CS - 344 (OS LAB)

Assignment - 3

Group-1

Members :

1. Parth Agarwal, CSE, 220101074
2. Shashwat Shankar, CSE, 220101092
3. Shobhit Gupta, CSE, 220101093
4. Shubhranshu Pandey, CSE, 220101094

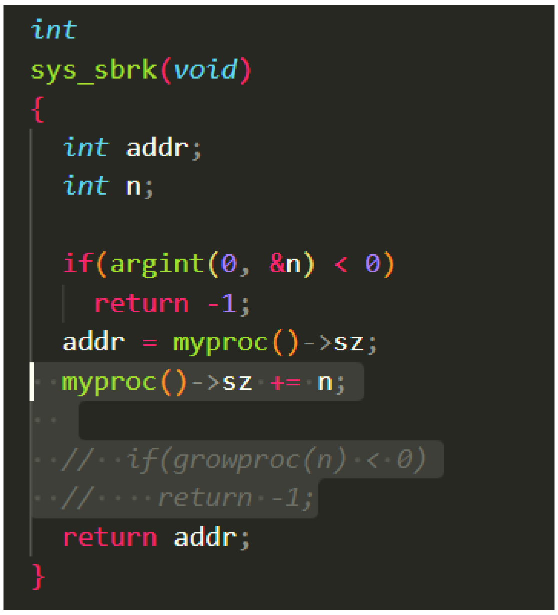
Drive Link :-

https://drive.google.com/drive/folders/1OqW4iWybr-oAx0GAJFntYrekIc
vCB39d?usp=sharing

PART-A: Lazy Memory Allocation

Task 1: Eliminate allocation from sbrk()

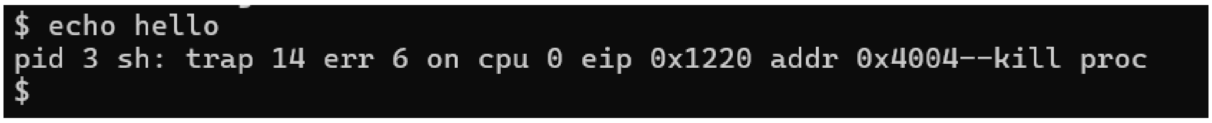
In this task we have made the following changes in sysproc.c using the patch file:



```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;
}
```

Now, sbrk(n) will just increment the process size by *"n"* and return the old size. It does not allocate memory, because the call to growproc() is commented out. However, it still increases proc->sz by n to trick the process into believing that it has the memory requested

On typing echo call the error it produced:



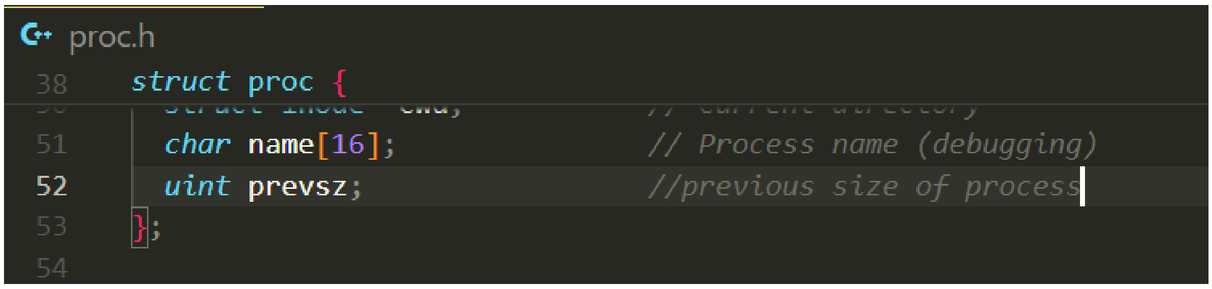
```
$ echo hello
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Task 2: Lazy Allocation

In this task we have made the following changes:

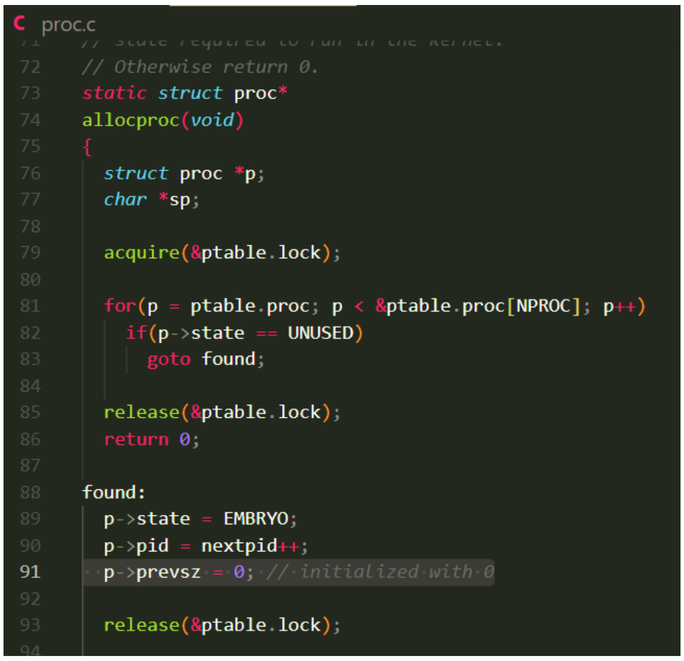
- **proc.h:** added unsigned int prevsz for storing the previous size of the process.



proc.h

```c
struct proc {
  char name[16];               // Process name (debugging)
  uint prevsz;                 //previous size of process
};
```

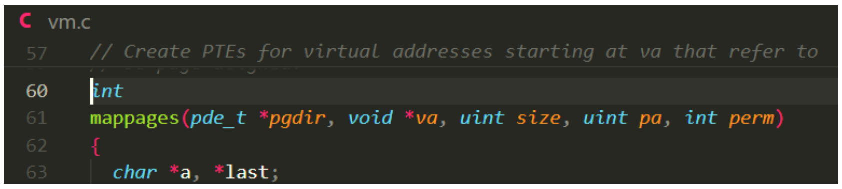
- **proc.c:**



