CS_344 Assignment 3

Group C21: Akshat Mittal, 200101011

Satvik Tiwari, 200101091

Pranjal Baranwal, 200101083

Part A: Lazy Memory Allocation

In this part of assignment, we have to implement lazy memory allocation technique. A process uses the sbrk() system call to signal that it requires more memory than is currently allocated. The growproc() method, which calls the allocuvm() function, was used in the system call. The latter is in charge of allocating the needed additional memory by creating pages and correctly mapping virtual addresses to their corresponding physical locations in the page tables.

We began implementing this using the patch that had already been provided to us. In sysproc.c, the implementation of sys_sbrk() has been changed. The growproc() function call is commented.

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  myproc()->sz += n;

  // delaying the memory allcoation by commenting growproc -> lazy allcoation
  //  if(growproc(n) < 0)
  //    return -1;

  return addr;
}
```

Because we are only increasing the value of the sz field (the size of the process) using the proc->sz statement in sysproc.c and not actually allocating any memory to the process, the process is only being misled into believing that it has been allotted the memory. As a result, if an attempt is made to access the memory that was previously requested, there will not be a page table entry corresponding to the virtual address (miss in the page table) and will result in a page fault.

```
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 start 58
init: starting sh
$ ls
pid 3 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
```

Therefore, in lazy memory allocation, if these page faults occur, then and only then, one page (from the list of free physical memory pages available) is allocated to the process and suitable page table entries are added or updated corresponding to this allocation. It is also referred to as demand paging.

Handling Page Fault in trap.c

We have seen that when a process uses the sbrk() system function, it is being deceived into thinking that memory is being allocated to it. However, we are simply increasing the process size. Therefore, when this process tries to access the aforementioned required memory (which it believes to have already been bought inside), it runs into a PAGE FAULT and sends a trap called T_PGFLT to the kernel.

```
case T_PGFLT:
  // rcr2() is giving the virtual address
  lazy_mem_alloc(myproc()->pgdir, rcr2());
  break;
```

The function lazy_mem_alloc(), defined in vm.c, which handles the actual memory allocation, is called in order to manage the page fault. The virtual address that resulted in the page fault is returned by this function, which accepts two inputs. myproc()->pgdir returns a pointer to the page directory of the process. The page directory of the process is nothing but the outer level of the 2-level page table used.

lazy_mem_alloc() in vm.c- Implementation details

```
// Allocate a new memory page to the process
void
lazy_mem_alloc(pde_t *pgdir, uint va)
{
  char *mem;
  mem = kalloc();

  if(mem == 0){
    cprintf("allocuvm out of memory (3)\n");
    return;
  }

  memset(mem, 0, PGSIZE);
  uint a = PGROUNDDOWN(va);
  if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
    cprintf("allocuvm out of memory (4)\n");
    kfree(mem);
  }
}
```

This function is responsible for the actual allocation of pages and corresponding update in the page table.
First, it rounds the virtual address (which caused the page fault) to the beginning of the page boundary using PGROUNDDOWN(va).
Once the list of accessible physical addresses has been obtained, kalloc() is called. If it returns 0 in any other case, it means that no memory can be allocated.
Then the physical address V2P(mem), virtual address and page size are passed as arguments to the mappages() function. The page's permissions are also specified to be readable using the PTE_W option and accessible by user processes using the PTE_U.
Any allocated memory is released using the kfree() function if mappages() encounters an error.

Correctness in output after doing lazy allocation

We can see that the commands ls and echo hi are functioning as expected after making all of these changes. The output can be seen below.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16260
echo           2 4 15116
forktest       2 5 9424
grep           2 6 18480
init           2 7 15700
kill           2 8 15148
ln             2 9 15000
ls             2 10 17628
mkdir          2 11 15244
rm             2 12 15220
sh             2 13 27864
stressfs       2 14 16136
usertests      2 15 67240
wc             2 16 17000
zombie         2 17 14812
console        3 18 0
$ echo hi
hi
```

Part B (Xv6 Memory)

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

The kernel keeps track of a linked list of free pages called freelist, which serves as the linked list's head, under struct kmem in the file kalloc.c. The list is empty at first. Every time a physical page is initialized or released; it is added to the linked list.

