page has been available we will wake up all the processes in the sleeping channel as we know a page is freed and added to freelist using the kfree function in kalloc.c . As we can see, we acquired a lock and changed the sleep count to zero and woke up all the processes in the sleeping channel.

Now let us see the implementation of swap out code as shown in the function below .

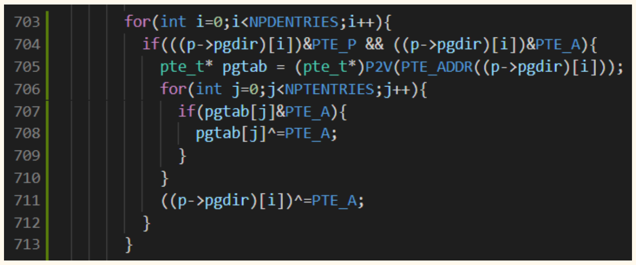


```
void swapoutcode(){
  acquire(&swapoutque.lock);
  while (swapoutque.start!=swapoutque.end)
  {
    struct proc *p = rqpop();
    pde_t* pgdirectory = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){
      if(pgdirectory[i]&PTE_A){
        continue;
      }
      pte_t *pgtable = (pte_t*)P2V(PTE_ADDR(pgdirectory[i]));
      for(int j=0;j<NPTENTRIES;j++){
        if((pgtable[j]&PTE_A) || !(pgtable[j]&PTE_P)){
          continue;
        }
        pte_t *pte = (pte_t*)P2V(PTE_ADDR(pgtable[j]));

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

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

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



```
          panic("swap_out_process");
        }

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

        kfree((char*)pte);
        memset(&pgtable[j],0,sizeof(pgtable[j]));

        //mark this page as being swapped out.
        pgtable[j]=((pgtable[j])^(0x080));

        break;
      }
    }
  }
  release(&swapoutque.lock);

  struct proc *p;
  if((p=myproc())==0){
    panic("SWAPOUTCODE");
  }
  swapoutpresent = 0;
  p->parent = 0;
  p->name[0]='#';
  p->killed = 0;
```

Firstly we have acquired a lock and we are iterating through all the requests present in the swapoutque and we are popping the requests one by one and we will choose the victim frame using the LRU policy for that we have been using the LRU approximation algorithm known as reference bit algorithm in this algo we will iterate through the all allocated frames to the given process and finds the frame with the access bit is unset . As we will know xv6 implement paging mechanism as a two level page table.So first we get the page directory i.e.outer page table of the current process and we check whether it is accessed or not if the page directory has been accessed then we skip going into the inner page table or else we will get the inner page table corresponding to the entry of outer page table and then we will check whether a page has been allocated a frame or not because we cannot replace the page for which page is not allotted so we check the present bit if it is not present or present and accessed we will skip it or else we have just found out the victim frame .