```
proc.c
  // Otherwise return 0.
  static struct proc*
  allocproc(void)
  {
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      if(p->state == UNUSED)
        goto found;

    release(&ptable.lock);
    return 0;

  found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->prevsz = 0; // initialized with 0

    release(&ptable.lock);
```

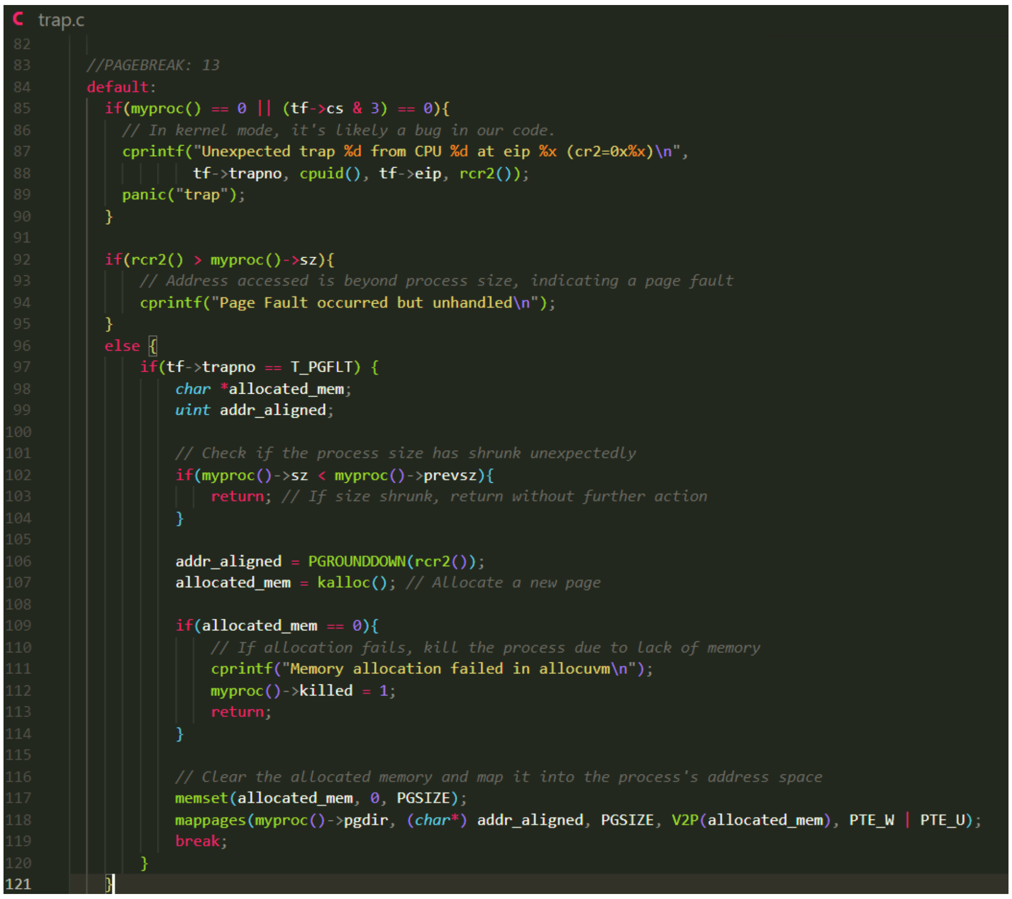
Initialized the prevsz variable with 0 inside the allocproc function.

- **vm.c:** Remove the static keyword for mappages function



```
vm.c
// Create PTEs for virtual addresses starting at va that refer to
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
```

- **trap.c:**

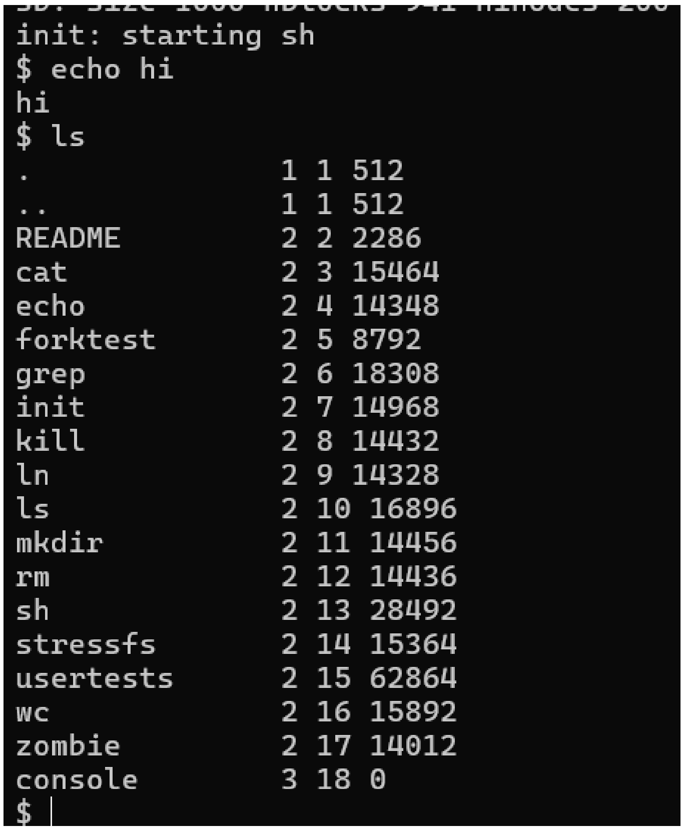


```
trap.c

  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs & 3) == 0){
      // In kernel mode, it's likely a bug in our code.
      cprintf("Unexpected trap %d from CPU %d at eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }

    if(rcr2() > myproc()->sz){
      // Address accessed is beyond process size, indicating a page fault
      cprintf("Page Fault occurred but unhandled\n");
    }
    else {
      if(tf->trapno == T_PGFLT) {
        char *allocated_mem;
        uint addr_aligned;

        // Check if the process size has shrunk unexpectedly
        if(myproc()->sz < myproc()->prevsz){
          return; // If size shrunk, return without further action
        }

        addr_aligned = PGROUNDDOWN(rcr2());
        allocated_mem = kalloc(); // Allocate a new page

        if(allocated_mem == 0){
          // If allocation fails, kill the process due to lack of memory
          cprintf("Memory allocation failed in allocuvm\n");
          myproc()->killed = 1;
          return;
        }

        // Clear the allocated memory and map it into the process's address space
        memset(allocated_mem, 0, PGSIZE);
        mappages(myproc()->pgdir, (char*) addr_aligned, PGSIZE, V2P(allocated_mem), PTE_W | PTE_U);
        break;
      }
    }
```

Add *“extern int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);”* and make the above changes in the trap function to handle lazy allocation.

In this modification to the default page trap, we handle page faults that originate from user space by allocating a new page of physical memory at the faulting address, allowing the process to continue execution. We begin by verifying that the trap is caused by a page fault with the condition *“(tf->trapno == T_PGFLT)”*. If *“kalloc()”* fails and returns 0, we bypass the cprintf statement by setting the process as killed and returning immediately.

After running the echo hi command we find that it is working properly:



```
init: starting sh
$ echo hi
hi
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15464
echo           2 4 14348
forktest       2 5 8792
grep           2 6 18308
init           2 7 14968
kill           2 8 14432
ln             2 9 14328
ls             2 10 16896
mkdir          2 11 14456
rm             2 12 14436
sh             2 13 28492
stressfs       2 14 15364
usertests      2 15 62864
wc             2 16 15892
zombie         2 17 14012
console        3 18 0
$ 
```

Here we also find that other commands (like ls) are also working properly.

PART-B: xv6 Memory Management

Answers to questions:

Q 1) How does the kernel know which physical pages are used and unused?

Ans. The kernel keeps track of used and unused physical pages using a free list, which contains all the available (free) physical pages. The physical memory manager in xv6 manages this list. Whenever a process allocates or deallocates memory, pages are removed from or added to this free list.

Q 2) What data structures are used to answer this question?

Ans. The primary data structure used to track physical page allocation in xv6 is the *struct run*, which represents each free memory block or page. A linked list of these *run* structures is maintained to represent the free list. Additionally, each process has its own page table, which maps virtual addresses to physical pages.

Q 3) Where do these reside?

Ans. These data structures reside in kernel memory. The free list is managed in the file *kalloc.c*, and page tables for individual processes reside in *proc->pgdir* (process directory) in *proc.h*.

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

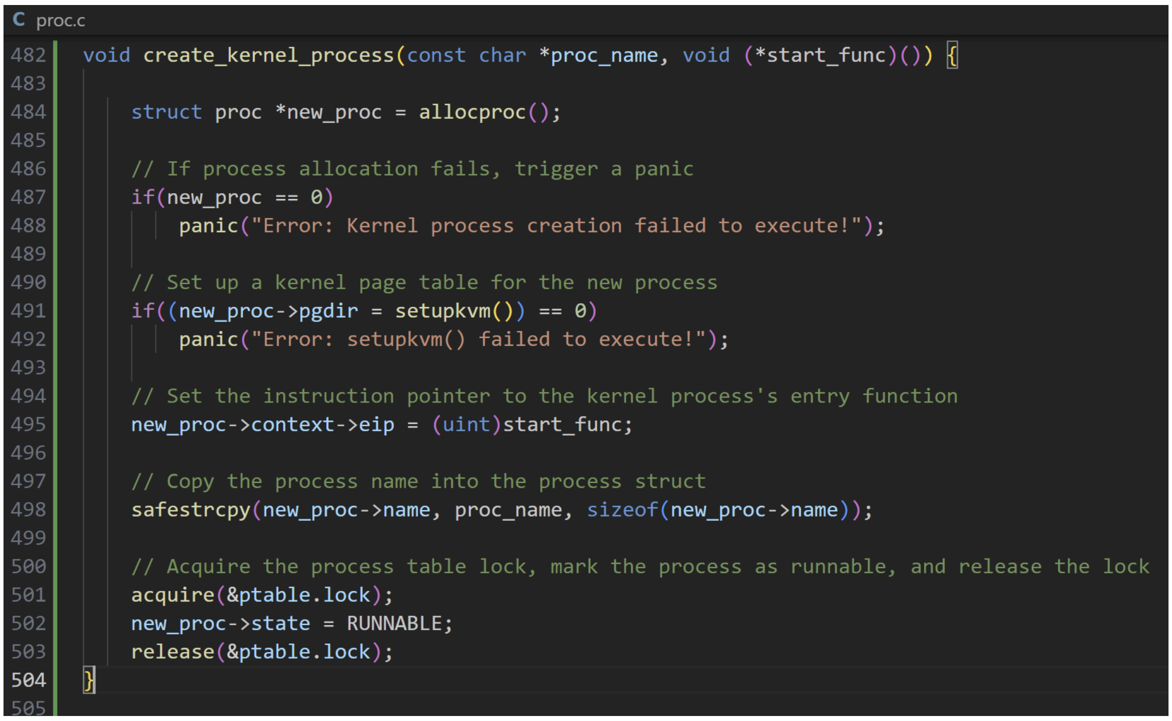
Ans. Yes, the xv6 memory mechanism limits the number of user processes. The limitation arises because xv6 uses a fixed-size memory model and the total amount of physical memory is limited. Each process requires memory for its page table and data, so the number of processes that can be created depends on the available physical memory.

Q5) 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 simultaneously is determined by the minimum memory needed to maintain one process, which includes the process control block (PCB), page tables, and other essential structures. If the physical memory is just enough to support a single process, then only one process can run at a time. However, this is theoretical as the kernel itself always requires some memory.

Task 1: Kernel Processes

- **proc.c**
  - create_kernel_process() added to proc.c for kernel-only processes.
  - No need to initialize trapframe, user space, or user page table.
  - The eip register holds the address of the next instruction.
  - Set the eip in the context to the entry point (function pointer).
  - allocproc assigns the process to the process table (ptable).
  - setupkvm sets up the kernel page table, mapping virtual addresses above KERNBASE to physical addresses between 0 and PHYSTOP.



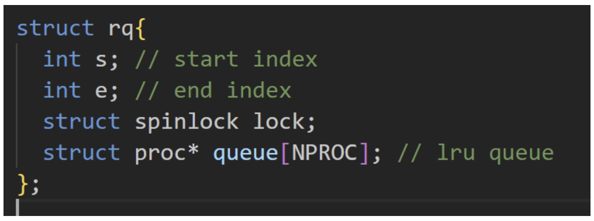
proc.c

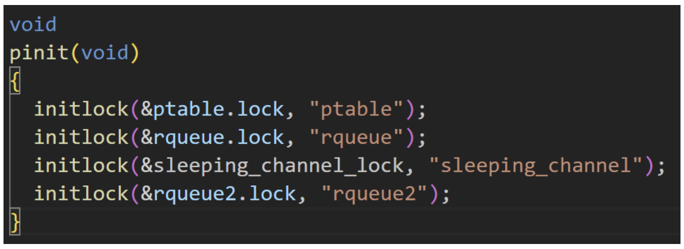
```c
void create_kernel_process(const char *proc_name, void (*start_func)()) {

    struct proc *new_proc = allocproc();

    // If process allocation fails, trigger a panic
    if(new_proc == 0)
        panic("Error: Kernel process creation failed to execute!");

    // Set up a kernel page table for the new process
    if((new_proc->pgdir = setupkvm()) == 0)
        panic("Error: setupkvm() failed to execute!");

    // Set the instruction pointer to the kernel process's entry function
    new_proc->context->eip = (uint)start_func;

    // Copy the process name into the process struct
    safestrcpy(new_proc->name, proc_name, sizeof(new_proc->name));