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

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

The linked list called freelist is the data structure employed for this. If we look at the definition, we can see that each node is a structure named struct run that is also declared in the file kalloc.c. The kernel keeps track of the free pages by using this struct run data structure. A linked list of free pages is created by the structure, which stores a pointer to the next empty page. The page itself contains a pointer to the next available page.

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

Q3. Where do these reside?

Within the kmem structure, in the file kalloc.c, this linked list is declared. Each node is a member of the struct run, whose declaration is also found in the file kalloc.c as seen above. The actual structures are kept in the kernel memory.

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

The size of the process table is limited (NPROC which is 64 by default and is defined in param.h). Due to this the number of user processes is limited as well.

Q5. If so, what is the lowest number of processes xv6 can have at the same time (assuming the kernel requires no memory whatsoever)?

There is only one process running when the xv6 operating system starts up, and its name is initproc. This forks the shell process, which in turn forks other incoming user processes. The 240MB physical memory limit is known (PHYSTOP). Taking any process into consideration, it can have a virtual address space of 2GB (KERNBASE), and since 2GB > 240MB, one process can consume all of the physical memory. As a result, there are never more than 1 processes running simultaneously in xv6. Furthermore, since all user interactions must be performed using user processes, there cannot be zero processes after boot.

Task 1: Kernel processes

Kernel processes are created and added to the processes queue by the method create_kernel_process(), which is defined in proc.c. It selects the newly generated process and places it in an empty spot in the process table. Following this, the kernel stack is allotted for the trapframe. Next, the context is set up, the exit() function is placed on the stack after the trapframe and the entrypoint function's eip is set to the trapframe's value. The page table is then formed by executing setupkvm(). The parent is set to initproc and the state is changed to RUNNABLE.

```c
// This function create a kernel process and add it to the processes queue.
void
create_kernel_process(const char *name, void (*entrypoint)()){
  char *sp;
  struct proc *p;

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  p->kstack = kalloc();
  if (p->kstack == 0)
  {
    p->state = UNUSED;
    return;
  }
  sp = p->kstack + KSTACKSIZE;
```

```c
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*) sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)exit; // end the kernel process upon return from entrypoint()

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  (p->context)->eip = (uint)entrypoint;

  if((p->pgdir = setupkvm()) == 0)
    panic("kernel process: out of memory?");

  p->sz = PGSIZE;
  p->parent = initproc;
  p->cwd = idup(initproc->cwd);

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

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

The exit() function ends the process once it has returned from the entrypoint() and prevents it from entering user mode again. forkret() calls the create_kernel_process() function, which in turn generates the swap-out and swap-in kernel processes.

```c
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the co
    // of a regular process (e.g., they call sleep), and t
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
    // Create two kernel processes for swap-in and swap-ou
    create_kernel_process("swapinprocess",swapin_proc);
    create_kernel_process("swapoutprocess",swapout_proc);
  }
```

Task 2: Swapping out mechanism

Two additional fields have been added to the proc data structure in proc.h:

1. satisfied: Shows if the swap out request has been fulfilled for the specified process.
2. trapva: Saves the virtual address of the page that is producing a page fault for a particular process.

The kernel process named swapout_proc() is responsible for swapping out pages from virtual memory of a given process whenever needed.
It supports a request queue which is implement as a circular queue ADT with the following features:

1. front, rear: pointers indicating start and end of the circular queue.
2. size: represents the size of the circular queue.
3. reqchan: The channel on which all the requesting processes for swapping out of a page wait on.
4. qchan: The channel on which the swap out function waits when there are no processes to serve in the request queue.
5. lock: A spinlock to protect the shared access of the queue among different processes.
6. queue: Array storing the pointer to PCB of the processes queued for swapping out requests.
7. enqueue( ): Method to add a process to the queue.
8. dequeue( ): Method to remove a process from the queue.

```c
// Circular Queue ADT
struct swapqueue{
  int front, rear;
  int size;
  char* reqchan;
  char* qchan;
  struct spinlock lock;
  struct proc* queue[NPROC+1];
};
```

```c
// Enqueue function for the queue
void
enqueue(struct swapqueue* sq, struct proc* np){
  if(sq->size == NPROC)
    return;
  sq->rear = (sq->rear + 1) % NPROC;
  sq->queue[sq->rear] = np;
  sq->size++;
}
```

```c
struct proc*
dequeue(struct swapqueue* sq){
  if (sq->size == 0)
    return 0;
  struct proc* res = sq->queue[sq->front];
  sq->front = (sq->front + 1) % NPROC;
  sq->size--;
  if(sq->size == 0){
    sq->front = 0;
    sq->rear = NPROC - 1;
  }
  return res;
```

req_swapout() function does the following:

1. Add the requesting process to the swap_out_queue.
2. Wake the swapout_proc() to swap out a page for accommodation of space of a new page for the requesting process.
3. Make the requesting process sleep until the request is served.
4. Set the satisfied field to 0.

