CS 344 Operating Systems Laboratory Assignment 3

Group Number 3

TEAM MEMBERS:

NAME	ROLL NUMBER
RITWIK GANGULY	180101067
PULKIT CHANGOIWALA	180101093
SAMAY VARSHNEY	180101097
UDANDARAO SAI SANDEEP	180123063

PART A: *Lazy Memory Allocation*

Task 1: *Eliminate allocation from sbrk()*

In this task we have made the following modifications:

1) **sysproc.c**

```
// modified sys_sbrk() in sysproc.c
int addr;
int n;
if (argint(0, &n) < 0) {
  return -1;
}
addr = proc->sz;
proc->sz += n;
return addr;
```

Here, it increases **proc->sz** by **n** to trick the process into believing that it possesses the memory requested.

This is the faulty output produced:

```
$ echo hi
pid 4 sh: trap 14 err 6 on cpu 0 eip 0x112c 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 modifications:

1) **proc.h**
   Added **uint oldsz;** in **struct proc**.
   // Here, we have declared the variable oldsz as an unsigned integer.
2) **proc.c**
   p->oldsz = 0;
   // Here, we initialized the above declared variable with zero value inside allocproc() function.
3) **trap.c**
   extern int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
   *//Here, we declared mappages in trap.c after removing **static** keyword for mappages function in vm.c*

```
//PAGEBREAK: 13
default:
  if(myproc() == 0 || (tf->cs&3) == 0){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
            tf->trapno, cpuid(), tf->eip, rcr2());
    panic("trap");
  }


  if(rcr2() > myproc()->sz){
      cprintf("Unhandled Page Fault \n");
  }

  else{

      if(tf->trapno == T_PGFLT){

          char *mem;
          uint a;
          if(myproc()->sz < myproc()->oldsz){
              return;
          }

          a = PGROUNDDOWN(rcr2());
          mem = kalloc();
          if(mem == 0){
              cprintf("allocuvm out of memory \n");
              myproc()->killed = 1;
              return;
          }

          memset(mem, 0, PGSIZE);
          mappages(myproc()->pgdir, (char*) a, PGSIZE, V2P(mem), PTE_W|PTE_U);
          break;

      }
  }
```

// Here, we modified the default page trap to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address and then returning back to the user space to let the process continue executing.
We first check if the trapped fault is indeed a page fault using the condition (tf->trapno == T_PGFLT)
To avoid the cprintf statement, we return if kalloc() returns 0.
After recompiling, we find echo hi properly working.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13616
echo           2 4 12628
forktest       2 5 8056
grep           2 6 15492
init           2 7 13208
kill           2 8 12680
ln             2 9 12576
ls             2 10 14764
mkdir          2 11 12760
rm             2 12 12736
sh             2 13 23224
stressfs       2 14 13408
usertests      2 15 56340
wc             2 16 14156
zombie         2 17 12400
console        3 18 0
$ echo hi
hi
$ cat README
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.828/, which
provides pointers to on-line resources for v6.
```

Here, we have run a couple of other shell commands like ls and cat in addition to echo to check that our output is properly showing.

PART B: *xv6 Memory Management*

NOTE:

We used the main reference as https://www.usebackpack.com/resources/24427/download for implementing the Part B.

Reading References:

https://www.cs.columbia.edu/~junfeng/11sp-w4118/lectures/mem.pdf

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

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

Ans:

xv6 maintains a record of free physical memory, to be used by the processes that are to run. It uses the physical memory from the end of the **loaded kernel's data segment** (denoted by **end**) till the end of available Physical Memory (denoted by **PHYSTOP**). xv6 allocates physical memory using **pages**. It maintains a **linked list** (guarded by a **spinlock**) of all physical pages currently available and deletes newly allocated pages from the list, and adds freed pages back to the linked list.

The **allocator** (implemented in **kalloc.c**) maintains a free list of memory addresses of **main memory pages** that are available for **data storage**.

Q2) What data structures are used to answer this question?

Ans:

Singly Linked List with no data, and a **single pointer** to the next node.

Each **struct run** represents a free page's list element. It records the page's run structure in the free page itself, since it is currently empty. The linked list **run** is protected by a **spin lock**. The lock needs to be acquired before changing the list. This **prevents simultaneous access** of the link list by multiple programs. The list and the lock are enclosed in a **structure** to emphasize that the lock protects the linked list.

