CS-344 ASSIGNMENT-3

GROUP-09
Patnana Sai Ashrritth - 190101061
Amancha Jagruth - 190101010
Bhukya Bharath - 190101024
Adduri Sri Sai Datta - 190101003

PART-A :- LAZY MEMORY ALLOCATION

Task -1 : eliminate allocation from sbrk()

The extra memory for a current process is obtained by using sbrk() system call, but the growproc() line in the sbrk system call is commented out here. The new sbrk(n) will increment the memory size parameter of the process by n (**line 54**) and return the previous size without actually increasing the memory. When this process tries to access the extra memory, a page fault occurs. Thus generating the T_PGFLT trap to the kernel.

```c
45 int
46 sys_sbrk(void)
47 {
48   int addr;
49   int n;
50
51   if(argint(0, &n) < 0)
52     return -1;
53   addr = myproc()->sz;
54   myproc()->sz+=n;
55   //if(growproc(n) < 0)
56   //  return -1;
57   return addr;
58 }
59
```

Task -2 : Lazy allocation

We need to restrain giving memory as soon as it is requested. Rather, we give the memory when it tries to access. This is Lazy Memory Allocation. The page fault is handled by **PGFLT_handler()** in trap.c.

Working of the PGFLT_handler function:

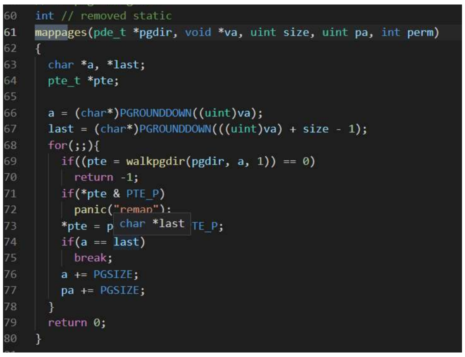
```c
  case T_PGFLT:
    if(PGFLT_handler()<0){
      cprintf("Could not allocate page. Sorry.\n");
      panic("trap");
    }
  break;
```

- This function is called when a T_PGFLT trap is generated.
- Now, rcr2() returns the virtual address at which the page fault occurs.
- rounded_addr points to the starting address to the page where this virtual address resides.(This rounded address is generated using PGROUNDDOWN macro.
- Then we call kalloc() which returns a free page from a linked list of free pages (freelist inside kmem) in the system.
- We have a physical page at our disposal. Now we need to map it to the virtual address rounded_addr which is done using mappages(), if no free pages are available, kalloc will return 0, then we will exit the function returning -1.
- To use mappages() in trap.c, we removed the static keyword in front of it in vm.c and declared its prototype in trap.c.
- mappages() takes the page directory (myproc()->pgdir) of the current process, virtual address of the start of the data where the page fault occurs, size of the data, physical memory at which the physical page resides (we give this parameter by using V2P macro which converts our virtual address to physical address by subtracting KERNBASE from it) and permissions corresponding to the page table entry as parameters.

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

int PGFLT_handler(){
  int addr=rcr2();
  int rounded_addr = PGROUNDDOWN(addr);
  char *mem=kalloc();
  // if there is memory available, we will allocate it to the process
  if(mem!=0){
    memset(mem, 0, PGSIZE);
    // maps the physical address to the virtual address
    if(mappages(myproc()->pgdir, (char*)rounded_addr, PGSIZE, V2P(mem), PTE_W|PTE_U)<0)
      return -1;
    return 0;
  } else
    return -1;
}
```

- In mappages() loop runs until all the pages from the first to last have been loaded successfully. For every page, it loads it into the page table using walkpgdir().

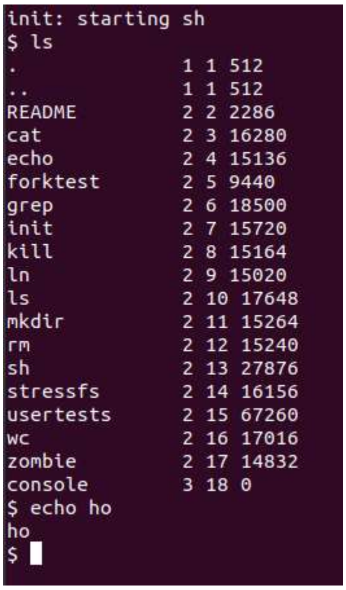


```
int // removed static
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  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 = p char *last TE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

Test cases :