```c
// Submits a request for a free page to the swapout
void req_swapout(){
  struct proc* p = myproc();
  // cprintf("Submitted request to swap-out %d\n",p-
  char my_pid[3];
  my_pid[2] = 0;
  my_pid[1] = '0' + p->pid%10;
  my_pid[0] = (p->pid/10 ? '0' + p->pid/10 : ' ');

  cprintf("| Submit Request to SwapOut |  %s | -  |

  acquire(&ptable.lock);

  acquire(&swap_out_queue.lock);
  p->satisfied = 0;
  enqueue(&swap_out_queue, p); // Enqueues the proce
  wakeup1(swap_out_queue.qchan); // Wakes up the Swa
  release(&swap_out_queue.lock);
  while(p->satisfied == 0) // Sleep process till not
    sleep(swap_out_queue.reqchan, &ptable.lock);

  release(&ptable.lock);
  return;
}
```

```c
void swapout_proc(){
  sleep(swap_out_queue.qchan, &ptable.lock);
  while(1){
    // cprintf("\n\nEntering swapout\n");
    cprintf("|      Swapout Resumes      |  -  |
    acquire(&swap_out_queue.lock);

    while(swap_out_queue.size){
      // Edge case handling
      while (flimit >= NOFILE){
        cprintf("flimit\n");
        wakeup1(swap_out_queue.reqchan);
        release(&swap_out_queue.lock);
        release(&ptable.lock);
        yield();
        acquire(&swap_out_queue.lock);
        acquire(&ptable.lock);
      }

      // Dequeue process from queue
      struct proc *p = dequeue(&swap_out_queue);

      // Edge case handling
      if(!select_victim_evict(p->pid)){
        wakeup1(swap_out_queue.reqchan);
        release(&swap_out_queue.lock);
        release(&ptable.lock);
        yield();
        acquire(&swap_out_queue.lock);
        acquire(&ptable.lock);
      }

      // When frame found set satified to true
      p->satisfied = 1;
    }

    wakeup1(swap_out_queue.reqchan);
    release(&swap_out_queue.lock);
    sleep(swap_out_queue.qchan, &ptable.lock);
  }
```

A kernel process called swapout_proc() stays running as long as there are requests to fulfil and shuts down when there are none. The following is what it does to fulfil a request:

1. A process is dequeued from the swap_out_queue.
2. A victim frame is selected in accordance with the replacement rules and allocated to requesting process.
3. satisfied field is set to 1.
4. Wake up the corresponding process.

LRU replacement policy is used for selecting the victim frame for eviction. To decide the preference order, the accessed bit and the dirty bit are concatenated to form an integer and the following preference order is maintained:

$$00 < 01 < 10 < 11$$

When a victim frame has been selected, the process is put into the SLEEPING state until the victim page has been written to disc, at which point the present bit for the relevant PTE is unset. In order to successfully swap out the target frame, the seventh bit (which is unset by default) must be set. If the victim frame is successfully evicted, the method returns 1, else it returns 0.

```c
// Chooses a victim frame using LRU and evicts it
int
select_victim_evict(int pid){
  struct proc* p;
  struct victim victims[4] = {{0,0,0},{0,0,0},{0,0,0},{0,0,0}};
  pde_t *pte;

  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if (p->state == UNUSED || p->state == EMBRYO
        || p->state == RUNNING || p->pid < 5 || p->pid == pid)
      continue;

    for(uint i=PGSIZE; i<p->sz; i+=PGSIZE){
      pte = (pte_t*)getpte(p->pgdir, (void *) i);
      if(!((*pte) & PTE_U) || !((*pte) & PTE_P))
        continue;

