CS 344 Assignment 3

Group M20
Aditya Pandey-200123004
Aman Bucha-200123006
Sarthak Diwan-200123076

Part A : Lazy Memory Allocation

1)

The sbrk system call is used by the current process to notify the xv6 OS whenever it requires more memory than it has been allotted. To meet this need, sbrk employs the growproc() function that is described in proc.c. A deeper look at the growproc() implementation reveals that growproc() calls allocuvm(), which is in charge of creating the extra pages and mapping the virtual addresses to their corresponding physical locations inside page tables in order to allocate the extra memory that is required.

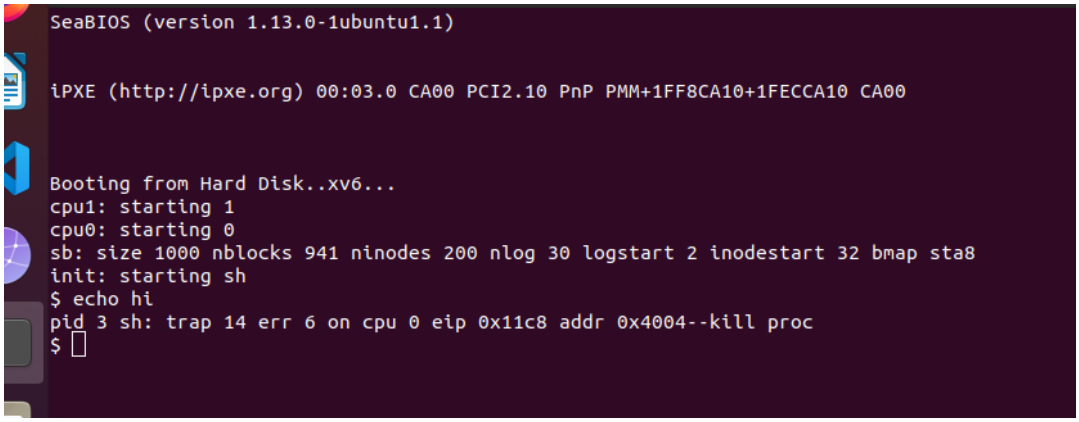
Our goal in this assignment is to hold back memory when it is required. Instead, we provide the memory upon access. The term "lazy memory allocation" refers to this. To achieve this, we comment out the sbrk system function's call to growproc().

```
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;
}
```

We alter the current process's size variable to the desired value, giving the impression that memory has been allotted to the process.
This process experiences a PAGE FAULT when it attempts to access the page (which it believes has already been brought into memory), resulting in a T_PGFLT trap being sent to the kernel.
Thus when we give the command ‘echo hi’ we get the following error:



By invoking handlePagefault in trap.c, this is handled ().

```
case T_PGFLT:
  if(handlePageFault()<0){
    cprintf("Could not allocate page. Sorry.\n");
    panic("trap");
  }
break;
```

rcr2() provides the virtual address where the page fault takes place in this. rounded _addr points to the page's beginning address where this virtual address is located. After that, we use the system function kalloc(), which retrieves a free page from a linked list of free pages (freelist inside kmem). We now have access to a physical page. Now, using mappages(), we must map it to the virtual address rounded_addr . We remove the static keyword from front of mappages() in vm.c and declare its prototype in trap.c in order to use it there.

```
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;
}
```

```
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

So now, when we run the command ‘echo hi’ we see that the command functions properly:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ echo hi
hi
$
```

Questions

I. How does the kernel know which pages are used and unused?
Xv6 maintains a list of free pages as a linked list in the kmem structure. ,

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

kinit1() initially allocates empty pages worth 4MB to this free list.

II. What data structures are used to answer these questions?
The *freelist is a linked list of free pages which are represented in the form of struct run.

III. Where do these reside?

This structure is declared inside kalloc.c inside a structure kmem.

IV. Does xv6 memory mechanism limit the number of processes?
Due to a limit on the size of ptable, a max. of NPROC = 64 processes can exist simultaneously. NPROC is defined in param.h.

V. If so, what is the lowest number of processes xv6 can ‘have’ at the same time (assuming the kernel requires no memory whatsoever)
There can be a minimum of 1 process running on xv6. All user interactions need to be done using user processes forked from initproc/sh.