Everything is working fine as the page fault is taken care of. Basic commands are running as shown in the below-left image..



```
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16280
echo           2 4 15136
forktest       2 5 9440
grep           2 6 18500
init           2 7 15720
kill           2 8 15164
ln             2 9 15020
ls             2 10 17648
mkdir          2 11 15264
rm             2 12 15240
sh             2 13 27876
stressfs       2 14 16156
usertests      2 15 67260
wc             2 16 17016
zombie         2 17 14832
console        3 18 0
$ echo ho
ho
$
```

<u>PART-B :- XV6 Memory</u>

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

- XV6 keeps a linked list of free pages called kmem in kalloc.c.
- The address to the next free page is stored in the first address of the present free page. This is one of the advantages of maintaining a linked list o free pages.
- These lists were empty initially, xv6 calls kinit from Main which then adds 4MB of free pages to the list

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

- Linked lists are used

Q3: Where do these reside?

- These linked lists reside in kalloc.c
- Every node of these linked lists is a structure defined in kalloc.c (struct run)

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

- The number of user processes are limited in xv6, due to the limit in size of ptable.

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 one process named initproc when the xv6 system boots up.
- Process can have a virtual address space of 2GB - KERNBASE and maximum physical memory of 240MB - PHYSTOP
- 1 process can take up all 240MB of physical memory, the lowest number of processes in xv6 is 1.
- There can't be zero processes after boot, since all user interactions need to be done using user processes which are forked from initproc.

Task-1: Kernel processes

In proc.c, create_kernel_process () function is created. During the whole time, the kernel process will remain in kernel mode. So, there is no need to initialize its trapframe, user space and the user section of the page table. Address of the instruction is stored in the eip register. We set the eip value of the context to the entry point as we want the process to start executing at the entry point.

```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);

}
```

Allocproc assigns the process a spot in the table.

SetupKvm sets the kernel part of the process table which maps virtual addresses above KERNBASE to physical addresses between 0 and PHYSTOP.

Task-2: swapping out mechanism

A circular queue is created as swap_req.

Swap_out_que is a specific queue, it holds the processes with swap out requests.

Functions corresponding to swap_req are also created as swap_req_push() and swap_req_pop().

Queue can be accessed with a lock that we have initialised in pinit.

We added prototypes in def.h as we need the queue and functions relating to it in other files too.

Now, in allocuvm() which is called by growproc(), whenever kalloc() is not able to allocate free pages. Then we create a SWAP_OUT_PROCESS using create_kernel_process.

```c
struct swap_req{
  struct spinlock lock; // lock to restrict access of this swap request queue
  struct proc* queue[NPROC];
  int start;
  int end;
};

// request queue for swapping out requests
struct swap_req swap_out_req;
// request queue for swapping in requests
struct swap_req swap_in_req;

struct proc* swap_req_pop(struct swap_req *q){

  acquire(&q->lock);
  if(q->start == q->end){
    release(&q->lock);
    return 0;
  }
  struct proc *p = q->queue[q->start];
  (q->start)++;
  (q->start) %= NPROC;
  release(&q->lock);

  return p;
}

int swap_req_push(struct proc *p, struct swap_req *q){

  acquire(&q->lock);
  if((q->end+1)%NPROC == q->start){
    release(&q->lock);
    return 0;
  }
  q->queue[q->end] = p;
  q->end++;
  (q->end) %= NPROC;
  release(&q->lock);

  return 1;
}
```

```c
mem = kalloc();
if(mem == 0){
  // cprintf("allocuvm out of memory\n");
  deallocuvm(pgdir, newsz, oldsz);

  //SLEEP
  myproc()->state = SLEEPING;
  acquire(&swapsleeplock);
  myproc()->chan = swapsleep;
  swapsleepcount++;
  release(&swapsleeplock);

  swap_req_push(myproc(),&swap_out_req);
  if(!SOP_PRESENT){
    // if condition to make sure that only one SWAP_OUT_PROCESS exists at a given time.
    SOP_PRESENT = 1;
    create_kernel_process("SWAP_OUT_PROCESS", &SWAP_OUT_PROCESS);
  }

  return 0;
}
```

Now, we need to change the process state to sleeping, and its channel is set to swapsleep.

So, a current process is added to swap out the request queue. (swap_out_req).