      // 96 = 1100000 in binary
      int idx = (((*pte)&(uint)96)>>5);
      if(idx>0 && idx<3)
        idx = 3-idx;

      victims[idx].pte = pte;
      victims[idx].va = i;
      victims[idx].pr = p;
    }
  }

  for(int i=0; i<4; i++){
    if(victims[i].pte == 0)
      continue;
```

```c
for(int i=0; i<4; i++){
  if(victims[i].pte == 0)
    continue;

  pte = victims[i].pte;
  int origstate = victims[i].pr->state;
  char* origchan = victims[i].pr->chan;
  victims[i].pr->state = SLEEPING;
  victims[i].pr->chan = 0;
  uint reqpte = *pte;
  *pte = ((*pte) & (~PTE_P)) | ((uint)1<<7);

  if(victims[i].pr->state != ZOMBIE){
    release(&swap_out_queue.lock);
    release(&ptable.lock);

    write_page(victims[i].pr->pid,(victims[i].va)>>12,
        (void *)P2V(PTE_ADDR(reqpte)));

    acquire(&swap_out_queue.lock);
    acquire(&ptable.lock);
  }

  kfree((char *)P2V(PTE_ADDR(reqpte)));
  lcr3(V2P(victims[i].pr->pgdir));
  victims[i].pr->state = origstate;
  victims[i].pr->chan = origchan;
  return 1;
}
return 0;
```

```c
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  // Until free frame not foun
  while (!r) {
    if (kmem.use_lock)
      release(&kmem.lock);
    req_swapout();
    if (kmem.use_lock)
      acquire(&kmem.lock);
    r = kmem.freelist;
  }
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

The kalloc() function is used to allocate the 4096 bytes of physical memory to meet swapping on demand. req_swapout() is called until a free frame is obtained.

The frame's contents are written to disc using write_page(). PID_VA[20:] is used as the filename. PID stands for process ID, while VA[20:] stands for first 20 bits of virtual address, which correspond to the evicted page. This is done using get_name() function.

To open/create files, the internal implementation of write_page() uses open_file(). To write the contents, use file_write(). These functions are present in sysfile.c and extracted from there.

Task 3: Swapping in mechanism

When swapping in, the first function used is swapin_proc(). This function periodically searches the swap_in_queue for a request to fulfil. Using kalloc(), it first obtains a free frame in main memory. Once it has a free frame, it reads the page into that frame from the disc. After that, it modifies the flags and physical page numbers (PPN) in the associated page table entry (PTE) using swapInMap() defined in vm.c. The corresponding process if then woken up.

When all the processes in the swap_in_queue are fulfilled, this process enters into SLEEPING state. Prior to this, all appropriate locks are opened.

When a page is swapped out into the buffer memory, the read_page() method aids in reading the new page. After calculating the filename using get_name(), it invokes file_read() to read the file's content.

```
// Entry point of the swapin process
void swapin_proc(){
  sleep(swap_in_queue.qchan, &ptable.lock);
  while(1){
    // cprintf("\n\nEntering swapin\n");
    cprintf("|       Swapin Resumes        |  -  | -  |    Sw
    acquire(&swap_in_queue.lock);

    while(swap_in_queue.size){
      struct proc *p = dequeue(&swap_in_queue);
      flimit--;
      release(&swap_in_queue.lock);
      release(&ptable.lock);

      char* mem = kalloc();
      read_page(p->pid,((p->trapva)>>12),mem);

      acquire(&swap_in_queue.lock);
      acquire(&ptable.lock);

      swapInMap(p->pgdir, (void *)PGROUNDDOWN(p->trapva),
                PGSIZE, V2P(mem));
      wakeup1(p->chan);
    }
    // cprintf("\n\n");
    release(&swap_in_queue.lock);
    sleep(swap_in_queue.qchan, &ptable.lock);
  }

}
```

```
// Submits a request to the swapin process
void req_swapin(){
  struct proc* p = myproc();
  // cprintf("submittes request to swap-in %d\n",p-
  char my_pid[3];
  my_pid[2] = 0;
  my_pid[1] = '0' + p->pid%10;
  my_pid[0] = (p->pid/10 ? '0' + p->pid/10 : ' ');
  cprintf("| Submit Request to SwapIn  |  %s | -

  acquire(&ptable.lock);

  acquire(&swap_in_queue.lock);
  enqueue(&swap_in_queue, p); // Enqueues the proce
  wakeup1(swap_in_queue.qchan); // Wake up the Swap
  release(&swap_in_queue.lock);
  sleep((char*)p->pid, &ptable.lock); // Suspend th

  release(&ptable.lock);
  return;
}
```

