CS344: OPERATING SYSTEMS LAB
Assignment 3
Group 13

Adarsh Kumar, 190101002
Upender Dahiya, 190101096
Tanishq Katare, 190101093
Keshav Chourasiya, 190101045

PART A

To implement the lazy memory allocation we need to change the following files:

- **sysproc.c :** (File already provided) The function ***sbrk()*** is used to allocate the physical memory. So, for lazy memory allocation implementation we just need to comment the ***growproc()*** function calling part which is responsible for this, and also keep track of the the size by incrementing the ***sz*** attribute of the process

```
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;
}
```

- **trap.c :** An additional condition was added in the switch-case section. A page fault occurs when ***tf->trapno*** equals ***T_PGFLT***, so therefore in this case ***kalloc( )*** and ***mappages( )*** were called to allocate the required space or to inform that no more memory is available (code reused from allocuvm( ) in vm.c)

```
case T_PGFLT:{
  char *mem;
  mem = kalloc();
  if (mem == 0) {
    cprintf("Out of Memory.\n");
  } else {
    memset(mem, 0, PGSIZE);
    // creating page table entry
    uint a = PGROUNDDOWN(rcr2());
    mappages(myproc()->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W | PTE_U);
  }
break;
}
```

- **vm.c :** Return type of function ***mappages( )*** was changed from ***static int*** to ***int*** so that it can be used in ***trap.c***

PART B
ANSWERING TO QUESTIONS

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

Ans: xv6 maintains a linked list of free pages in ***kalloc.c*** called ***kmem***. This list is empty first and xv6 initialises this list with 4MB free pages from ***main.c*** file where it calls ***kinit*** for kernel initialisation

- What data structures are used to answer this question?

Ans: xv6 uses ***singly linked list*** as the data structure for these free pages

- Where do these reside?

Ans: This linked list is declared inside ***kalloc.c*** inside a structure ***kmem***. Every node is of the type ***struct run*** which is also defined inside kalloc.c

- Does xv6 memory mechanism limit the number of user processes. If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

Ans: Under file ***param.h*** the macro ***NPROC*** set as ***64,*** limits the number of user processes that can be present simultaneously in the ptable. Lowest number of process xv6 can have is ***1***, there cannot be 0 processes after boot since all user interactions need to be done using user processes which are forked from ***initproc*** and ***sh***

TASK 1

In ***proc.c*** , the ***create_kernel_process()*** function was created and this kernel process will stay in kernel mode throughout its lifetime. We don't need to initialise its trapframe, user space, or the user sector of its page table.

The address of the next instruction is stored in the process context's ***eip*** register. We want the process to begin at the point of entrance (which is a function pointer given as an argument) therefore we set the context's eip value to the entry point, note entry point is the address of a function. ***allocproc*** assigns a position in ptable to the process. The kernel section of the process page table, which translates virtual addresses above ***KERNBASE*** to physical locations between 0 and ***PHYSTOP***, is set up using ***setupkvm***.

```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");
    p->context->eip = (uint)entrypoint;

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

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

TASK 2

Dividing this task into modules and therefore firstly, we required a process queue to keep track of the processes that were denied more memory due to lack of free pages. For this we generated the ***rq*** circular queue struct and the queue ***rqueue*** to contain the processes which have swap out requests. We've also created the ***rpush*()** and ***rpop*()** functions.

Code snippets from proc.c:

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

```
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+=1;
    rqueue.s%=NPROC;
    release(&rqueue.lock);
    return p;
}
```

```
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;
}
```

We will access this queue using a lock which we have initialised in ***pinit***. In ***userinit***, we've set the starting values of ***s*** and ***e*** to zero. We introduced prototypes in ***defs.h*** as well, because the queue and functions related to it are used in other files.

```
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);
```

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

If ***kalloc*** is unable to allocate pages to a process, it now returns 0. This informs ***allocuvm*** that the requested memory(mem=0) wasn't allocated. To begin, we must set the process state to 'sleeping'. The process sleeps on a unique sleeping channel called ***sleeping_channel*** which is protected by a lock ***sleeping_channel_lock***. When the system boots, ***sleeping_channel_count*** is used for the edge cases. Next, we must add the current process to the ***rqueue***.

Code snippet of allocuvm function:

```c

  //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);
  }

```

create_kernel_process here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. The ***swap_out_process_exists*** variable is set to 0 when the swap out process is completed. This is done to avoid the creation of multiple swap out operations.

Next, we develop a mechanism that wakes up all processes sleeping on ***sleeping_channel*** whenever free pages become available. Now we make the following changes to ***kfree*** function in ***kalloc.c*** Basically, all processes that were preempted due to lack of availability of pages were sent sleeping on the sleeping channel. The ***wakeup()*** system call is used to wake up any processes that are currently sleeping on the sleeping channel.

Code snippet from 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.
  // 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);

}
```

Now, we will explain the **swapping out process**. The entry point for the swapping out process is in the ***swap_out_process_function***.