Now here, SOP_PRESENT ensures that only one swap out process exists at a given moment. This bit is set to 0 in SWAP_OUT_PROCESS function,

```c
61  kfree(char *v)
62  {
63    struct run *r;
64
65    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
66      panic("kfree");
67
68    // Fill with junk to catch dangling refs.
69    // memset(v, 1, PGSIZE);
70    for(int i=0; i<PGSIZE; i++) v[i] = 1;
71
72    if(kmem.use_lock)
73      acquire(&kmem.lock);
74    r = (struct run*)v;
75    r->next = kmem.freelist;
76    kmem.freelist = r;
77    if(kmem.use_lock)
78      release(&kmem.lock);
79
80    //Wake up processes sleeping on swapsleep channel.
81    if(kmem.use_lock)
82      acquire(&swapsleeplock);
83    if(swapsleepcount) {
84      wakeup(swapsleep);
85      swapsleepcount=0;
86    }
87    if(kmem.use_lock)
88      release(&swapsleeplock);
89  }
90
```

Now, if memory is allocated it simply maps the new pages to the virtual addresses using mappages. (not included in the screenshot).

Kalloc.c:-

We created a mechanism which wokes up all the sleeping processes in swapsleep whenever free pages are available.

Kfree in kalloc.c is edited in this way:

Processes that were preempted because of lack of availability of pages were sent to sleeping on the swapsleep.

wakeup() system call wakes up all the processes currently sleeping in swapsleep. Here we also commented on the memset line(69) as we do not want to risk erasing data before we move it to the hard disk.

Now, lets see the **SWAP_OUT_PROCESS** function.

Process runs while the loop till the swap out requests queue is not empty.

Loops runs with popping the first process in the queue and uses the LRU policy to find a victim page in the page table.

We iterate through each entry in the process table(pgdir) which thereby extracts the physical address for the secondary page.

And for the secondary page table, we iterate among the page table and look for accessed bit(A) on each of the entries.

When the secondary page table entry is found with an accessed bit unset, it chooses this entry's physical page number as the victim page.

This page then swapped out and stored to drive.

Pid and virtual address of the page to be eliminated to name the file storing this page.

New function int2str copies integers into a given string.

```c
void SWAP_OUT_PROCESS() {

  acquire(&swap_out_req.lock);
  while(swap_out_req.start != swap_out_req.end){
    struct proc *p = swap_req_pop(&swap_out_req);

    pde_t* pgdir = p->pgdir;
    for(int i=0; i<NPDENTRIES; i++){ // going throigh the page directory entries.

      //skip page table if accessed. chances are high, not every page table was accessed.
      if(pgdir[i] & PTE_A) continue;

      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pgdir[i]));
      for(int j=0; j<NPTENTRIES; j++){ // going through the the page table entries

        //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;
        // file name contians virtual address of the swaping out which helps SWAP_IN_PROCESS
        // to swap in a particular page fault at a given address by the process.
        int virt_addr = ((1<<22)*i)+((1<<12)*j);
        //file name
        char c[50];
        num_to_str(pid,c);
        int x = strlen(c);
        c[x] = '-';
        num_to_str(virt_addr,c+x+1);
        memset(&pgtab[j], 0, sizeof(pgtab[j]));

        //mark this page as being swapped out.
        pgtab[j] = ((pgtab[j])^(0x080));

        break;
      }
    }

  }

  release(&swap_out_req.lock);

  struct proc *p;
  if((p=myproc()) == 0)
    panic("swap out process");

  SOP_PRESENT = 0; // setting it zero so that another new SWAP_OUT_PROCESS can be cereated.
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED; // Killing this swapping out process.
  sched(); // calling scheduler.
}
```

When no requests are left, kernel process is suspended:

When the queue is empty, loop breaks and process suspension is initiated. We couldn't clear their kstack from inside the process while exiting the kernel processes as they won't know which process to execute next.

So, we need to clear the kstack from outside the process. We first preempt the process process and wait for the scheduler to find this process. If a kernel process in UNUSED states is found by scheduler, it clears process kstack and name. This is identified by checking its name in which the first character was changed to "*" when the process ended. And when scheduler selects a process, its access bit of ever PDE and PTE are reset. We defined another bit flag named PTE_A of value 0x020 which marks an entry accessed or not.

```
  //If the swap processes have 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;
  }

  if(p->state != RUNNABLE)
    continue;

  // we will reset the access bit of the selected process as we will just mark a recently used if it used in the last quantum of the process.
  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;
    }
  }
```

Task-3 : Swapping in Mechanism

- When the kernel detects a page fault, it must check if the cause of this page fault is in the swapping out mechanism.