```
for(int i=0;i<NPDENTRIES;i++){
  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;
  }
}
```

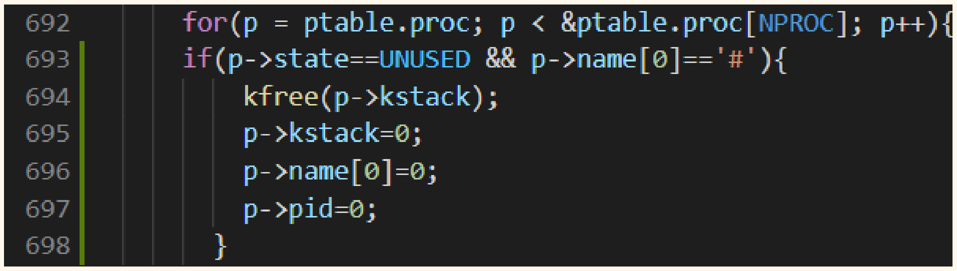
In the scheduler function we will be unsetting the access bit when the process has been context switched . so basically the code here unsets the access bit for each page table entry which has been context switching out .

Now we have to store the victim frame in backing store as the notation given in the question so for that we are getting the pid and we will be getting the VA[20:] as the outer for loop represents the first 10 bits and then we will name the file according to the convention and we have used the function int_to_string to convert into string.Now we have to write this into file and we cannot access the functions in sysfile.c so for that i have written the new functions in proc.c named as proc_open, proc_read,proc_write,proc_create so on the basic code has been taken from the sysfile.c file where the implementation of file_open file_read are present . Now we have opened a file with the filename as specified and we will use the fd and open using proc_open with the file name and we set permissions as O_CREATE and O_RDWR where all these macros are present in fcntl.h. O_CREATE creates the file if not present and now we get the page table entry pte and we get the corresponding frame and we will write the page into the file using proc_write function and we will close the file and now we will free the frame using kfree and we set the bit at 8th position to mark that this page has been swapped out and resets the page content .

SUSPENDING SWAP OUT PROCESS:

When the request queue is empty we will come out of loop and makes the boolean to false and

Make process state to unused and killed is set to 0 and and name 0th character is set to # to identify in the scheduler to free the kstack allotted as shown below.



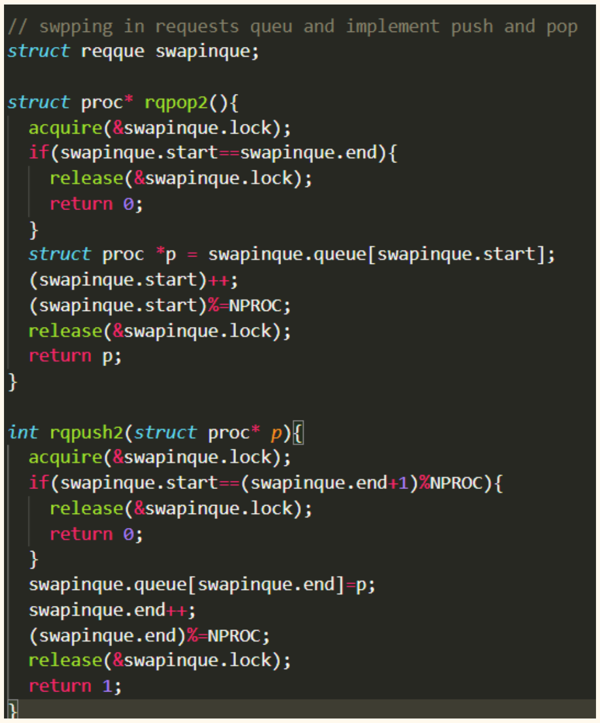
```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state==UNUSED && p->name[0]=='#'){
        kfree(p->kstack);
        p->kstack=0;
        p->name[0]=0;
        p->pid=0;
      }
```

All check marks provided have been accomplished:

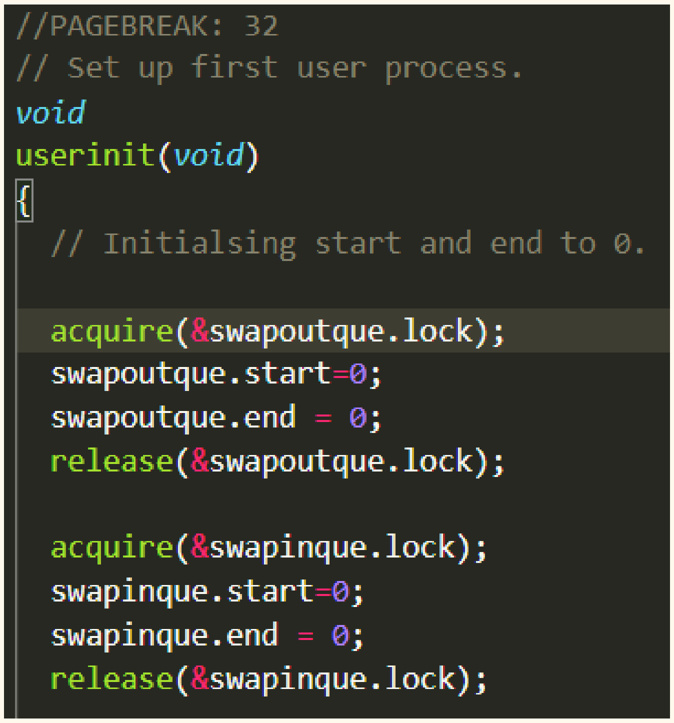
1. Swapoutque has been implemented to support the request queue.
2. Whenever there are no pending requests for the swapping out process, this process must be suspended from execution.(just implemented above)
3. Whenever there exists at least one free physical page, all the processes that were suspended due to lack of physical memory must be woken up.(sleeping channel and kfree implementation as mentioned above)
4. Only user-space memory can be swapped out (this does not include the second-level page table)

TASK-03:

In this task , as we had to handle multiple swap in requests, we have declared a **swap in request queue** of the type **reqque**, as described in task 2, in **proc.c**. We name this request queue as **swapinque.** We have also declared its global prototype using extern keyword in defs.h. Along with declaring the queue, we also created the corresponding functions for **swapinque** (**rqpop2()** and **rqpush2()**) in proc.c and declared their prototype in defs.h. We also initialised its lock in **pinit** function . We also initialised its start and end variables in **userinit** function which sets up the first user process.

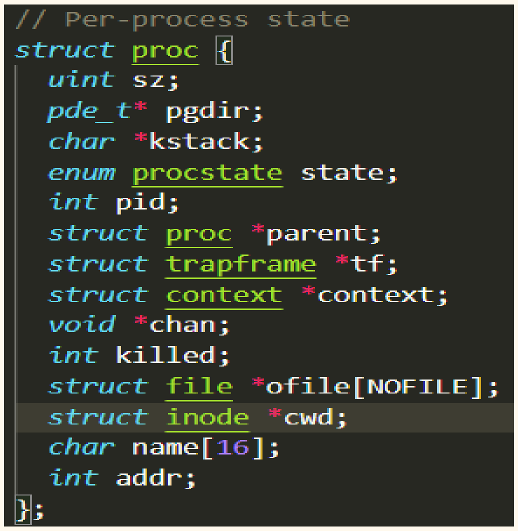