    // Acquire the process table lock, mark the process as runnable, and release the lock
    acquire(&ptable.lock);
    new_proc->state = RUNNABLE;
    release(&ptable.lock);
}
```

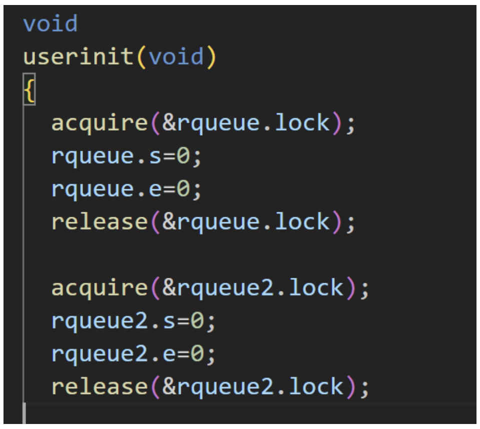
Task 2: Swapping Out Mechanism

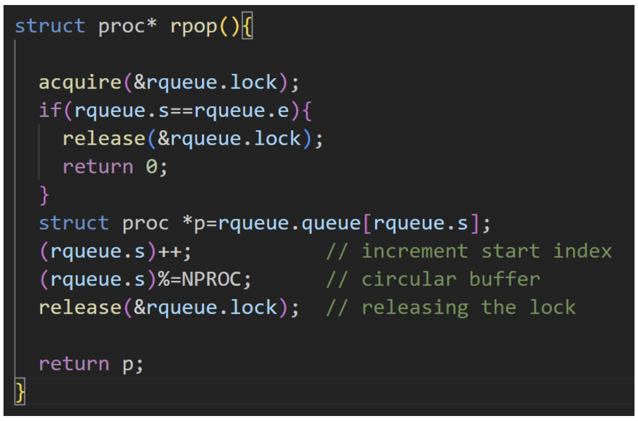
- **proc.c:**
  - A process queue (rqueue) was created to track processes denied additional memory due to lack of free pages.
  - A circular queue struct (rq) was defined.
  - Functions rpush() and rpop() were implemented for queue operations.
  - A lock for queue access was initialized in pinit.
  - The initial values of s and e were set to zero in userinit.
  - Prototypes for the queue and related functions were added to defs.h for use in other files.

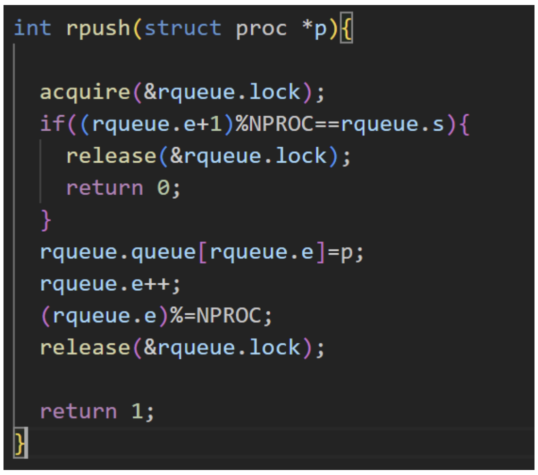
-


```c
struct rq{
  int s; // start index
  int e; // end index
  struct spinlock lock;
  struct proc* queue[NPROC]; // lru queue
};
```

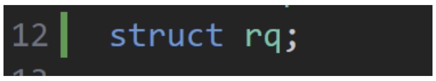

```c
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&rqueue.lock, "rqueue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&rqueue2.lock, "rqueue2");
}
```

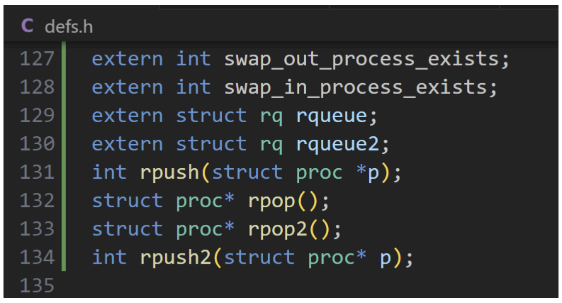

```c
void
userinit(void)
{
  acquire(&rqueue.lock);
  rqueue.s=0;
  rqueue.e=0;
  release(&rqueue.lock);

  acquire(&rqueue2.lock);
  rqueue2.s=0;
  rqueue2.e=0;
  release(&rqueue2.lock);
```


```c
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)++;             // increment start index
  (rqueue.s)%=NPROC;        // circular buffer
  release(&rqueue.lock);   // releasing the lock

  return p;
}
```


```c
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;
}
```

- **Defs.h:**


```c
struct rq;
```

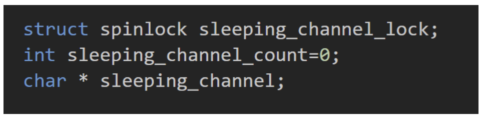


```c
extern int swap_out_process_exists;
extern int swap_in_process_exists;
extern struct rq rqueue;
extern struct rq rqueue2;
int rpush(struct proc *p);
struct proc* rpop();
struct proc* rpop2();
int rpush2(struct proc* p);
```

- When kalloc fails to allocate pages, it returns 0, signaling allocuvm that the memory request wasn't fulfilled.
- The process state is changed to sleeping on a special sleeping channel, sleeping_channel, secured by the sleeping_channel_lock.
- sleeping_channel_count is used for corner cases during system boot.
- The current process is added to the swap out request queue, rqueue.
- Global declarations for these variables are made in vm.c (also declared in defs.h as extern variables).

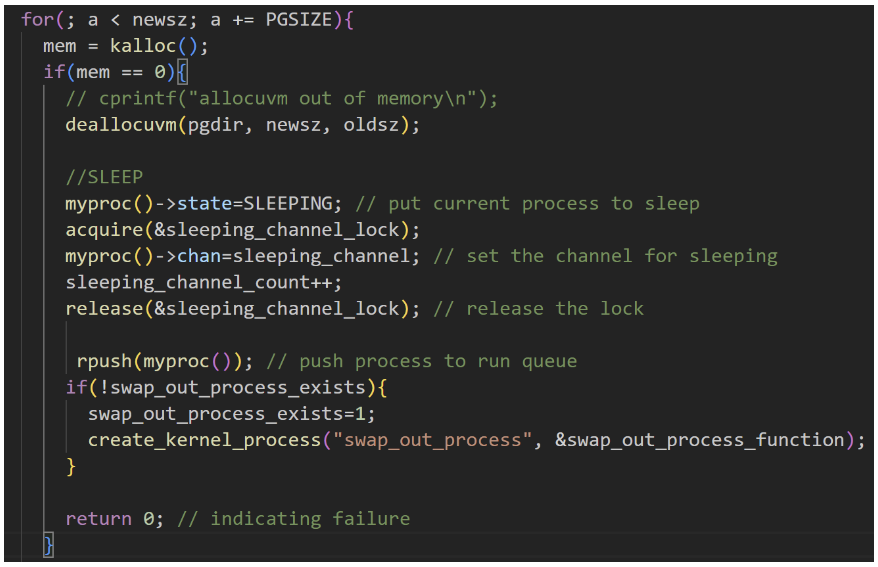
- **Vm.c:**

  Global declarations (Note: These are also declared in defs.h as extern variables. I am not adding the defs.h screenshots):



```
struct spinlock sleeping_channel_lock;
int sleeping_channel_count=0;
char * sleeping_channel;
```

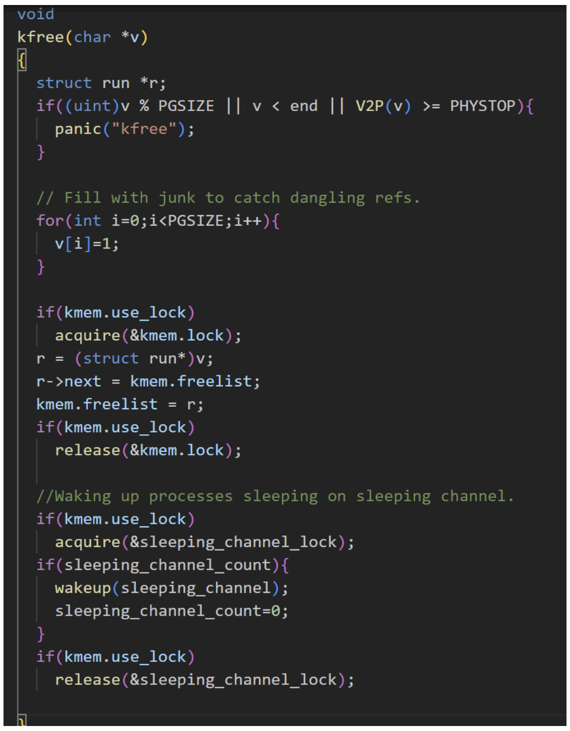
- **allocuvm:**



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

    //SLEEP
    myproc()->state=SLEEPING; // put current process to sleep
    acquire(&sleeping_channel_lock);
    myproc()->chan=sleeping_channel; // set the channel for sleeping
    sleeping_channel_count++;
    release(&sleeping_channel_lock); // release the lock

     rpush(myproc()); // push process to run queue
    if(!swap_out_process_exists){
      swap_out_process_exists=1;
      create_kernel_process("swap_out_process", &swap_out_process_function);
    }

    return 0; // indicating failure
  }
```

