OS Lab Assignment – 3

Group Number : M14

Akshat Jain (200123005)

Jwalit Bharatkumar Devalia (200123026)

Varun Bhardwaj (200123068)

PART A

Lazy Memory Allocation

The **Lazy Memory Allocation** for xv6 feature, which is present in the majority of contemporary operating systems, has been implemented in this section of the lab. When using the original xv6, physical memory is allocated and mapped to the virtual address space using the sbrk() system call. We updated the **sbrk()** system call in the first part to remove the memory allocation and result in a page fault. In the second phase, we updated the **trap.c** file to use lazy allocation to fix this page fault.

1. Eliminate allocation from **sbrk()**

The sbrk() system call has been updated in this section (also provided to us in the patch file). The four crucial lines of the sbrk() system call follow initial declarations and error handling.

```
1. addr = myproc()->sz;
2. myproc()->sz += n;

3. if(growproc(n) < 0)
    return -1;
4. return addr;
```

1. Assigns a addr to the start of the newly allocated region.
2. Increases the current process's size by n times.
3. Calls the **growproc(n)** function in proc.c, which allots n bytes of memory.
4. Returns the addr.

Now that line 3 has been commented out, the process will think that it has the requested amount of RAM even though it does not. When we attempt to do an operation like **echo hi or ls**, this will result in a trap error with the code 14. The code 14 corresponds to the page fault error, or **T_PGFLT**.

The modified **code** for system call **sbrk()** is a below:

```
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;
}
```

On running echo hi in the terminal, we get the following **output**.

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc
```

2. Lazy Allocation in xv6

In this section, we handle the page fault resulting from the changes in part A.1.

1. The code that causes the trap error seen in part A.1 is found in the file trap.c. This is available in the **switch(tf->trapnodefault)**'s case as seen below:

```
// In user space, assume the process misbehaved.

cprintf("pid %d %s: trap %d err %d on cpu %d ""eip 0x%x addr
0x%x--kill proc\n",myproc()->pid, myproc()->name,
tf->trapno,tf->err, cpuid(), tf->eip, rcr2());

myproc()->killed = 1;
```

2. By comparing the outcome with the output from the previous line, we can see that **rcr2()** indicates the data in control register 2, which has the incorrect virtual address. This is what later enters into **PGROUNDDOWN(va)** as input. Inside the **T_PGFLT** case. we use PGROUNDDOWN(va) to round down the virtual address to the beginning of the page boundary.
3. In **vm.c**, we have the function **allocuvm()** which is what **sbrk()** makes use of via the **growproc()** function.
4. Studying the **allocuvm()**, makes it clear that it assigns **4KB(PGSIZE)** of pages to a function making use of kalloc(), in a loop for as many pages as are needed. Our situation calls for a similar solution, however we may do without the loop and simply designate 1 page of size 4KB (PGSIZE) as and whenever a page error happens.
5. As a last point, the **mappages** function in **vm.c** should no longer have the static keyword attached to it and should instead be declared as extern in trap.c. By doing this, we will be able to call it from inside the switch case. In order to prevent fall-through and the execution of the default statements, we additionally include the break; statement.

The code changed in various files as follows:

In trap.c

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

```
case T_PGFLT:
   {
     // code from allocuvm
     // cprintf("Trap Number : %x\n", tf->trapno);
     cprintf("rcr2() : 0x%x\n", rcr2());

     uint newsz = myproc()->sz;
     uint a = PGROUNDDOWN(rcr2());
     if(a < newsz){
       char *mem = kalloc();
       if(mem == 0) {
         cprintf("out of memory\n");
         exit();
         break;
       }
       memset(mem, 0, PGSIZE);
       mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem),
PTE_W|PTE_U);
     }
     break;
   }
```

In vm.c

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

The output for running the commands like echo and ls are as follows. The trap error no longer occurs, as can be shown. The faulting virtual address for each case is also printed in our code using **cprintf**, and this is visible in the terminal output.

