CS-344 | OS LAB | ASSIGNMENT-3 | GROUP-5 |

DRISTIRON SAIKIA, KRISHNA PANDE, NIKUNJ HEDA, NIYATI CHAUDHARY |
180101022, 180101038, 180101049, 18101053

PART A: Lazy Memory Allocation

Standard memory page allocation in xv6 is done in sysproc.c in sys_sbrk() function. In the given patch, 2 lines are commented which will increase the value of myproc()->sz, but not actually allocate the memory.
In lazy memory allocation, we allocate the memory to pages only when there is a page fault.

```c
if(tf->trapno == T_PGFLT){
  uint addr = rcr2();
  struct proc *curproc = myproc();
  uint nva = addr+curproc->sz;
  my_allocuvm(curproc->pgdir, addr, nva);
  return;
}
```

We added the shown code in the function trap() inside the file trap.c, which calls **my_allocuvm()** function which we created for this part of the assignment.
The virtual address of the address that triggered the fault is available in the cr2 register; xv6 provides the rcr2() function to read its value.

```c
int
my_allocuvm(pde_t *pgdir, uint va, uint nva)
{
  char *mem;
  uint a;
  a = PGROUNDDOWN(va);
  for(; a < nva; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, va, nva);
      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, va, nva);
      kfree(mem);
      return 0;
    }
  }
  return 0;
}
```

Since the first access might be in the middle of the page, so we used the PGROUNDDOWN function to round down to the nearest PGSIZE bytes.

We call mappages which creates translations from va (virtual address) to pa (physical address) in existing page table pgdir and returns 0 if successful, -1 if not.

The nva sent from the function call is the final address expected as per the size of the process. Thus we allocate all the pages needed for a process and handle most test cases like echo, ls & ls with pipe command.

```
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
$ echo hi
hi
$ ls | grep README
README         2 2 2286
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13632
echo           2 4 12644
forktest       2 5 8072
grep           2 6 15508
init           2 7 13224
kill           2 8 12696
ln             2 9 12592
ls             2 10 14780
mkdir          2 11 12776
rm             2 12 12752
sh             2 13 23240
stressfs       2 14 13424
usertests      2 15 56356
wc             2 16 14172
zombie         2 17 12416
console        3 18 0
```

PART B: Virtual Memory Managing: Paging

Task1:

```
//Creating kernel process
void create_kernel_process(const char *name, void(*entrypoint)()){
  struct proc* kernel_p; //declaring kernel process
  if((kernel_p = allocproc())==0){
    return;
  }

  if((kernel_p->pgdir = setupkvm()) == 0){ //in turn calls mappages for virtual to physical address maping.
    panic("out of memory?");
  }

  kernel_p->sz = PGSIZE;
  kernel_p->parent = initproc; // setting first user process as parent

  memset(kernel_p->tf, 0, sizeof(*kernel_p->tf));
  kernel_p->tf->cs = (SEG_UCODE << 3)|0;
  kernel_p->tf->ds = (SEG_UDATA << 3)|0;
  kernel_p->tf->es =  kernel_p->tf->ds;
  kernel_p->tf->ss =  kernel_p->tf->ds;
  kernel_p->tf->eflags = FL_IF;
  kernel_p->tf->esp = PGSIZE;
  kernel_p->tf->eip = (uint)entrypoint;  // setting eip to the entrypoint

  safestrcpy(kernel_p->name, name, sizeof(kernel_p->name));
  kernel_p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  kernel_p->state = SLEEPING; // process is made to sleep to suspend from execution
```

We create a kernel process for swapin and swapout functions. We allocate the process and set up its virtual to physical address mapping. We do not initialise inituvm() function since we need to always be in the user space and not enter the user virtual space. We initialise the instruction pointer of the kernel process to the address where the corresponding void functions are residing in the memory, which is passed as pointer to void function as entrypoint.