Refer to below structures :-

```
struct run {
 struct run *next;
};

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

Q3) Where do these reside?

Ans:

The allocator implementation and the corresponding data structures can be found in file **kalloc.c.** Each **struct run** represents a free page's list element. It records the page's run structure in the free page itself, since it is currently empty.

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

Ans:

Since xv6 doesn't support paging to disk by default, this means that if it allocates **memory** for the **user process** (using the **sbrk()** system call, used by **malloc()** in userspace), then it keeps it in the **physical memory** till the process is terminated. Since the **physical memory** is bounded (from **end** till **PHYSTOP**), only a limited number of processes can exist in the physical memory at a given time. We can't allocate memory to any further process as the newly created process would require free physical memory (not available due to currently running processes).

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:

With respect to the xv6 operating system, the **minimum number** of processes running at the same time will be **1** (the init() process). **Init** is a daemon **process** that continues running until the system is shut down.

And, we have set the variable NPROC to 64 in the file param.h. So, the maximum number of processes is 64 (including both kernel and user process) at a given instance. This limit has been set to avoid memory overflow due to a higher number of running processes.

TASK 1: *Kernel Processes*

To implement the kernel processes, we made modifications to the following files:

1) **proc.c:**

   We added the function **create_kernel_process()** in the file in **proc.c**. Implementation of create_kernel_process() is somewhat a mixture of functions like fork(), allocproc(), and userinit(). It roughly is similar to fork() but there are certain modifications. **fork()** retrieves the **address space, registers**, etc. from its parent process but create_kernel_process() refrains from such a method. In the locations where fork() function copies data from the parent process, create_kernel_process() initializes the data with random values the same way allocproc() and userinit() do.

   The ***np->tf = *proc->tf** (setting up of trap frame) of the fork() function is replaced by setting up the entire trap frame the same way userinit() does, in create_kernel_process() function.

   At the end of create_kernel_process(), it sets **np->context->eip = (uint) entrypoint**. This means that the process will start running at the function entrypoint when it starts. Please refer to the below screenshot for the function **create_kernel_process()**.

```c
void
create_kernel_process(const char *name, void (*entrypoint)()){
  struct proc *np;
  struct qnode *qn;

  if ((np = allocproc()) == 0) panic("Failing allocating kernel process");

  qn = freenode;
  freenode = freenode->next;

  if(freenode != 0) {
    freenode->prev = 0;
  }

  if((np->pgdir = setupkvm()) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    panic("Failed setup pgdir for kernel process");
  }

  np->sz = PGSIZE;
  np->parent = initproc;
  memset(np->tf, 0, sizeof(*np->tf));
  np->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  np->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  np->tf->es = np->tf->ds;
  np->tf->ss = np->tf->ds;
  np->tf->eflags = FL_IF;
  np->tf->esp = PGSIZE;

  // beginning of initcode.S
  np->tf->eip = 0;

  // Set eax = 0 so that fork return 0 in the child
  np->tf->eax = 0;
  np->cwd = namei("/");
  safestrcpy(np->name, name, sizeof(name));
  qn->p = np;

  // lock to force the compiler to emit the np-state write last.
  acquire(&ptable.lock);
  np->context->eip = (uint)entrypoint;
  np->state = RUNNABLE;
  release(&ptable.lock);
}
```

2) main.c:

To test the above function, the create_kernel_process() function is called twice, once for the swapin() process and once for the swapout() process in the main() function in main.c.

Note: Both the functions have been commented out in the submission.

TASK 2: ***Swapping Out Mechanism***

We have a function swap_out():

It **sleeps** on the channel if there is nothing to swap out.
When it wakes up (when a page needs to be swapped out), it creates a file on the disk using **filealloc()** (defined in file.c), finds the **least recently used** page, reads the respective page from the page table as a stream of bytes, saves the bytes to the file that was created using **filewrite()** (defined in file.c), then frees the page from the page table.

We have used the below functions to mimic the **swap out** mechanism

pte_t* select_a_victim(pde_t *pgdir)

- The above function selects the victim frame that is to be replaced during swapping out.
- refer to vm.c for the function code

int swap_out(pte_t *mapped_victim_pte, unsigned int offset)

- When a victim page is to be swapped out, it calculates the **swapfile offset**
- Then we increase the count of variable **pages_swapped_out**
- **P2V()** function converts virtual address to physical address
- Using **P2V()** function we convert the victime frame address to physical frame.
- Then we call **filewrite()** function to write swapped out functions in the respective file.