The process runs a loop until the swap out requests queue (***rqueue1***) is non empty. When the queue is empty, a set of instructions are executed for the termination of ***swap_out_process***.

```c
220 void swap_out_process_function(){
221   acquire(&rqueue.lock);
222   while(rqueue.s!=rqueue.e){
223     struct proc *p=rpop();
224     pde_t* pd = p->pgdir;
225     for(int i=0;i<NPDENTRIES;i++){
226       //skip page table if accessed. chances are high, not every pa
227       if(pd[i]&PTE_A)
228         continue;
229       pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
230       for(int j=0;j<NPTENTRIES;j++){
231         //Skip if found
232         if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P))
233           continue;
234         pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));
235 
236         //for file name
237         int pid=p->pid;
238         int virt = ((1<<22)*i)+((1<<12)*j);
239 
240         //file name
241         char c[50];
242         int_to_string(pid,c);
243         int x=strlen(c);
244         c[x]='_';
245         int_to_string(virt,c+x+1);
246         safestrcpy(c+strlen(c),".swp",5);
247 
248         // file management
249         int fd=proc_open(c, O_CREATE | O_RDWR);
250         if(fd<0){
251           cprintf("error creating or opening file: %s\n", c);
252           panic("swap_out_process");
253         }
```

```c
248         // file management
249         int fd=proc_open(c, O_CREATE | O_RDWR);
250         if(fd<0){
251           cprintf("error creating or opening file: %s\n", c);
252           panic("swap_out_process");
253         }
254         if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
255           cprintf("error writing to file: %s\n", c);
256           panic("swap_out_process");
257         }
258         proc_close(fd);
259         kfree((char*)pte);       You, 2 hours ago • Uncommitted ch
260         memset(&pgtab[j],0,sizeof(pgtab[j]));
261         //mark this page as being swapped out.
262         pgtab[j]=((pgtab[j])^(0x080));
263         break;
264       }
265     }
266   }
267   release(&rqueue.lock);
268   struct proc *p;
269   if((p=myproc())==0)
270     panic("swap out process");
271 
272   swap_out_process_exists=0;
273   p->parent = 0;
274   p->name[0] = '*';
275   p->killed = 0;
276   p->state = UNUSED;
277   sched();
278 }
```

The loop begins by removing the first process from ***rqueue*** and determining a victim page in the page table using the **LRU policy**. We derive the physical address for each secondary page table by iterating through each entry in the process page table, ***pgdir***. We go through the page table for each secondary page table, looking at the accessed bit (A) on each entry. When the scheduler switches the process's context, all accessed bits are cleared. Since we are doing this, the accessed bit seen by ***swap_out_process_function*** will indicate whether the entry was accessed in the last iteration 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.

Now, back to the ***swap_out_process_function***. As soon as the function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number as the victim page. After that, this page is then swapped out and saved on the drive. To name the file that saves this page, we use the process **pid**(line 237) and **virtual address** of the page(line 238) to be erased. We need to write the contents of the victim page to the file with the name ***<pid>_<virt>.swp***.

But we have a problem here. The filename is saved in a string named **c**. Calls to the file system cannot be made from ***proc.c***. So we copied the **open**, **write**, **read**, **close** functions from ***sysfile.c*** to ***proc.c***,and updated and renamed them to ***proc_open***, ***proc_read***, ***proc_write***, ***proc_close***, and so on so we could use them in ***proc.c***. Here are some examples:

```
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;
}

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);
}
```

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**. The page is then added to the free page queue using **kfree**, making it available for usage (note that when **kfree** adds a page to the free queue, it also wakes up all processes sleeping on the sleeping channel). We then use **memset** to clear the page table entry as well. For Task 3, we need to know if the page that caused a fault was swapped out or not. For marking the page as swapped out, we set the 8th bit from the right (2^7) in the secondary page table entry using **xor**.

TASK 3

To begin, we must first make a swap in the request queue. In ***proc.c***, we created a swap in request queue called ***rqueue2*** using the same ***struct rq***. In ***defs.h***, we additionally specify an extern prototype for ***rqueue2***. We developed the matching functions for ***rqueue2*** for push and pop operatinons named ***rpop2()*** and ***rpush2()*** in proc.c. In **pinit**, we also initialised its lock and in **userinit**, we additionally set the ***s*** and ***e*** attributes.