Task 2&3:

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint va;                     // Process virtual address;
  // bool va_set;                 // Process virtual address initialised or not
  // Every process will have a file to which it will write in case of swapping.
  struct file *swapFile;

  // for every process following are necessary attributes
  struct page_in_mem paging_meta_data[MAX_TOTAL_PAGES];   //Array of paging meta_data
  struct page_in_mem *head_page_in_mem;    //header to the doubly linked list for the process
  uint num_pages_in_file;
  uint num_pages_in_main_mem;
};
```

```
#define MAX_PSYC_PAGES 15  //maximum number of process's pages in the physical memory
#define MAX_TOTAL_PAGES 30 // maximum number of pages for process.

struct page {
  uint virtualAddr; //Page's virtual address
  uint pagestate; /* pagestates: NOTUSED = 0, MAINMEMORY = 1, SWAPFILE = 2 */
  uint offset_in_file;         // If page is in file, this field stores its offset inside the swap file
  uint counter;     //counter to see when last used.
  uint pages_array_index;     // The page's index in the pages array

};

// Doubly linked list for pages currently in main memory
struct page_in_mem {
  struct page pg; //There will be a page
  struct page_in_mem *next; //pointer to next page in list
  struct page_in_mem *prev; //pointer to prev page in list
};

// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

In proc.h we declared **struct page** for each page with attributes virtual address, pagestate and counter to keep track of when the page was last accessed. Struct page_in_mem represents nodes of a doubly linked list, where each of the nodes represents a page that is currently in the main memory.

Struct proc represents the structure of the proc with attributes such a pointer to a **swapfile**, for each process. An array of pages_meta_data which holds account for all pages of the process. **Head_page_in_mem pointer** pointing to the head of the linked list of pages corresponding to the process. Also we are keeping track of pages and where they are residing.

```
void update_recently_accessed(void){
  pte_t* pte = 0;
  struct proc * p;

  // Go over all processes
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p && (p->pid > 2) && (p->state != UNUSED)  && (p->state != EMBRYO) && (p->state != ZOMBIE))
    {

      // we have the head
      struct page_in_mem *head = p->head_page_in_mem;
      while(head){
        if (head->pg.pagestate == 1){
          pte = walkpgdir(p->pgdir, (void *) head->pg.virtualAddr, 0);

          if(*pte){
            head->pg.counter = time_counter++;
          }
        }
        head = head->next;
      }
    }
  }
  release(&ptable.lock);

}
```

In proc.c the **Update_recently_accessed** function updates the access time of the particular page to global counter variable value. We walk through the processes and for every process we traverse the list of pages currently in the main memory and check if the page was accessed. If it is, then we update the counter attribute else skip to the next page.

LRU Implementation:

```
struct page_in_mem *
choosePageFromPhysMem(void)
{
  struct page_in_mem *pageToRemove = myproc()->head_page_in_mem;
  struct page_in_mem *tmp;
   // If list is null (no page to swap out) --> return to calling function allocuvm
  if (!pageToRemove)
  {
    return 0;
  }
  tmp = pageToRemove;
  uint minAccessTime = tmp->pg.counter;
  while(tmp->next != 0){
    tmp = tmp->next;
    if (tmp->pg.counter < minAccessTime)
      {
        pageToRemove = tmp;
        minAccessTime = tmp->pg.counter;
      }
  }
  return pageToRemove;
}
```

Whenever we need to replace a page from the table, we choose the victim through the Least Recently Used Policy. The access time for a page will be updated every time it is accessed. Thus the page with the min access time will be the Least Recently Used. Thus we store that in the pageToRemove variable and that is replaced from the page table.

```
int createSwapFile(struct proc *p){
  char filename[30];
  int temp = p->pid;
  char z[10];
  int curr = 0;
  while(temp){
    z[curr++]=temp%10+'0';
    temp/=10;
  }
  for(int i=0;i<curr;i++){
    filename[curr-i-1] = z[i];
  }
  filename[curr++] = '_';
  for(uint i=31;i>12; i--){
    filename[curr++] = ((p->va&(1<<i))+'0');
  }
  filename[curr++] = '.';
  filename[curr++] = 's';
  filename[curr++] = 'w';
  filename[curr++] = 'p';
  filename[curr] = '\0';
  begin_op();
  struct inode * in = create(filename, T_FILE, 0, 0);
  iunlock(in);
  p->swapFile = filealloc();
  if(p->swapFile == 0){
    panic("no slot for files to store");
  }
  p->swapFile->ip = in;
  p->swapFile->type = FD_INODE;
  p->swapFile->off = 0;
  p->swapFile->readable = O_WRONLY;
  p->swapFile->writable = O_RDWR;
  end_op();
  return 0;
}
```