- create_kernel_process creates a swap-out kernel process to allocate a page if none exists for the process.
- When the swap-out process finishes, the swap_out_process_exists variable (declared as extern in defs.h and initialized to 0 in proc.c) is reset to 0.
- When the swap-out process is created, swap_out_process_exists is set to 1 to prevent multiple swap-out processes from being created.
- More details on the swap_out_process are explained later.

- **kalloc.c:**



```c
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.
  for(int i=0;i<PGSIZE;i++){
    v[i]=1;
  }

  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);

  //Waking up processes sleeping on sleeping channel.
  if(kmem.use_lock)
    acquire(&sleeping_channel_lock);
  if(sleeping_channel_count){
    wakeup(sleeping_channel);
    sleeping_channel_count=0;
  }
  if(kmem.use_lock)
    release(&sleeping_channel_lock);

}
```

A mechanism is added to wake up processes when free pages become available.

In kfree (in kalloc.c), the system is modified to wake all processes sleeping on sleeping_channel.

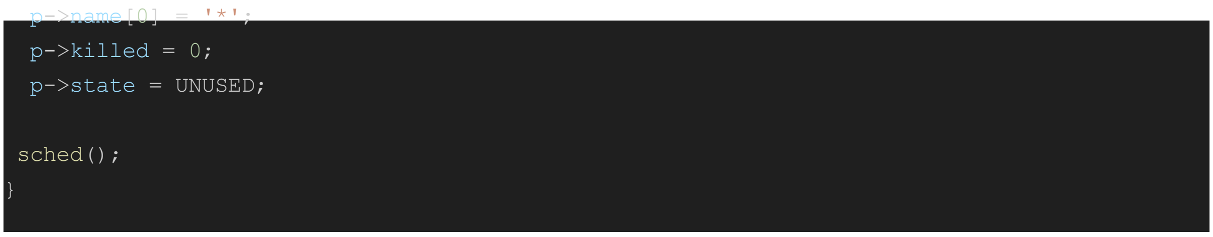
These processes were previously put to sleep due to page unavailability.

The wakeup() system call is used to wake all processes currently sleeping on sleeping_channel.

- **proc.c:**

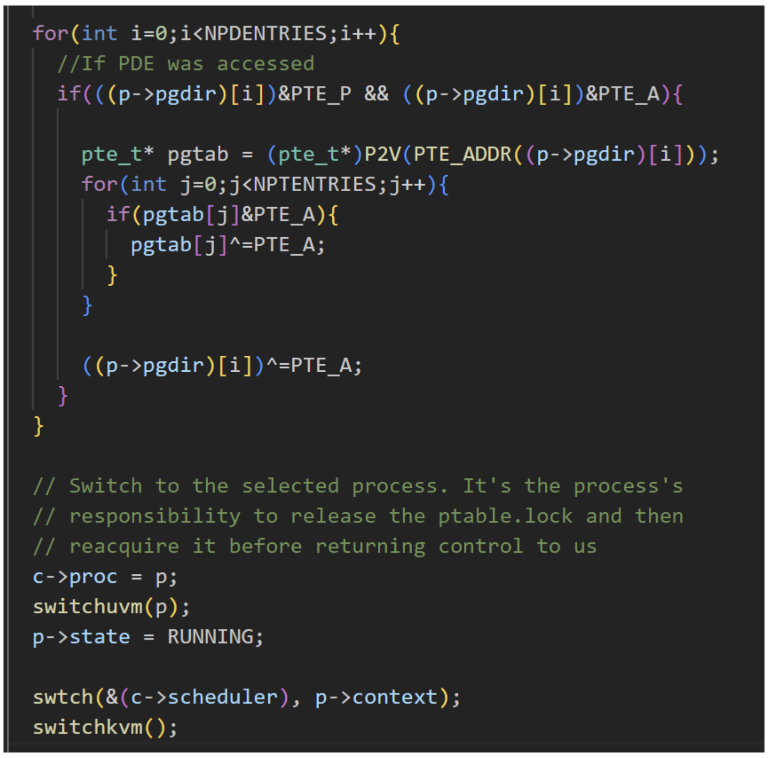
Now in proc.c we will add the **swap_out_process** function responsible for **swapping out process**:

```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 high, not every page table was
       accessed.
      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);

        // 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;

 p->parent = 0;
```



```
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;

 sched();
}
```

- The swap-out process runs in a loop until the swap-out request queue (rqueue1) is empty.
- When rqueue1 is empty, specific termination instructions are executed (explained in Image 2).
- The loop pops the first process from rqueue and applies the LRU policy to evaluate each entry in the process's page table (pgdir).
- The physical address is extracted for each secondary page table.
- For each secondary page table, the accessed bit (A) is checked for each entry.
- The accessed bit is the 6th bit from the right and is checked using bitwise & with PTE_A (defined as 32 in mmu.c).
- Accessed flag note: The accessed bits are cleared during context switching by the scheduler.
- This ensures the accessed bit seen by swap_out_process_function indicates whether the entry was accessed in the last process iteration.



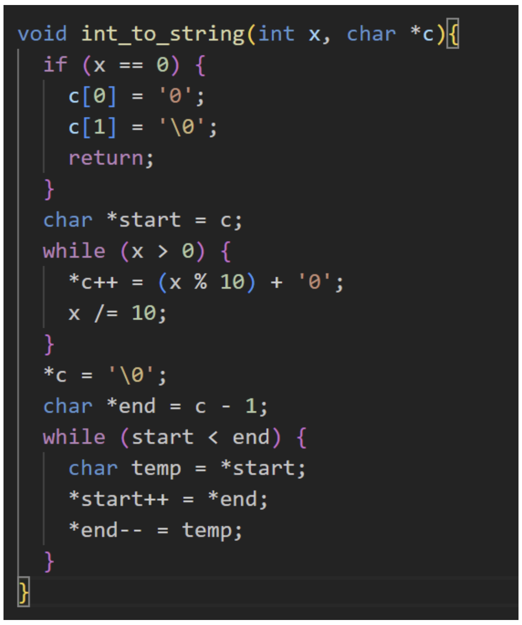
```
for(int i=0;i<NPDENTRIES;i++){
  //If PDE was accessed
  if(((p->pgdir)[i])&PTE_P && ((p->pgdir)[i])&PTE_A){

    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;
  }
}

// Switch to the selected process. It's the process's
// responsibility to release the ptable.lock and then
// reacquire it before returning control to us
c->proc = p;
switchuvm(p);
p->state = RUNNING;

swtch(&(c->scheduler), p->context);
switchkvm();
```

The scheduler code unsets every accessed bit in the process's page table and its secondary page tables.