```
// swpping in requests queu and implement push and pop
struct reqque swapinque;

struct proc* rqpop2(){
  acquire(&swapinque.lock);
  if(swapinque.start==swapinque.end){
    release(&swapinque.lock);
    return 0;
  }
  struct proc *p = swapinque.queue[swapinque.start];
  (swapinque.start)++;
  (swapinque.start)%=NPROC;
  release(&swapinque.lock);
  return p;
}

int rqpush2(struct proc* p){
  acquire(&swapinque.lock);
  if(swapinque.start==(swapinque.end+1)%NPROC){
    release(&swapinque.lock);
    return 0;
  }
  swapinque.queue[swapinque.end]=p;
  swapinque.end++;
  (swapinque.end)%=NPROC;
  release(&swapinque.lock);
  return 1;
}
```

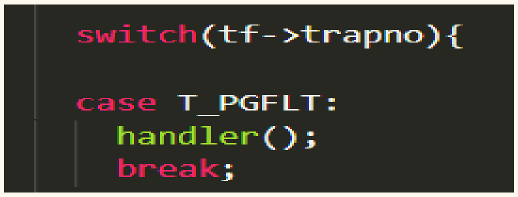


```
//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  // Initialsing start and end to 0.

  acquire(&swapoutque.lock);
  swapoutque.start=0;
  swapoutque.end = 0;
  release(&swapoutque.lock);