In fs.c the function **createSwapFile()** for each process we create a swap file for the pages being swapped out. The naming convention given in the question has been followed. The variable in of type struct inode is used to store the file created. For each process swapFile variable stores the type (FD_INODE) and the permissions of the file.

```c
case T_PGFLT:
  if(myproc()->pid <= 2){
    cprintf("T_PGFLT due to init shell\n");
    goto pf;
  }

  uint faultAddr = PGROUNDDOWN(rcr2());    //get fault address
  cprintf("REACHED PGFLT! looking for page table entry by process: %d , fault address: 0x%x\n\n", myproc()->pid, faultAddr);
  pte_t* pte = walkpgdir(myproc()->pgdir, (char*)faultAddr, 0);
  if ((*pte & PTE_PG) == 0)
  {
    goto pf;      // This is a real Page Fault (PTE_PG is off --> page does not exist on file nor on page directory)
  }
  else
  {
    // Swap pages - get the required page from the swap file and write a chosen page by policy from physical memory to the swap file instead
    if (swapPages(faultAddr) == -1)
    {
      // swapPages didn't find the page in the swap file --> ERROR, shouldn't reach here because we check PTE_PG bit
      cprintf("T_PGFLT: 'swap-pages' func didnt find the address: 0x%x in the swap file\n",faultAddr);
      goto pf;
    }
  }
  break;
```

In trap.c , we handle the actual working of page swapping so that processes whose virtual memory size is greater than physical memory size can also execute. When a page is required by a process which is currently not in the main memory, we get a page fault, i.e T_PGFLT. So as soon we get the page fault we look if it is due to a user process and note down the virtual address. We then check if the page resides in the file and call for the swapPage function to execute the swapping, so that we get a victim page from the physical memory and then swap it out from the main memory and bring the required page to the main memory.

```c
int
file2mem(struct page_in_mem * pageToSwapFromFile)
{
  int check;
  int offset = pageToSwapFromFile->pg.offset_in_file;
  // Read page buffer from file to buffer
  if (offset < 0)
  {
    cprintf("file2mem  : page not in file\n");
    return -1;
  }
  // Check if should use the system global pgdir
  pte_t * pgdir = myproc()->pgdir;
  if (use_system_pgdir == 1)
  {
    pgdir = system_pgdir;
  }
  pte_t * pte;
  pte = walkpgdir(pgdir, (void *)(PTE_ADDR(pageToSwapFromFile->pg.virtualAddr)), 0);
  char * memInPgdir;
  if((memInPgdir = kalloc()) == 0)
  {
    panic("file2mem: out of memory\n");
  }
  mappages(pgdir, (char *) pageToSwapFromFile->pg.virtualAddr, PGSIZE, V2P(memInPgdir), PTE_W | PTE_U);
  if ((check = readFromSwapFile(myproc(), memInPgdir, offset, PGSIZE)) == -1)
  {
    cprintf("file2mem: READ FAIL\n");
    return -1;
  }
  if ((pte = walkpgdir(pgdir, (void*)pageToSwapFromFile->pg.virtualAddr, 0)) == 0) {
    panic("file2mem: pte should exist");
  }
  set_present_clear_pagedout(pgdir, (char *) pageToSwapFromFile->pg.virtualAddr);
  myproc()->num_pages_in_main_mem++;
  // Update page's fields
  // pageToSwapFromFile->pg.num_times_accessed = 0;
  pageToSwapFromFile->pg.offset_in_file = -1;
  pageToSwapFromFile->pg.pagestate = 1;
  myproc()->num_pages_in_file--;
  // Insert page to physical memory pages list
```

In vm.c the function **file2mem()** reads the page from the swapfile and stores it to the memory on eviction of the page. This is used while swapping a page out from the page table. In the file memlayout.h we lowered the value of the variable PHYSTOP. to enable a successful execution of the test cases.

Task 4:

OS_3B > xv6-public > C memtest.c > main(int, char * [])

```c
int main(int argc, char *argv[])
{
    int N = 20;
     int pids[N];
     int rets[N];
    for (int i = 0; i < N; i++)
    {
        int ret = fork(); //create new child process
        if (ret == 0)
        {
            //loop with 10 iterations
            for(int j=0;j<10;j++)
            {
                //malloc and assign values in each iteration
                char *ch= (char *)malloc(sizeof(char)*4000);
                fill(ch);
                if(check(ch)==0)
                {
                    printf(1, "memory  error \n");
                }
            }
            exit();
        }
        else if (ret > 0)
        {
            pids[i] = ret;
        }
        else
        {
            printf(1, "fork error \n");
            exit();
        }
    }
```

<u>**The sanity test**</u> : As per the question we created a test program to run on the memory management system created above. The main process forks 20 child processes and each of these run a loop to malloc 4KB memory. The fill() function assigns values to the memory allocated and the check() function validates the data. In case an error is found it returns a memory error to std output. There are cprintf() statements in the code to signify the swapping in of pages whenever page faults are found.