```
int swap_out(pte_t *mapped_victim_pte, unsigned int offset)
{
   struct swap_info_struct *p = &swap_info[0];
   int file_offset = offset + 1, retval = -1;
   uint old_offset;
   char *kernel_addr = P2V(PTE_ADDR(*mapped_victim_pte));

   if(p->swap_file == NULL)
         return -1;
   old_offset = p->swap_file->off;

   // Quick and dirty hack for now. Need a lock-protected state variable later
   myproc()->pages_swapped_out++;

   // Write contents to swapfile
   p->swap_file->off = (unsigned int)(file_offset * PGSIZE);
   retval = filewrite(p->swap_file,kernel_addr,PGSIZE);
   p->swap_file->off = old_offset;
   return retval;
}
```

TASK 3: *Swapping In Mechanism*

We have a function swap_in() in proc.c.
In swap_in(), it **sleeps** on the channel if there is nothing to swap in.
When it wakes (when a page needs to be swapped in), it finds the **needed file on the disk**, finds a **free page** in the page table, writes the **stream of bytes** from the file to the page, then deletes the file from disk.

We have used the below functions to mimic the swap in mechanism.

int swap_in(void *page_addr, unsigned int offset)

- When a required page is to be **swapped in**, it calculates the **swapfile offset**
- Then we **decrease** the count of variable **pages_swapped_out**
- **P2V()** function converts virtual address to physical address
- Using **P2V()** function we convert the victime frame address to physical frame.
- Then we call **fileread()** function to read pages from the respective file.

```
// Swap a page into main memory from the specified slot
int swap_in(void *page_addr, unsigned int offset)
{
   struct swap_info_struct *p = &swap_info[0];
   int file_offset = offset + 1, retval = -1;
   uint old_offset;

   // SWAPFILE pointer not set yet with ksetswapfileptr() system call
   if (p->swap_file == NULL)
         return -1;

   old_offset = p->swap_file->off;

   // Quick and dirty hack for now. Need a lock-protected state variable later
   myproc()->pages_swapped_out--;

   // Read contents from swapfile
   p->swap_file->off = (unsigned int)(file_offset * PGSIZE);
   retval = fileread(p->swap_file,page_addr,PGSIZE);
   p->swap_file->off = old_offset;
   return retval;
}
```

void map_address(pde_t *pgdir, uint addr)

The above function maps a **physical page** to the **virtual address** addr. If the page table entry points to a swapped block restore the content of the page from the swapped block and free the swapped block.

1) **kalloc** a physical page
2) map **physical page to virtual page** (addr)
3) Set the **access bit** of the page (last **12** bits are same in physical and virtual page), so they share the access bit

void handle_pgfault(void)

The above function **handles page faults**, when the page is not found in the page table.

```
// page fault handler
void handle_pgfault()
{
   unsigned addr;
   struct proc *curproc = myprocxv6();
   asm volatile ("movl %%cr2, %0 \n\t" : "=r" (addr));
   addr &= ~0xfff;
   map_address(curproc->pgdir, addr);
}
```

TASK 4: *Sanity Test*

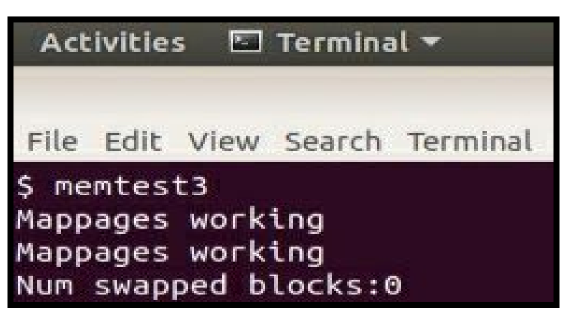
We have changed value of **PHYSTOP** in **memlayout.h**
Initial statement in **memlayout.h**: **#define PHYSTOP 0xE000000**
After changing the statement: **#define PHYSTOP 0x4000000**

This will **prevent** the kernel from being able to hold **all processes memory** in the RAM. (and enables us to observe the **paging mechanism**)

We have created different test files: One to check processes with memory allocation and counting their swap count, and one without memory allocation to check whether the swap count is zero or not.

Refer to the below screenshots for the terminal outputs.

1) ***Test with no memory allocation task:***



2) **Test output with 4kb memory allocation:**

Output “mem ok 17” signifies memory allocation was successful.

We used bstat sys call: It returns the global count of swapped pages.
Here 17 is the number of swapped pages.

```
kalloc success
CurrCount: 605
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Mappages working
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
CurrCount: 606
CurrCount: 607


While Loop Over
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
```

```
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
mem ok 17
$
```