Then, in **proc.h**, we added an extra entry named ***addr*** to the **struct proc**. This item will inform the swapping in function about the virtual address at which page fault was found:

```c
  char name[16];               // Process name (debugging)
  int addr;                    // Virtual address of pagefault
};
```

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

```c
  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();
  }
}
```

In **handlePageFault()**, we use ***rcr2()*** to identify the virtual address where the page fault occurred, exactly as we did in Part A.We then use a new lock called ***swap_in_lock*** to put the current process to sleep.Then we get the page table entry analogous to this address. Now we have to find if this page was swapped out or not.

When we swapped out a page in Task 2, we set its page table entry's bit of 7th order (2^7).
To detect if the page was swapped out or not, we apply bitwise & with 0x080 to check its 7th order bit. If it's set, ***swap_in_process*** is initiated. Otherwise, we may safely suspend the process using ***exit()***.

Now we'll go over the swapping in procedure. As you can see in handlePageFault, the ***swap_in_process_function*** is the entry point for the swapping out process.
Until ***rqueue2*** is not empty, the file management procedure loops. In the loop, it gets the file name by popping a process from the queue and extracting its ***pid*** and ***addr*** values. It then uses ***int_to_string*** to construct the filename in a string named "c”. Then, given file descriptor ***fd***, it used ***proc_open*** to open this file in read-only mode. Using ***kalloc***, we then allocate a free frame to this process. Using ***proc_read***, we read from the file with the fd, file descriptor into this free frame. By deleting the ***static*** keyword from ***vm.c*** and declaring a prototype in ***proc.c***, we make ***mappages*** accessible to ***proc.c.*** The page corresponding to ***addr*** is then mapped with the physical page obtained using ***kalloc*** and read into using ***mappages***.Then we wake up, which is a procedure in which we allocate a new page to correct the page fault. We perform the kernel process termination instructions once the loop is complete.

```c
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 task, a test user-program is created named ***memtest***.
- This program has main process that forks and creates ***20 child processes***
- Each of this child process is looping through ***10 iterations*** and at each iteration, ***4096B*** of memory is being allocated using the ***malloc()*** function
- As asked a random mathematical expression is used at every ith iteration to store some value at ith index of the array
- After filling the values in the memory, to validate used the same mathematical function to compare the stored values
- Since this user program is a new one hence under ***UPROGS*** and ***EXTRA*** in the ***Makefile*** we need to update the appropriate function name

We can notice from the output that the sanity test gives the correct output since the unmatched column that denotes the total unmatched bytes are all 0B and hence shows that the indices store the correct value

Now upon testing on different values of ***PHYSTOP*** defines under the file ***memlayout.h*** The correct implementation should show identical results.

Therefore on changing the default value of PHYSTOP i.e. ***0xe000000(224MB)*** to something smaller such as ***0x0400000(4MB)*** the output doesn't change indicating that the implementation is correct

```
memtest        2 18 16972
console        3 19 0
$ memtest
          Iteration Matched Unmatched
         ____________________________
Child 1 |     1      4096B      0B
Child 1 |     2      4096B      0B
Child 1 |     3      4096B      0B
Child 1 |     4      4096B      0B
Child 1 |     5      4096B      0B
Child 1 |     6      4096B      0B
Child 1 |     7      4096B      0B
Child 1 |     8      4096B      0B
Child 1 |     9      4096B      0B
Child 1 |    10      4096B      0B
_____________________________________
Child 2 |     1      4096B      0B
Child 2 |     2      4096B      0B
Child 2 |     3      4096B      0B
Child 2 |     4      4096B      0B
Child 2 |     5      4096B      0B
Child 2 |     6      4096B      0B
Child 2 |     7      4096B      0B
Child 2 |     8      4096B      0B
Child 2 |     9      4096B      0B
Child 2 |    10      4096B      0B
_____________________________________
Child 3 |     1      4096B      0B
Child 3 |     2      4096B      0B
Child 3 |     3      4096B      0B
Child 3 |     4      4096B      0B
Child 3 |     5      4096B      0B
Child 3 |     6      4096B      0B
Child 3 |     7      4096B      0B
Child 3 |     8      4096B      0B
Child 3 |     9      4096B      0B
Child 3 |    10      4096B      0B
_____________________________________
Child 4 |     1      4096B      0B
Child 4 |     2      4096B      0B
Child 4 |     3      4096B      0B
Child 4 |     4      4096B      0B
Child 4 |     5      4096B      0B
```