In **swap_out_process_function**, if a secondary page table entry has the accessed bit unset, it is selected as the victim page.

The victim page is swapped out and stored on the drive.

The filename for the stored page is generated using the process's **PID** and the **virtual address of the page**.

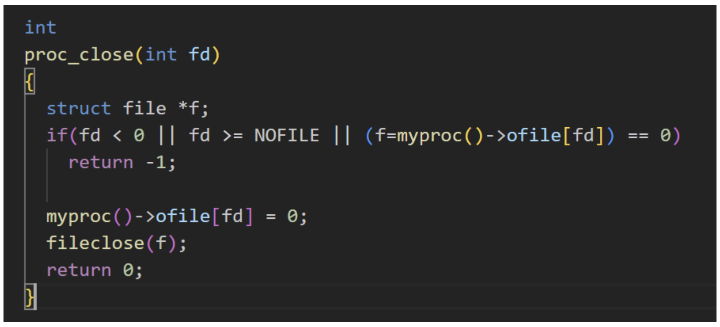


```c
void int_to_string(int x, char *c){
  if (x == 0) {
    c[0] = '0';
    c[1] = '\0';
    return;
  }
  char *start = c;
  while (x > 0) {
    *c++ = (x % 10) + '0';
    x /= 10;
  }
  *c = '\0';
  char *end = c - 1;
  while (start < end) {
    char temp = *start;
    *start++ = *end;
    *end-- = temp;
  }
}
```

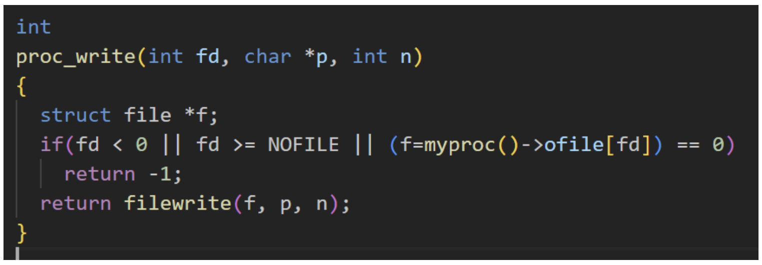
A new function, **int_to_string**, copies integers to a string as we need to create the filename in the format **<pid>_<virt>.swp.**

Since file system calls cannot be made from proc.c, functions like open, write, read, and close were copied from sysfile.c, modified for argument compatibility, and renamed to **proc_open, proc_read, proc_write, and proc_close**.

Examples:



```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;
}
```



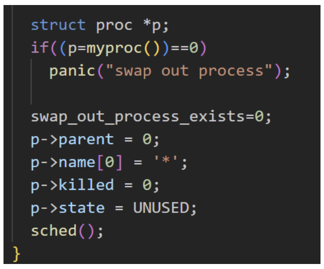
```c
int
proc_write(int fd, char *p, int n)
{
  struct file *f;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  return filewrite(f, p, n);
}
```

- Using these functions, a page is written back to storage.
- A file is opened with proc_open using **O_CREATE** and **O_RDWR** permissions (defined in fcntl.h).
- **O_CREATE** creates the file if it doesn't already exist.
- **O_RDWR** refers to read/write permissions; the file descriptor is stored in an integer called fd.
- The page is written to the file using **proc_write** with the file descriptor.
- The page is added to the free page queue using **kfree,** making it available for use.
- Processes sleeping on **sleeping_channel** are woken up when a page is added to the free queue.

- The corresponding page table entry is cleared using **memset**.
- To track whether the page that caused a fault was swapped out, the 8th bit from the right (2^7) in the secondary page table entry is set using XOR (LINE 295 in the image).
- Suspending the kernel process:
    - When the queue is empty, the loop breaks, initiating the process suspension.
    - The kernel process's stack cannot be cleared from within the process, as it would lose context on the next process to execute.
    - The process is preempted and the scheduler is waited upon to find it.
    - When the scheduler finds a kernel process in the **UNUSED state**, it clears the process's stack and name.
    - The scheduler identifies a kernel process in the unused state by checking its name, which starts with **'*'** when the process ended.

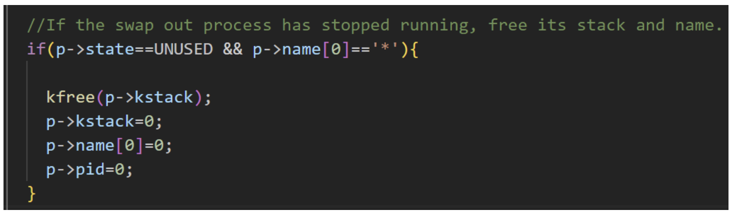
Thus the ending of kernel processes has two parts:

1. From within process



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

  swap_out_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

2. From scheduler



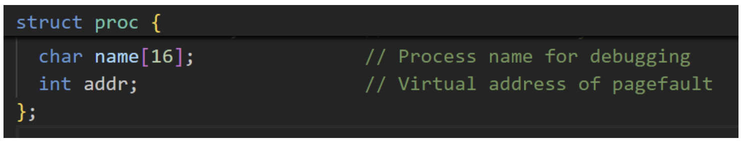
```
  //If the swap out process has stopped running, free its stack and name.
  if(p->state==UNUSED && p->name[0]=='*'){

    kfree(p->kstack);
    p->kstack=0;
    p->name[0]=0;
    p->pid=0;
  }
```

- Swapping out process supports a request queue for swap requests (refer to pages 1 and 2 of this report).
- The swapping out process is suspended when there are no pending requests (see page 5 of this report, just above this section).
- When at least one free physical page is available, all processes suspended due to a lack of physical memory are woken up (discussed on page 3 of this report regarding **kfree** and **sleeping_channel**).
- Only user-space memory can be swapped out (the second-level page table is excluded). The first user-space page that was not accessed in the last

iteration is swapped out since we iterate through the top tables from top to bottom, with user-space entries coming first (up to **KERNBASE**).

Task 3: Swapping In Mechanism

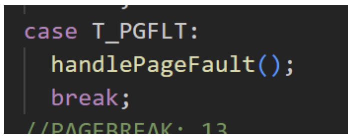
- **Proc.h:**
    - A swap-in request queue called rqueue2 is created using the same struct (rq) as in Task 2 in proc.c.
    - An extern prototype for rqueue2 is declared in defs.h.
    - Functions rpop2() and rpush2() are created in proc.c, with their prototypes declared in defs.h.
    - The lock for rqueue2 is initialized in pinit.
    - The s and e variables for rqueue2 are initialized in userinit.
    - An additional entry, addr (int), is added to the struct proc in proc.h to indicate the virtual address at which the page fault occurred.



