CS344 Assignment 3

Group 36

Dhruv Shah (200101124)
Pranshu Shah (200101085)
Vedant Shah (200101115)

- ## Part A: Lazy Memory Allocation

Our aim here is to avoid giving memory the moment it is requested. We give memory only when it is accessed.
Originally in xv6 whenever the current process needed extra memory, it informed the OS about this requirement with the sbrk system call. growproc() is used by sbrk to fulfil this requirement.growproc() calls allocuvm() which is responsible for allocation of the extra memory.
We start with the patch file provided. The changes in this file tricks the process into thinking that it has been allocated the memory which it asked for by updating the value of proc->sz. But in reality we didn't allocate any physical memory because we commented the call to growproc() in sysproc.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 when any process requests access to the memory which it thinks it has, then it results in a page fault because in reality no such memory has been allocated.Our lazy allocator comes into picture here and now allocates one page from the free physical memory available to the process and also updates the page table about this new allocation.

Handling Page Fault and Allocating a New Page

The page fault generates a **T_PGFLT** trap to the kernel. We added the **T_PGFLT** case in trap.c and called the **handlePageFault()** function. In this the rcr2() returns the virtual address at which the page fault occurred. Rounded_addr points to the start of the page where this virtual address resides. Then **kalloc()** is called which finds and returns a free page from a linked list of free pages. Now we map the virtual address rounded_addr to the physical free page using **mappages()**.

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

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

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

Now, **mappages()** is a static function in vm.c. To use it in trap.c we removed the static keyword in front of it in vm.c and declared it's prototype in trap.c. **mappages()** takes the following as parameters :

1. Page table of current process
2. Virtual address of the start of data
3. Size of data
4. Physical memory at which physical page resides
5. Permissions corresponding to the page table entry

```
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;
}
```

mappages() uses pgdir to locate(and create,if required) the page table containing the virtual address a and creating a corresponding page table entry having physical address **P2V(mem)**, as obtained above with permissions set to writable(**PTE_W**) and user process accessible(**PTE_U**).

- ## Part B Questions:

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

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

A linked list of free pages is maintained in **kalloc.c** called **kmem**. **kinit1** is called through main() which adds 4MB of free pages to the list.

Q2) What data structures are used to answer this question?
Ans. Linked List is used. A new structure named struct run is made in kalloc.c and used as a linked list node.

Q3) Where do these reside?
Ans. They reside in kalloc.c, where kmem structure is instantiated. It contains a lock and linked list head.

Q4) Does xv6 memory mechanism limit the number of user processes?
Ans. Yes, the maximum number of user processes that can be active simultaneously are 64. They are set by default. We can change if we want.

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. At a time, lowest process running is 1, i.e. sh. Initproc is initially made runnable, but later it sleeps continuously. After every command execution, shell sleeps and again becomes runnable. So lowest number is 1

- ## Part B:

Task 1 : Kernel Processes

Kernel processes are the ones that reside their whole life in kernel mode. We made a **create_kernel_process()** function in proc.c. We don't need to initialise its trapframe because it resides in the kernel mode. The eip register of the process' context stores the address of the next instruction. We want the process to start executing at the entry point (which is a function pointer). Thus, we set the eip value of the context to entry point (Since entry point is the address of a function). allocproc assigns the process a spot in the ptable. setupkvm sets up the kernel part of the process' page table that maps virtual addresses above KERNBASE to physical addresses between 0 and PHYSTOP.

proc.c:

```
void create_kernel_process(const char *name, void (*entrypoint)()){

  struct proc *p = allocproc();

  if(p == 0)
    panic("create_kernel_process failed");

  //Setting up kernel page table using setupkvm
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");

  //This is a kernel process. Trap frame stores user space registers. We don't need to initialise tf.
  //Also, since this doesn't need to have a userspace, we don't need to assign a size to this process.

  //eip stores address of next instruction to be executed
  p->context->eip = (uint)entrypoint;

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

  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);

}
```