  acquire(&swapinque.lock);
  swapinque.start=0;
  swapinque.end = 0;
  release(&swapinque.lock);
```

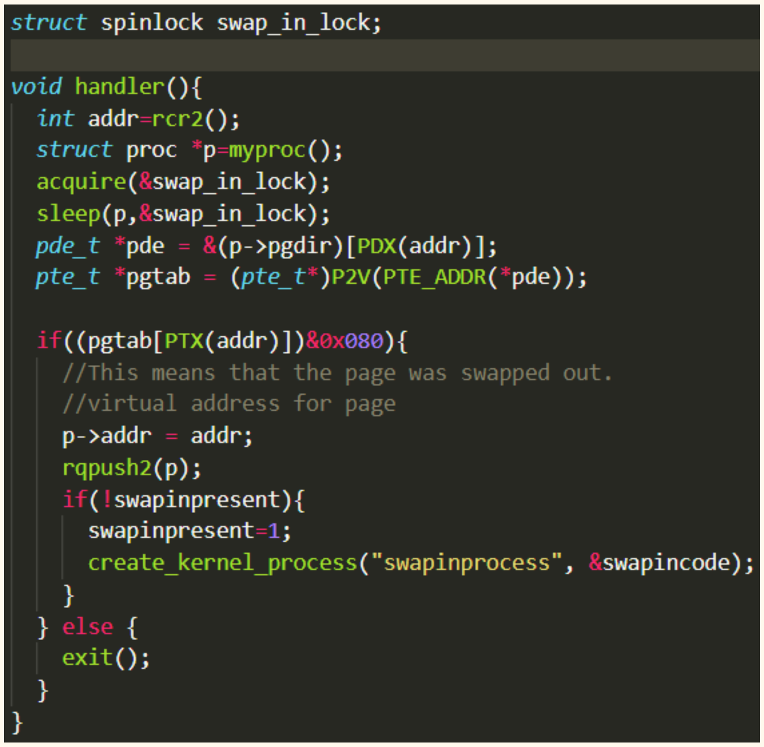
Now as we need to know the virtual address of the page for which the fault occurred we add **addr** variable to the **proc** struct definition in proc.h. On page fault trap we call the function **handler()** defined in trap.c which handles the page fault. In the handler function, just like Part A, we find the virtual address at which the page fault occurred by using rcr2(). We then put the current process to sleep with a new lock called swap_in_lock (initialised locally in trap.c and globally using extern in defs.h). We then obtain the page table entry corresponding to this address using the macros PDX, P2V and PTE_ADDR as shown in figure. Now, we need to check whether this page was swapped out. In Task 2, whenever we swapped out a page , we set its page table entry's bit (8th bit from the LSB side) as described earlier in the report. Thus, in order to



```
// Per-process state
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
  int addr;
};
```



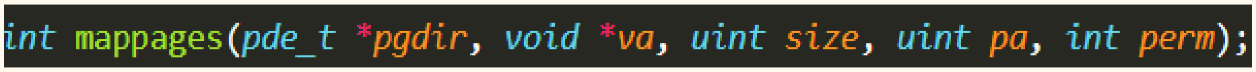
```
  switch(tf->trapno){