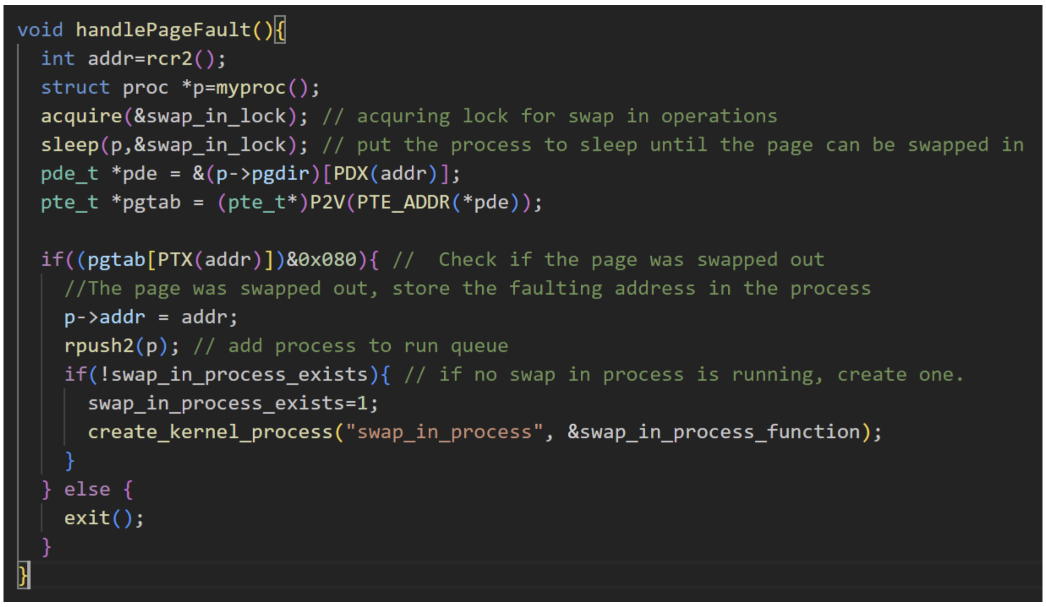
```
struct proc {
  char name[16];               // Process name for debugging
  int addr;                    // Virtual address of pagefault
};
```

Trap.c:

Handling page fault (T_PGFLT) traps is implemented in a function called handlePageFault() in trap.c.



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

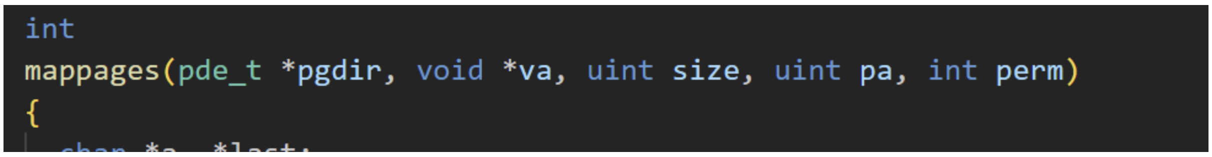


```
void handlePageFault(){
  int addr=rcr2();
  struct proc *p=myproc();
  acquire(&swap_in_lock); // acquring lock for swap in operations
  sleep(p,&swap_in_lock); // put the process to sleep until the page can be swapped in
  pde_t *pde = &(p->pgdir)[PDX(addr)];
  pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

  if((pgtab[PTX(addr)])&0x080){ //  Check if the page was swapped out
    //The page was swapped out, store the faulting address in the process
    p->addr = addr;
    rpush2(p); // add process to run queue
    if(!swap_in_process_exists){ // if no swap in process is running, create one.
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

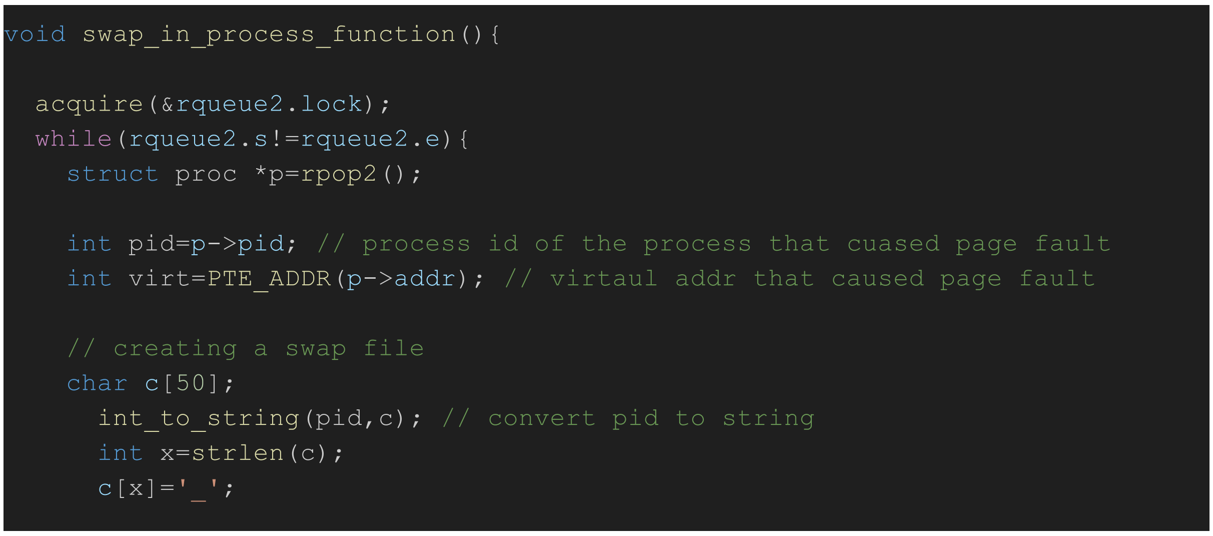
- In handlePageFault(), the virtual address of the page fault is found using rcr2().
- The current process is put to sleep with a new lock called swap_in_lock (initialized in trap.c and declared as extern in defs.h).
- The page table entry corresponding to the faulting address is obtained (using logic similar to walkpgdir).

- To check if the page was swapped out, the 7th order bit (2^7) of the page table entry is checked using bitwise & with 0x080.
- If the bit is set, the swap-in process is initiated (if it doesn't already exist, checked with swap_in_process_exists).
- If the bit is not set, the process is safely suspended using exit() as required by the assignment.
- The entry point for the swapping in process is swap_in_process_function (declared in proc.c), which is detailed on the next page.
- The function runs a loop until rqueue2 is not empty:
- In each iteration, it pops a process from the queue and extracts its PID and addr value to construct the filename.
- The filename is created using int_to_string (described in Task 2, page 4).
- The file is opened in read-only mode (0_RDONLY) using proc_open, with the file descriptor stored in fd.
- A free frame (mem) is allocated for the process using kalloc.
- The file is read from the disk using the relevant function.
- The file descriptor (fd) is read into the free frame using proc_read.
- mappages is made accessible to proc.c by removing the static keyword in vm.c and declaring its prototype in proc.c.
- mappages is then used to map the page corresponding to addr with the physical page allocated with kalloc and read into (mem).
- The process for which a new page was allocated is woken up to resolve the page fault using wakeup.
- Once the loop is completed, the kernel process termination instructions described on page 5 are executed.
- All checkmarks for Task 3 have been accomplished.

In **vm.c** we removed the static keyword from the mappages function



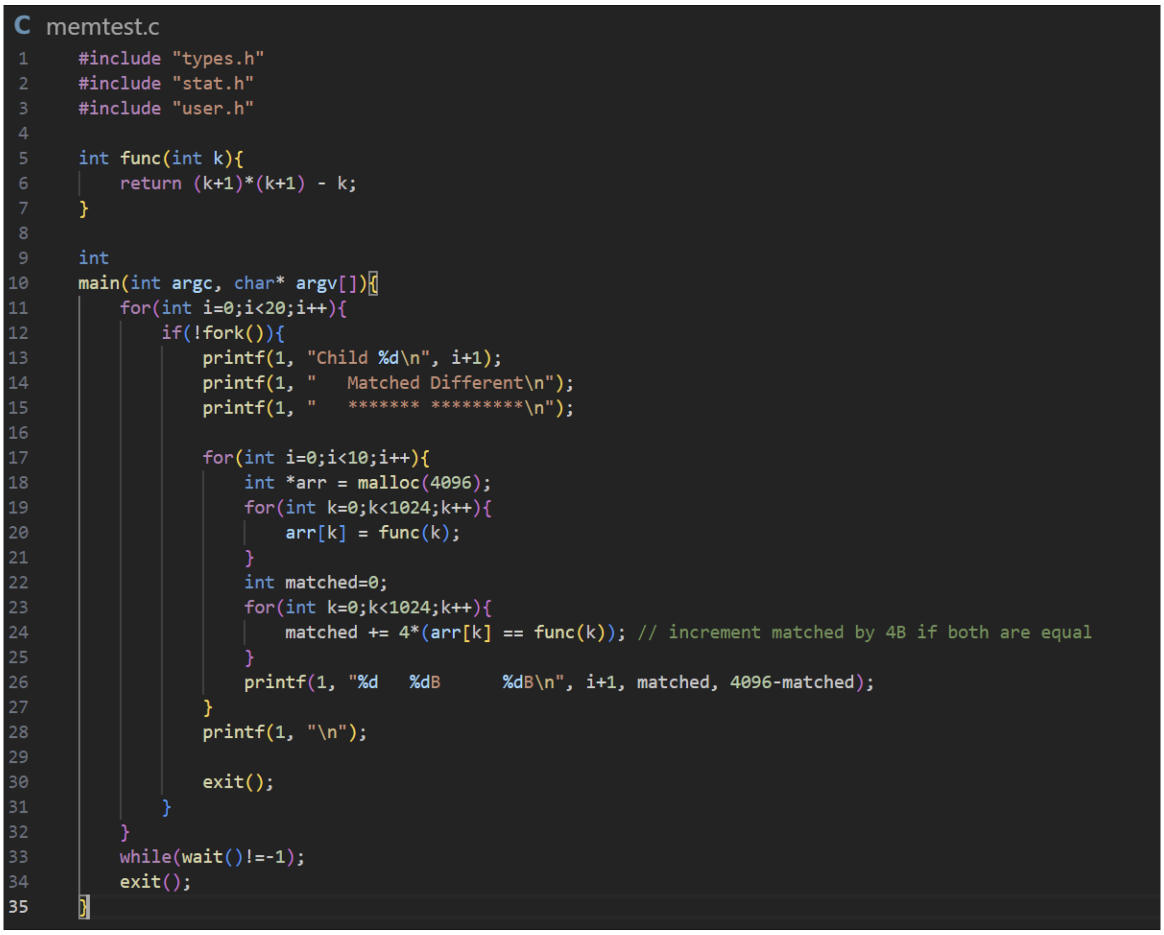
```
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
```