In Task 2, if we swapped out a page we set its page table entries a bit of 7th order(0x080 also PTE_PS). So to check is the page was swapped out we check its 7th bit if it is set we call swap_in_process else exit.

```
void PGFLT_handler() {
  int addr=rcr2();
  struct proc *p = myproc();
  acquire(&swapinlock);
  sleep(p, &swapinlock);
  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
    // storing the address where page fault occurs. This is later used to swap in the respective file .swp file
    p->PGFLT_addr = addr;
    swap_req_push(p,&swap_in_req);
    if(!SIP_PRESENT){
      SIP_PRESENT = 1;
      create_kernel_process("SWAP_IN_PROCESS", &SWAP_IN_PROCESS);
    }
  }
  else exit();
}
```

- Swapping in the function runs a while loop until Swap_in_que is not empty. In the loop, it pops a process from the queue and extracts its pid and addr value to get the file name. Then, it creates the filename in a string called "c" using num_to_string() Then, it uses file_open() to open this file in read only mode (O_RDONLY) with file descriptor fd. We then

```
void SWAP_IN_PROCESS() {

    acquire(&swap_in_req.lock);
    while(swap_in_req.start != swap_in_req.end){
    struct proc *p = swap_req_pop(&swap_in_req);

    int pid = p->pid;
    int virt_addr = PTE_ADDR(p->PGFLT_addr);

    char c[50];
    num_to_str(pid,c);
    int x = strlen(c);
    c[x] = '-';
    num_to_str(virt_addr,c+x+1); // getting the page which existed at this va before getting swapped out.
    safestrcpy(c+strlen(c),".swp",5);

    int fd = open_file(c,O_RDONLY);
    if(fd<0){
      release(&swap_in_req.lock);
      cprintf("could not find page file in memory: %start\n", c);
      panic("SWAP_IN_PROCESS");
    }
    char *mem = kalloc();
    read_file(fd,PGSIZE,mem);

    if(mappages(p->pgdir, (void *)virt_addr, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
      release(&swap_in_req.lock);
      panic("mappages");
    }
    wakeup(p);
  }
```

allocate a free frame (mem) to this process using kalloc. We read from the file with the fd file descriptor into this free frame using read2. We then make mappages available to proc.c by removing the static keyword from it in vm.c and then declaring a prototype in proc.c. We then use mappages to map the page corresponding to addr with the physical page that got using kalloc and read into (mem). Then we wake up, the process for which we allocated a new page to fix the page fault using wakeup. Once the loop is completed.

```
      release(&swap_in_req.lock);
      struct proc *p;
    if((p=myproc()) == 0)
      panic("SWAP_IN_PROCESS");

    SIP_PRESENT = 0; // resetting the value so a new SWAP_OUT_PROCESS may be created.
    p->parent = 0;
    p->name[0] = '*';
    p->killed = 0;
    p->state = UNUSED;
    sched(); // calling the scheduler.
  }
```

Suspending kernel process when no requests are left: When the queue is empty, the loop breaks and suspension of the process is initiated just like in task 2.

Task-4 : Sanity Test

Observation in implementation:

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

int allocnum(int n){
    return n*n - 4*n + 1;
}

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

    for(int i=0; i<20; i++){
        if(fork() == 0){
            printf(1, "Child %d\n", i+1);
            printf(1, "   S.no    Matched    Error\n");
            printf(1, "--------- ------- ---------\n\n");

            for(int j=0; j<10; j++){
                int *a = malloc(4096);
                for(int k=0; k<1024; k++) a[k] = allocnum(k);

                int Matched_B = 0;
                for(int k=0; k<1024; k++) if(a[k] == allocnum(k)) Matched_B += 4;

                if(j<9) printf(1, "    %d      %dB      %dB\n", j+1, Matched_B, 4096 - Matched_B);
                else printf(1, "   %d      %dB      %dB\n", j+1, Matched_B, 4096 - Matched_B);

            }
            printf(1, "\n");

            exit();
        }
    }

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

}
```

- The main process will fork 20 child processes.
- Each child process executes a loop with 10 iterations.
- At each iteration the process will allocate 4KB of memory using malloc system call.
- next it will fill the memory with values obtained from allocnum function (which returns n^2-4*n-1).

● Num_of_bytes_matched variable counts number of bytes that are matched in the stored value and the calculated value.

Testing using different values of the PHYSTOP values

Case 1 : PHYSTOP value (0xE000000-224MB)

```
$ sanity
Child 1
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 2
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 3
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
```

Case 2 : PHYSTOP value (0x0800000)

we get the same result as case1 even after reducing the memory size.

```
$ sanity
Child 1
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 2
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 3
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
```

Case 3 : PHYSTOP value (0x0400000-4MB)

```
$ sanity
Child 1
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 2
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 3
   S.no   Matched   Error
--------- ------- ---------


    1      4096B      0B
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 4
   S.no   Matched   Error
```

Here we use 4MB because of the minimum memory needed by memory to execute Kinit1. The obtained output is the same as the previous output representing the implementation is correct.