  case T_PGFLT:
    handler();
    break;
```



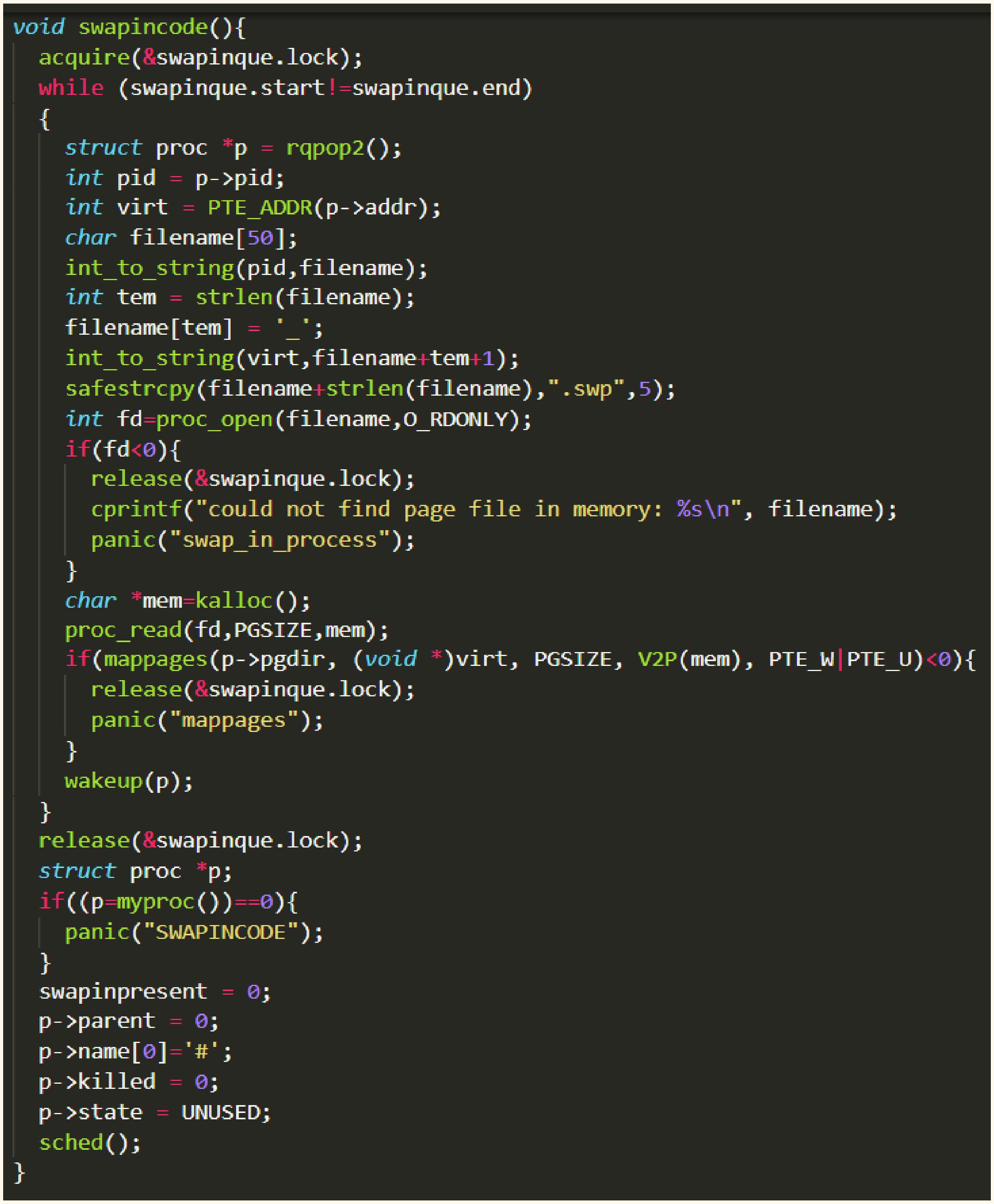
```
struct spinlock swap_in_lock;

void handler(){
  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;
    rqpush2(p);
    if(!swapinpresent){
      swapinpresent=1;
      create_kernel_process("swapinprocess", &swapincode);
    }
  } else {
    exit();
  }
}
```

check whether the page was swapped out or not, we check the page table entry bit using bitwise & with 0x080. If it is set, push this process into the swap in request queue and initiate swap in process if it doesn't already exist Otherwise we safely suspend the process using **exit()** as asked in the assignment.

Now we will see swapinprocess whose entry point is swapincode function(proc.c) as shown in handler function. In this function we run a loop until the swapinque is not empty. We pop out a process one by one from the swapinque and create its filename using its virtual address and process id with the help of int_to_string function. Then we open the file in read only mode(O_RDONLY) using the proc_open function and save its file descriptor in the fd variable. Now we allocate a free frame(mem) using kalloc() and read the file into this frame using the proc_read function. We then use mappages to map the page corresponding to virtual address (virt) with the physical page (mem) that we got using kalloc and read into. To use mappages in proc.c we remove **static** keyword from its declaration to declare a prototype in proc.c. Now after swapping in the page we wakeup the process. After we finish the loop (i.e. All the pages are swapped in) we terminate the kernel process as described in task 2.

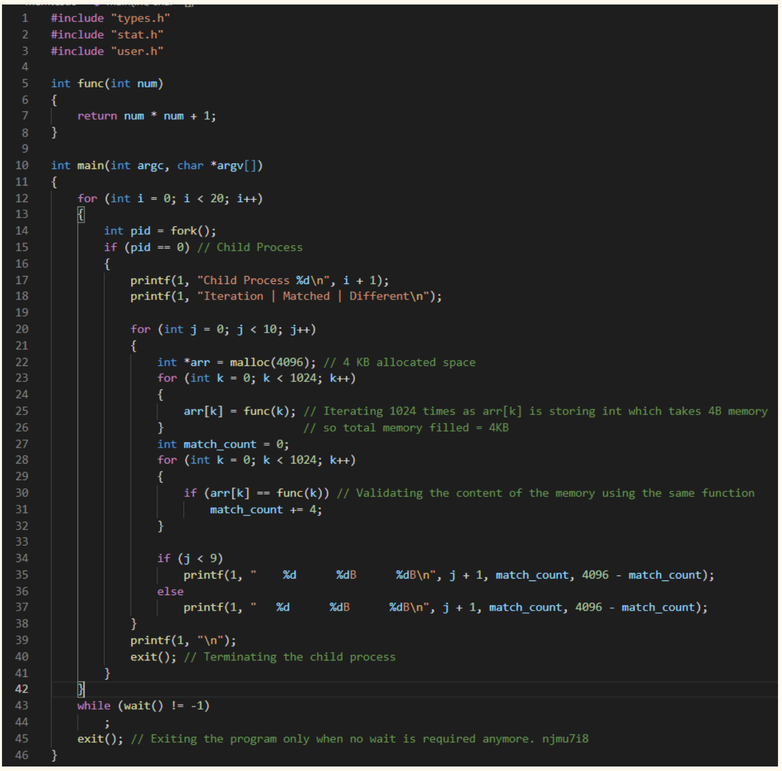