```
void swap_in_process_function(){

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

    int pid=p->pid; // process id of the process that cuased page fault
    int virt=PTE_ADDR(p->addr); // virtaul addr that caused page fault

    // creating a swap file
    char c[50];
      int_to_string(pid,c); // convert pid to string
      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); /// opening the swap file
      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;
  if((p=myproc())==0)
    panic("swap_in_process");

  swap_in_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();

}
```

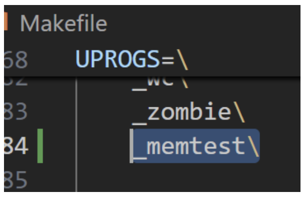
Task 4: Sanity Test

- This aims to create a testing mechanism to verify the functionalities developed in previous tasks.
- A user-space program named memtest will be implemented for this purpose.
- The implementation details of memtest are provided below.

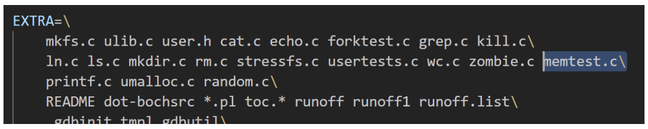


```c
memtest.c
#include "types.h"
#include "stat.h"
#include "user.h"

int func(int k){
    return (k+1)*(k+1) - k;
}

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

            for(int i=0;i<10;i++){
                int *arr = malloc(4096);
                for(int k=0;k<1024;k++){
                    arr[k] = func(k);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    matched += 4*(arr[k] == func(k)); // increment matched by 4B if both are equal
                }
                printf(1, "%d   %dB      %dB\n", i+1, matched, 4096-matched);
            }
            printf(1, "\n");

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

- To run memtest, it needs to be included in the Makefile under the UPROGS and EXTRA sections to make it accessible to the XV6 user.

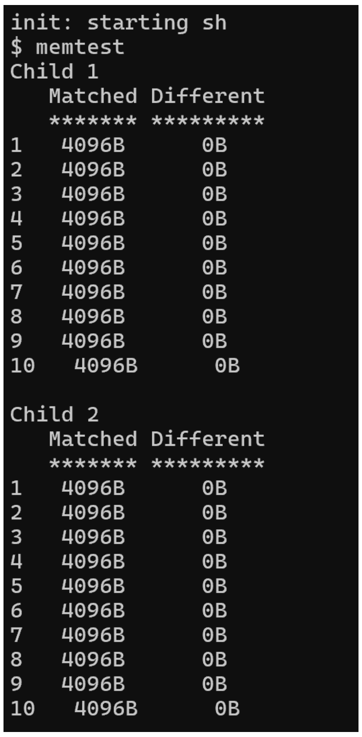


```
Makefile
68  UPROGS=\
        _wc\
83      _zombie\
84      _memtest\
85
```



```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c memtest.c\
    printf.c umalloc.c random.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

- Upon running memtest, the following output is obtained:
- The output shows that the implementation passes the **sanity** test, as all indices store the correct values.
- To further validate the implementation, tests are run with different values of PHYSTOP (defined in memlayout.h).
- The default value of PHYSTOP is 0XE000000 (224MB), which is changed to 0x0400000 (4MB).
- 4MB is selected as it is the minimum memory required by XV6 to execute kinit1.



```
init: starting sh
$ memtest
Child 1
   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
   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
```

- Upon running memtest with the new value, the output remains identical to the previous output, confirming that the implementation is correct.