The function req_swapin() is invoked in the event of a page fault in trap.c if it happens because an earlier page was swapped out. The following is done by this function:

1. Add the requesting process to the swap_in_queue.
2. Suspend the running process.

The swap out pages that were previously written to the disc are now removed after the process execution is complete and it is about to quit. To accomplish this, the delete_page_files() function is employed. It performs the subsequent:

1. Goes through the list of files to be removed iteratively.

2. Deletes the file if it has not already been done so.

```
// On exit delete the swapout page-files created
void delete_page_files()
{
  struct proc *p;

  acquire(&ptable.lock);

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

    if(p->pid==2 || p->pid==3){
      for(int fd=0; fd<NOFILE; fd++){
        if(p->ofile[fd]){
          struct file* f = p->ofile[fd];

          if(f->ref < 1){
            p->ofile[fd] = 0;
```

Task 4: Sanity Test

We created the user program test.c to evaluate how well memory swaps are performing. This process forks 20 child processes, each of which requests 4KB memory using malloc every 20 iterations.

At each memory location, we are using the following function to store the data:

If the child number = i, iteration number = j and byte number = k, then the value stored in that memory location is (i + j*k)%128.

We repeat the iteration and determine whether the value we initially stored was appropriately stored after all iterations. If the value is incorrect, an error message is printed.

```
for (int i = 1; i <= NUM_CHILD; i++){
    if (fork() != 0)
        continue;

    char *ptr[NUM];

    // Allocate 4KB to each of the char pointer
    for(int j = 0; j < NUM; j++)
        ptr[j] = (char *)malloc(PAGE_SIZE);

    // Assign values to the allocated memory
    for (int j = 0; j < NUM; j++){
        for (int k = 0; k < PAGE_SIZE; k++)
            ptr[j][k] = (i + j * k) % 128;
    }

    // Error detection to check correct fuction
    for (int j=0; j < NUM; j++){
        for (int k=0; k < PAGE_SIZE; k++){
            if (ptr[j][k] != (i + j * k) % 128)
                printf(1, "Error at i = %d, j =
        }
    }

    exit();
}

for (int i = 1; i <= NUM_CHILD; i++)
    wait();
```

Output :

```
$ test
-------------------------------------------------------------------------------------------
|          Event            | PID | VA |                     Remark                       |
-------------------------------------------------------------------------------------------
| Submit Request to SwapOut |  23 | -  |           Process 23 is queued to swapout        |
| Submit Request to SwapOut |  24 | -  |           Process 24 is queued to swapout        |
| Submit Request to SwapOut |  25 | -  |           Process 25 is queued to swapout        |
|      Swapout Resumes      |  -  | -  |    Swapout queue is non-empty => start execution |
|    Page File Creation     |  24 | 10 |       Contents of page 10 saved in 24_10.swp     |
|    Page File Creation     |  23 | 18 |       Contents of page 18 saved in 23_18.swp     |
|    Page File Creation     |  23 | 10 |       Contents of page 10 saved in 23_10.swp     |
|        Page Fault         |  -  | -  | Page fault has occured due to insufficient memory |
|        Page Fault         |  -  | -  | Page fault has occured due to insufficient memory |
| Submit Request to SwapIn  |  23 | -  |           Process 23 is queued to swapin         |
| Submit Request to SwapIn  |  24 | -  |           Process 24 is queued to swapin         |
|      Swapin Resumes       |  -  | -  |    Swapin queue is non-empty => start execution  |
|        Page Fault         |  -  | -  | Page fault has occured due to insufficient memory |
| Submit Request to SwapIn  |  23 | -  |           Process 23 is queued to swapin         |
|        Page Fault         |  -  | -  | Page fault has occured due to insufficient memory |
|    Page File Deletion     |  24 | 10 |            Page file 24_10.swp is deleted        |
|    Page File Deletion     |  23 | 18 |            Page file 23_18.swp is deleted        |
|    Page File Deletion     |  23 | 10 |            Page file 23_10.swp is deleted        |
-------------------------------------------------------------------------------------------

Total no. of Swap in: 3
Total no. of Swap out: 3
```

Submission

The submitted C21.zip contains the following:

1. Report.pdf
2. partA.patch
3. PartA folder having all the modified files for part A.
4. xv6-public-3A
5. partB.patch
6. PartB folder having all the modified files for part B.
7. xv6-public-3B