```c
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```



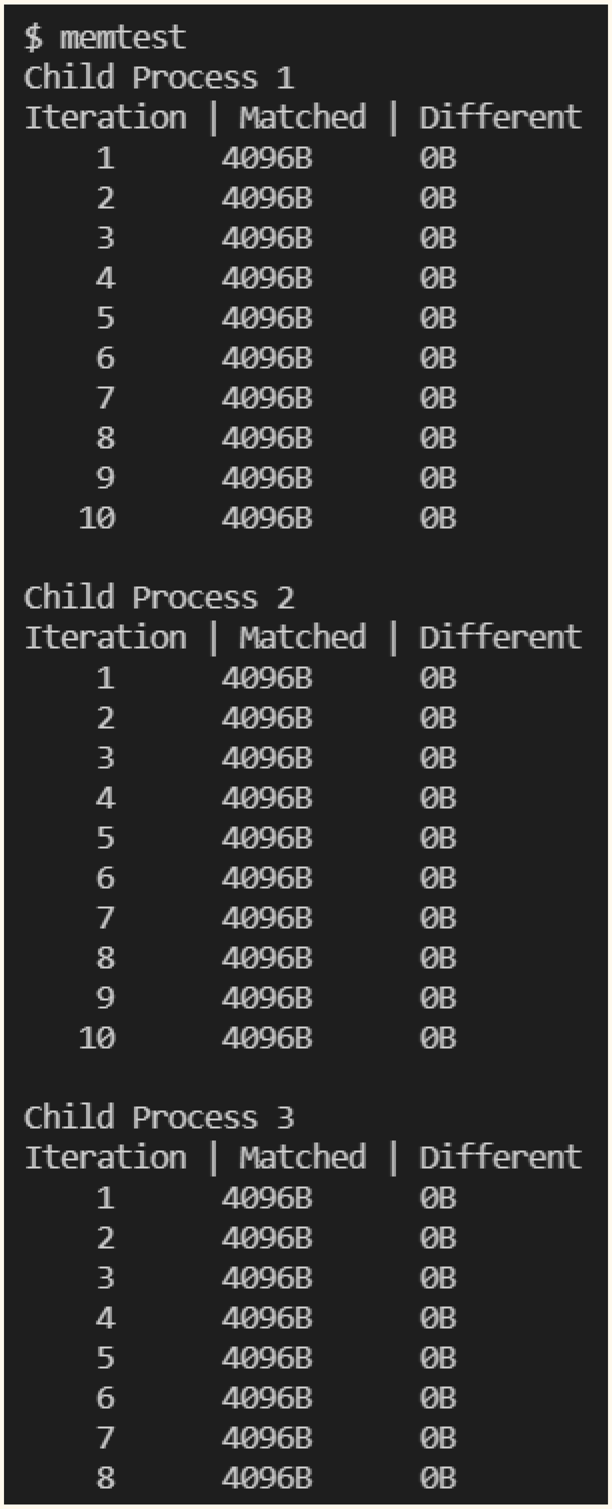
```c
void swapincode(){
  acquire(&swapinque.lock);
  while (swapinque.start!=swapinque.end)
  {
    struct proc *p = rqpop2();
    int pid = p->pid;
    int virt = PTE_ADDR(p->addr);
    char filename[50];
    int_to_string(pid,filename);
    int tem = strlen(filename);
    filename[tem] = '_';
    int_to_string(virt,filename+tem+1);
    safestrcpy(filename+strlen(filename),".swp",5);
    int fd=proc_open(filename,O_RDONLY);
    if(fd<0){
      release(&swapinque.lock);
      cprintf("could not find page file in memory: %s\n", filename);
      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(&swapinque.lock);
      panic("mappages");
    }
    wakeup(p);
  }
  release(&swapinque.lock);
  struct proc *p;
  if((p=myproc())==0){
    panic("SWAPINCODE");
  }
  swapinpresent = 0;
  p->parent = 0;
  p->name[0]='#';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

TASK-04:



```c
#include "types.h"
#include "stat.h"
#include "user.h"

int func(int num)
{
    return num * num + 1;
}

int main(int argc, char *argv[])
{
    for (int i = 0; i < 20; i++)
    {
        int pid = fork();
        if (pid == 0) // Child Process
        {
            printf(1, "Child Process %d\n", i + 1);
            printf(1, "Iteration | Matched | Different\n");

            for (int j = 0; j < 10; j++)
            {
                int *arr = malloc(4096); // 4 KB allocated space
                for (int k = 0; k < 1024; k++)
                {
                    arr[k] = func(k); // Iterating 1024 times as arr[k] is storing int which takes 4B memory
                }                     // so total memory filled = 4KB
                int match_count = 0;
                for (int k = 0; k < 1024; k++)
                {
                    if (arr[k] == func(k)) // Validating the content of the memory using the same function
                        match_count += 4;
                }

                if (j < 9)
                    printf(1, "    %d       %dB       %dB\n", j + 1, match_count, 4096 - match_count);
                else
                    printf(1, "   %d       %dB       %dB\n", j + 1, match_count, 4096 - match_count);
            }
            printf(1, "\n");
            exit(); // Terminating the child process
        }
    }
    while (wait() != -1)
        ;
    exit(); // Exiting the program only when no wait is required anymore. njmu7i8
}
```

In the user program **memtest,** we are supposed to create 20 child processes. So therefore in the outer loop we are storing the PID of the child process created using fork() call in the pid variable. Then if pid=0 then it means that it is a child process and thus then we will enter the if statements. Inside it as asked in the problem we are iterating 10 by using a for loop and in each iteration we are allocating 4KB memory space and storing the starting address of the allocated space in the variable **arr.** To fill this memory with values we have created the function **func(i)** which will return **i*i+1**. In each iteration we iterate 1024 times for assigning the values to the memory because as arr[k] is storing an integer value which takes 4B memory so repeating it for 1024 times will assign all 4096B of memory. Then we have created the variable **match_count** for validating the content of the memory. It will store the number of bytes storing the right value. So if the value stored at kth position is equal to the value returned by function func then increment match_count by 4. Then after printing out the result we are terminating this child process as we want only 20 child processes to be formed. Then finally we are exiting the program when no wait is required anymore. To run the memtest we need to include it in Makefile under UPROGS and EXTRA which will make it accessible to the xv6 user.



```
$ memtest
Child Process 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 Process 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 Process 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
```

The picture shown in the left is the result obtained on running memtest. The implementation passes the sanity test as we can see all 4096B memory stores the correct value inside them.

To check the paging mechanism we can reduce the PHYSTOP which will prevent the kernel from being able to hold all the processes memory in RAM. The default size of PHYSTOP is 224 MB(0XE000000). We are changing this value to 0X0400000 i.e. 4MB which is also the minimum memory required to execute kinit1. After this on again running the memtest, the newly obtained result was the same as the one obtained previously thus indicating that our implementation is correct.