Task 2 : Swapping Out Mechanism

We need that the process that is not allocated any memory due to some reason must be suspended from execution. So firstly we need to store the processes that were not allotted the requested memory due to unavailability of free pages. To handle this we have created a circular queue struct called **req_queue** in proc.c. The name of the queue that holds processes with swap out requests is **queue1**. Supporting functions such as **rpush()** and **rpop()** have been created. Also a lock has been initialised in pinit and the queue needs to be accessed with that lock. Values of s and e have been initialised to zero in userinit. Also prototypes are added in defs.h so that it is accessible by the kernel in other files.

defs.h

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

proc.c:

```c
}

struct req_queue{
    struct spinlock lock;
    struct proc* queue[NPROC];
    int s;
    int e;
};
```

```c
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&queue1.lock, "queue1");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&queue2.lock, "queue2");
}
```

```c
void
userinit(void)
{
  acquire(&queue1.lock);
  queue1.s=0;
  queue1.e=0;
  release(&queue1.lock);
```

```c
struct proc* rpop(){

  acquire(&queue1.lock);
  if(queue1.s==queue1.e){
    release(&queue1.lock);
    return 0;
  }
  struct proc *p=queue1.queue[queue1.s];
  (queue1.s)++;
  (queue1.s)%=NPROC;
  release(&queue1.lock);

  return p;
}
```

```c
int rpush(struct proc *p){

  acquire(&queue1.lock);
  if((queue1.e+1)%NPROC==queue1.s){
    release(&queue1.lock);
    return 0;
  }
  queue1.queue[queue1.e]=p;
  queue1.e++;
  (queue1.e)%=NPROC;
  release(&queue1.lock);

  return 1;
}
```

When **kalloc()** returns zero, that means it was unable to allocate pages to a process. This is where we change the process state to sleeping. A special channel called sleeping_channel is where the process sleeps.sleeping_channel_count is used for corner cases when system boots. Then to keep a track of the processes that are to be swapped, we add the current process to the swap out request queue, **queue1**.

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

```c
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_ struct spinlock sleeping_channel_lock
  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;
}
```

Note: The create_kernel_process creates a kernel process (swapping out), which helps to allocate a page for this process if it doesn't already exist. The **swap_out_process_exists**

variable, which was initialised to 0, is set to 0. It is set to 1, when it is created. The purpose of this variable is to avoid the creation of multiple swaps.

Now, to make sure that whenever free pages are available, all sleeping processes(on the sleeping channel) are woken up we edit kfree in kalloc.c. We wake the processes by calling the **wakeup()** system call.

```c
void
kfree(char *v)
{

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

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTO
    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);

}
```

Swapping Out Process

```c
void swap_out_process_function(){

  acquire(&queue1.lock);
  while(queue1.s!=queue1.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 pa
      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");
        }
```


```c
        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");
        }
```


```c

        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(&queue1.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();
}
```

swap_out_process_function is the entry point for the swapping out process. This process runs a loop till the request queue (queue1) becomes nonempty. A set of instructions are executed for the termination of swap_out_process, when the queue1 becomes empty. First we pop the first process from the queue1 and then use pseudo LRU policy to determine the victim page in the page table. Iterating through the pgdir page table, we extract the physical address for each secondary page table. For each secondary page table, we iterate through the page table and look at the accessed bit (A) on each of the entries, which is the 6th MSB bit.

```
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;
  }
}
```

Getting a victim page in the secondary page table entry, we swap it out and store it onto the disk. Using the default convention given in the question itself, we name the file that stores this page.
To open, read, write, & close files in proc.c, we took help from the functions defined in the sysfile.c and constructed the new functions proc_open, proc_read, proc_write, proc_close.

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

}
```

```
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 the defined **O_CREATE** macro, we create the file and **O_RDWR** for the read/write access on the file. Then we simply write the page to the file using the proc_write function. Using memset, we clear the respective page table entry of the victim block. The page is added to the free page queue using the kfree to make it available for use.
To mark the page as *swapped*, we have set the 8th MSB bit in the secondary page table entry. Purpose of this is to know in future whether this page is already present in the swapped out page collection.