Part B

Task 1

In proc.c, a function called create_kernel_process() was written. The kernel process will continue to operate in kernel mode throughout. Its trapframe, user space, and user part of its page table do not need to be initialized because trap frames contain user space register values. The address of the following instruction is kept in the eip register of the process context. At the entrance point, the process should begin to run (which is a function pointer).
As a result, we changed the context's eip value to entry point (Since entry point is the address of a function).
Allocproc allocates a space for the process in the ptable. To map virtual addresses above KERNBASE to physical locations between 0 and PHYSTOP, setupkvm configures the kernel portion of the process' page table.

```
void
create_kernel_process(const char *name, void (*entrypoint)())
{
  struct proc *p;

  // Allocate process.
  if((p = allocproc()) == 0){
    return;
  }

  // Setup page table.
  if((p->pgdir = setupkvm()) == 0)
  {
    kfree(p->kstack);
    p->kstack = 0;
    p->state = UNUSED;
    panic("kernel process: out of memory?");
  }

  // Specify other parameters
  p->sz = PGSIZE;
  p->parent = initproc;

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

  acquire(&ptable.lock);

  p->context->eip = (uint)entrypoint;
  p->state = RUNNABLE;

  release(&ptable.lock);

  return;
}
```

Task 2

It consists of several components. The first thing we require is a process queue that records the processes that were rejected additional memory since there were no empty pages. A circular queuing structure was made, known as rq. And rqueue is the particular queue that contains processes with swap out processes. Additionally, we made the rq-corresponding functions, rpush() and rpop (). The queue needs to be accessed with a lock that pinit has initialized. Additionally, we set the starting values of s and e in userinit to zero. We also introduced prototypes in defs.h because the queue and the functions related to it are required in other files.

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

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

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

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

Now, kalloc returns zero every time it is unable to allot pages to a process. This informs allocuvm that no memory was allocated for the required amount (mem=0). In this case, we must first set the process status to sleeping. (*Note: The process sleeps on a unique channel named sleeping channel, which is protected by a lock called sleeping channel lock. When the system boots, sleeping channel count is used for edge scenarios.) Next, we must add the running process to the rqueue for swap out requests.

*Note: create_kernel_process here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. When the swap out process ends, the swap_out_process_exists (declared as extern in defs.h and initialized in proc.c to 0) variable is set to 0. When it is created, it is set to 1 (as seen above). This is done so multiple swap out processes are not created. swap_out_process is explained later.
Next, we create a mechanism by which whenever free pages are available, all the processes sleeping on sleeping_channel are woken up. We edit kfree in kalloc.c in the following way:

Basically, all processes that were preempted due to lack of availability of pages were sent sleeping on the sleeping channel. We wake all processes currently sleeping on `sleeping_channel` by calling the `wakeup()` system call.

```
void
kfree(char *v)
{

  struct run *r;
  // struct proc *p=myproc();

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

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

  //Wake 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);

}
```

```
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);
```

Now, I will explain the swapping out process. The entry point for the swapping out process in swap_out_process_function. Since the function is very long, I have attached two screenshots:

Until the swap out requests queue (rqueue1) is not empty, the procedure iterates in a loop. A set of instructions are carried out to end the swap out process when the queue is empty (Image 2). The loop begins by removing the first process from the rqueue, then it searches its page table for a victim page using the LRU policy. The physical address for each secondary page table is extracted by repeatedly going through each entry in the process page table (pgdir). The accessed bit (A), which is the sixth bit from the right on each entry in the page table for each secondary page table, is examined throughout each iteration of the page table.
By comparing the bitwise & of the entry and PTE_A (which we defined as 32 in mmu.c), we can determine if it is set. Important information regarding the Accessed flag: All accessed bits are unset whenever the scheduler switches the context of the process. We are doing this, so the accessed bit detected by the swap out process function will show whether the item was accessed in the previous round of the process.
This code resides in the scheduler and it basically unsets every accessed bit in the process page table and its secondary page tables.

Returning to the swap_out_process_function now. The victim page is selected by the function (using the macros stated in section A report) as soon as it locates a secondary page table entry with the accessed bit unset. The following step is to replace and store this page on drive.

```
        // 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 file containing this page is named using the process pid and virtual address of the page that is to be deleted. A new function we call "int_to_string" copies an integer into a specified string. We create the filename using the integers pid and virt using this function. This is the declaration for that function in proc.c:

We need to write the contents of the victim page to the file with the name _.swp. But we encounter a problem here. We store the filename in a string called c. File system calls cannot

be called from proc.c. The solution was that we copied the open, write, read, close etc. functions from sysfile.c to proc.c, modified them since the sysfile.c functions used a different way to take arguments and then renamed them to proc_open, proc_read, proc_write, proc_close etc. so we can use them in proc.c.

Some examples:

```
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);
}
```

```
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;
}
```

There are many more functions (proc_open, proc_fdalloc etc.) and you can check them out in proc.c.
Now, using these functions, we write back a page to storage. We open a file (using proc_open) with O_CREATE and O_RDWR permissions (we have imported fcntl.h with these macros). O_CREATE creates this file if it doesn't exist and O_RDWR refers to read/write. The file descriptor is stored in an integer called fd. Using this file descriptor, we write the page to this file using proc_write.
Then, this page is added to the free page queue using kfree so it is available for use (remember we also wake up all processes sleeping on sleeping_channel when kfree adds a page to the free queue). We then clear the page table entry too using memset.

Suspending kernel process when no requests are left:
When the queue is empty, the loop breaks and suspension of the process is initiated. While exiting the kernel processes that are running, we can't clear their kstack from within the process because after this, they will not know which process to execute next. We need to clear their kstack from outside the process. For this, we first preempt the process and wait for the scheduler to find this process.

When the scheduler finds a kernel process in the UNUSED state, it clears this process kstack and name. The scheduler identifies the kernel process in unused state by checking its name in which the first character was changed to "*" when the process ended.

All check marks in assignment accomplished:
1 . Note: the swapping out process must support a request queue for the swapping requests. (page 1 and 2 of this report).
2 . Note: whenever there are no pending requests for the swapping out process, this process must be suspended from execution. (page 5 of this report, just above this)
3 . Note: whenever there exists at least one free physical page, all processes that were suspended due to lack of physical memory must be woken up. (page 3 of this report: kfree and sleeping_channel)
4 . Note: only user-space memory can be swapped out (this does not include the second level page table) (since we are iterating all top tables from to bottom and all user space entries come first (until KERNBASE), we will swap out the first user space page that was not accessed in the last iteration.)

Task 3:

We first need to create a swap in request queue. We used the same struct (rq) as in Task 2 to create a swap in request queue called rqueue2 in proc.c. We also declare an extern prototype for rqueue2 in defs.h. Along with declaring the queue, we also created the corresponding functions for rqueue2 (rpop2() and rpush2()) in proc.c and declared their prototype in defs.h. We also initialised its lock in pinit. We also initialised its s and e variables in userinit.
Since all the functions/variables are similar to the ones shown in Task 2, I am not attaching their screenshots here.
Next, we add an additional entry to the struct proc in proc.h called addr (int). This entry will tell the swapping in function at which virtual address the page fault occurred: proc.h (in struct proc):

Next, we add an additional entry to the struct proc in proc.h called addr (int). This entry will tell the swapping in function at which virtual address the page fault occurred: proc.h (in struct proc):

Next, we need to handle page fault (T_PGFLT) traps raised in trap.c. We do it in a function called handlePageFault():
Trap.c:

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

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

In handlePageFault, 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 in trap.c and with extern in defs.h). We then obtain the page table entry corresponding to this address (the logic is identical to walkpgdir). 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 entries bit of 7th order (2^7). This is mentioned at the beginning of the 5th page of this report. Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using bitwise & with 0x080. If it is set, we initiate swap_in_process (if it doesn't already exist - check using swap_in_process_exists). Otherwise, we safely suspend the process using exit() as the assignment asked us to do.

Now, we go through the swapping in process. The entry point for the swapping out process is swap_in_process_function (declared in proc.c) as you can see in handlePageFault. Note: swap_in_process_function is shown on the next page since it is long. Refer to the next page for the actual function.

I have already mentioned how we have implemented file management functions in proc.c in the Task 2 part of the report. I will just mention which functions I used and how I used them here. The function runs a loop until rqueue2 is not empty. In the loop, it pops a process from the queue and extracts its pid and addr value to get the file name. Then, it creates the filename in a string called "c" using int_to_string (described in Task 2, page 4 of this report). Then, it used proc_open to open this file in read only mode (O_RDONLY) with file descriptor fd. We then allocate a free frame (mem) to this process using kalloc.

We read from the file with the fd file descriptor into this free frame using proc_read. We then make mappages available to proc.c by removing the static keyword from it in vm.c and then declaring a prototype in proc.c. We then use mappages to map the page corresponding to addr with the physical page that got using kalloc and read into (mem). Then we wake up, the process for which we allocated a new page to fix the page fault using wakeup. Once the loop is

completed, we run the kernel process termination instructions that were described on page 5 of this report.
In Task 3 too, all the check marks were accomplished.

```
void swap_in_process_function(){

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

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

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

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

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

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

In this part, our aim is to create and run a sanity test for our memory management module. We will implement a program that forks a lot of processes and allocates and uses memory to test the working of our program.

Observations:

1) Our process forks 20 children processes using fork().
2) Each child process runs 10 iterations.
3) In each iteration 4KB of memory is allocated.
4) The value at index i is i^2-4*i+1.

After this we also decrease the value of PHYSTOP defined in memlayout.h. The default value of PHYSTOP is 0xE0000000 (224MB). We changed its value to 0x0400000 (4MB). On running memtest, we still get the identical output.

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

int math_func(int num){
    return num*num*num;
}

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] = math_func(k);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    if(arr[k] == math_func(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();
```

Output

Since it is a long output, I am attaching the screenshot of first three child processes only

```
$ 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
   10      4096B      0B
```