The loop is broken and all the process are suspended when the queue is empty. We clear the kernel process's kstack from outside the process. As soon as the scheduler finds a kernel process in the UNUSED state, the scheduler clears the process's kstack and its name.

The kernel process has been ended in 2 parts.

```c
    release(&queue1.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();
}
```

—> (1) from within process

```c
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      //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;
      }
```

—-> (2) from scheduler

Task 3 : Swapping In Mechanism

Like we created a swap out queue in task 2, here we need to create a swap in queue. It is very analogous to the **queue1** created in task 2. We named it **queue2** and supporting functions **rpop2()** and **rpush2()**. Also extern prototype was declared in defs.h. Initialisation of its s and e variables is done in userinit and lock in pinit.

We require a way to know at which virtual address the page fault occurred so we added a new field to struct proc in proc.h called addr(int).

Now, we handle page fault traps(**T_PGFLT**) raised in trap.c.

```c
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;

};
```

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

```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();
  }
}
```

We do it similarly as done in part A. We create a function named **handlePageFault()**. Similar to the first time we find the virtual address at which the page fault occurred by **rcr2()**. After putting the process to sleep with a new lock(**swap_in_lock**), we search for the page table entry corresponding to this address. Now, we need to check whether this page was swapped out or not. To help us here we set the page table entry's 7th order(2^7) bit while swapping out a page in task 2. To check if this bit is set or not we do a simple bitwise & test with 0x080.
Now it is written in the assignment to exit if it is not set so we do so by using **exit()**. Incase it is set , we continue with the swap_in_process(if it doesn't already exist - check using swap_in_process_exists).

Swapping In Process

The swapping in process initiates from the **swap_in_process_function(declared in proc.c)**. We have already mention how the file management functions in proc.c have been implemented. The function runs on a loop until **queue2** is not empty. Each time it takes a process from the queue , finds the filename(using pid and addr fields), then creates the filename in a string called "c" using int_to_string(refer task 2). Then it used proc-open to open this file in read only mode with file descriptor fd. We then allocate a free frame(mem) to this process using **kalloc()**. Then it reads from the file into this free frame using **proc_read**. Then we map the page corresponding to addr with the physical page that we got using **mappages().** Then we wakeup this process as the page fault is now fixed. After the loop is completed, we run the kernel process termination instructions.

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

```c
void swap_in_process_function(){

  acquire(&queue2.lock);
  while(queue2.s!=queue2.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(&queue2.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(&queue2.lock);
        panic("mappages");
      }
      wakeup(p);
  }

    release(&queue2.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

After we implemented the swapping out and swapping in mechanisms in earlier tasks we now need to test if it is indeed working properly. So our objective here is to implement a testing function to test the functionalities. The code for **memtest** is given below.

memtest.c > child_process(int)

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

#define PGSIZE 4096
#define ITERATIONS 10
#define CHILDREN 20


void child_process(int i){
    char *ptr[ITERATIONS];

    for(int j=0;j<ITERATIONS;j++){
        // set all values
        ptr[j]=(char*)malloc(PGSIZE);
        for(int k=0;k<(PGSIZE);k++){
            ptr[j][k]=(i+j*k)%128;
        }

        int matched=0;
        // check all values
        for(int k=0;k<(PGSIZE);k++){
            if(ptr[j][k]==(i+j*k)%128){
                matched++;
            }
        }
        printf(1, "Process: %d\tIteration: %d\tMatched: %dB\tDifferent: %dB\n",i+1, j+1, matched, 4096-matched)
    }

}

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

    for(int i=0;i<CHILDREN;i++){
        if(!fork()){

            child_process(i);

            printf(1, "\n");
            exit();
        }
    }

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

}
```

As asked in the question here the main process creates 20 child processes using **fork()** system call, each child process executes a loop with 10 iterations and at each iteration 4096B of memory is allocated using **malloc()**.
The value stored at index i is given by a complex mathematical equation. A counter named matched is maintained which stores the number of bytes that contain the right values. This is done by checking the value stored at every index with the value returned by the function for that index.

Running memtest

Firstly we need to include memtest in the makefile under UPROGS and EXTRA to make it accessible to the user. Type memtest in terminal to get the output.

```
Process: 16     Iteration: 6    Matched: 4096B  Different: 0B
Process: 16     Iteration: 7    Matched: 4096B  Different: 0B
Process: 16     Iteration: 8    Matched: 4096B  Different: 0B
Process: 16     Iteration: 9    Matched: 4096B  Different: 0B
Process: 16     Iteration: 10   Matched: 4096B  Different: 0B

Process: 17     Iteration: 1    Matched: 4096B  Different: 0B
Process: 17     Iteration: 2    Matched: 4096B  Different: 0B
Process: 17     Iteration: 3    Matched: 4096B  Different: 0B
Process: 17     Iteration: 4    Matched: 4096B  Different: 0B
Process: 17     Iteration: 5    Matched: 4096B  Different: 0B
Process: 17     Iteration: 6    Matched: 4096B  Different: 0B
Process: 17     Iteration: 7    Matched: 4096B  Different: 0B
Process: 17     Iteration: 8    Matched: 4096B  Different: 0B
Process: 17     Iteration: 9    Matched: 4096B  Different: 0B
Process: 17     Iteration: 10   Matched: 4096B  Different: 0B

Process: 18     Iteration: 1    Matched: 4096B  Different: 0B
Process: 18     Iteration: 2    Matched: 4096B  Different: 0B
Process: 18     Iteration: 3    Matched: 4096B  Different: 0B
Process: 18     Iteration: 4    Matched: 4096B  Different: 0B
Process: 18     Iteration: 5    Matched: 4096B  Different: 0B
Process: 18     Iteration: 6    Matched: 4096B  Different: 0B
Process: 18     Iteration: 7    Matched: 4096B  Different: 0B
Process: 18     Iteration: 8    Matched: 4096B  Different: 0B
Process: 18     Iteration: 9    Matched: 4096B  Different: 0B
Process: 18     Iteration: 10   Matched: 4096B  Different: 0B

Process: 19     Iteration: 1    Matched: 4096B  Different: 0B
Process: 19     Iteration: 2    Matched: 4096B  Different: 0B
Process: 19     Iteration: 3    Matched: 4096B  Different: 0B
Process: 19     Iteration: 4    Matched: 4096B  Different: 0B
Process: 19     Iteration: 5    Matched: 4096B  Different: 0B
Process: 19     Iteration: 6    Matched: 4096B  Different: 0B
Process: 19     Iteration: 7    Matched: 4096B  Different: 0B
Process: 19     Iteration: 8    Matched: 4096B  Different: 0B
Process: 19     Iteration: 9    Matched: 4096B  Different: 0B
Process: 19     Iteration: 10   Matched: 4096B  Different: 0B

Process: 20     Iteration: 1    Matched: 4096B  Different: 0B
Process: 20     Iteration: 2    Matched: 4096B  Different: 0B
Process: 20     Iteration: 3    Matched: 4096B  Different: 0B
Process: 20     Iteration: 4    Matched: 4096B  Different: 0B
Process: 20     Iteration: 5    Matched: 4096B  Different: 0B
Process: 20     Iteration: 6    Matched: 4096B  Different: 0B
Process: 20     Iteration: 7    Matched: 4096B  Different: 0B
Process: 20     Iteration: 8    Matched: 4096B  Different: 0B
Process: 20     Iteration: 9    Matched: 4096B  Different: 0B
Process: 20     Iteration: 10   Matched: 4096B  Different: 0B
```

We can see here that our code was correct because all the indices have the correct value. To test in a more random way we can run our tests on different values of PHYSTOP. The default value is 0xE000000, we changed it to 0x0500000. On running memtest the output is still same as the earlier one.

Thus we can be sure that the code we have written is correct.