```
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
rcr2() : 0x4004
rcr2() : 0xbfa4
hi
$ ls
rcr2() : 0x4004
rcr2() : 0xbfa4
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13620
echo           2 4 12632
forktest       2 5 8060
grep           2 6 15496
init           2 7 13216
kill           2 8 12684
ln             2 9 12580
ls             2 10 14768
mkdir          2 11 12764
rm             2 12 12740
sh             2 13 23228
stressfs       2 14 13412
usertests      2 15 56344
wc             2 16 14160
zombie         2 17 12404
console        3 18 0
```

PART B

Few questions answered :

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

```
struct {
```

```
 struct spinlock lock;
 int use_lock;
 struct run *freelist; // linked list of free pages
} kmem;
```

kmem struct contains a linked list of free pages. The kernel uses **kalloc**() function to allocates pages. kalloc uses the **kmem linked list** to find out unused pages.

What data structures are used to answer this question?

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

kmem contains a list of pages stored as struct run freelist. It is a linked list.

Where do these reside?

Above used data structures reside in kmem struct in kalloc.c

Does xv6 memory mechanism limit the number of user processes?

```
#define NPROC        64  // maximum number of processes
```

In param.h the following is defined which limits the number of page table elements, i.e number of processes.

If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)? Upon boot up, xv6 runs the initproc function which forks the sh process and forks other processes. Hence lowest number of processes are 1. As all processes are forked by sh which is initially called by initproc, we cannot have 0 processes.

Task1:Kernel Processes

Firstly allocproc is used to return an unused process in the process table. The first unused entry is returned after setting its state to **EMBRYO**. Then **setupkvm**() is used to set up the portion of page table for kernel. Other virtual addresses greater than **KERNBASE** are mapped to physical addresses. We need not initialise the trap frame as the process will run in kernel mode. Only user space processes are required to do so. We set the size as well the parent process - **initproc** - as it is the parent which invokes other processes. We then set the process name according to the parameter name passed. Then we set the **eip** value which stores the address of the next instruction. Here, we set it as the entrypoint passed and set the state as **RUNNABLE**. This is done after locking the process table.

```
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);

}
```

Task2:Swapping Out Mechanism

It consists of several components. We must first create a process queue to keep track of the processes that were denied more memory because there were no vacant pages. A circular queue structure called **rqueue** was developed. And rqueue is the particular queue that contains processes with swap out requests. Additionally, we developed the rq-related functions **rpush**() and **rpop** (). The lock that we initialised in **pinit** must be used to access the queue. Additionally, we set the starting values of s and e in userinit to zero. We also introduced prototypes in defs.h because the queue and the functions related to it are required in other files.

proc.c:

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

//circular request queue for swapping out requests.
struct rq rqueue;
```

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

```
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)++;
  (rqueue.s)%=NPROC;
  release(&rqueue.lock);

  return p;
}
```

```
// Set up first user process.
void
userinit(void)
{
  acquire(&rqueue.lock);
  rqueue.s=0;
  rqueue.e=0;
  release(&rqueue.lock);
```

```
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;
}
```

Defs.h:

```
extern struct rq rqueue;
extern struct rq rqueue2;
int rpush(struct proc *p);
struct proc* rpop();
struct proc* rpop2();
int rpush2(struct proc* p);
```

```
struct rq;
```

Now, **kalloc** returns **zero** every time it is unable to allot pages to a process. This informs allocuvm that no memory was allocated for the required amount (**mem**=0). In this case, we must first set the process status to sleeping. Then, we must add the current process to the queue for swap out requests, **rqueue**: (* Note: The process sleeps on a unique sleeping channel named **sleeping_channel**, which is protected by a lock called **sleeping_channel_lock**.)

Vm.c:

```
struct spinlock sleeping_channel_lock;
int sleeping_channel_count=0;
char * sleeping_channel;
```

Allocuvm.c:

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

  //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);
  }

  return 0;
```

If a page isn't already allocated for this process, **create_kernel** process here generates a swapping out kernel process to do so. The swap out process exists variable, which was initialised to 0 in proc.c and declared as extern in defs.h, is set to 0 when the swap out process completes. It's initially set to 1 when it's created (as seen above). This is done to prevent the emergence of multiple swap out processes. Later on, swap_out process is explained.

Next, we create a mechanism by which whenever free pages are available, all the processes sleeping on **sleeping_channel** are woken up. We edit **kfree** in **kalloc.c** in the following way:

Basically, all processes that were preempted due to lack of availability of pages were sent sleeping on the sleeping channel. We wake all processes currently sleeping on **sleeping_channel** by calling the **wakeup()** system call.

```
void
kfree(char *v)
{

  struct run *r;
  // struct proc *p=myproc();

  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);

}
```

Swapping out process:

```
void swap_out_process_function(){

  acquire(&rqueue.lock);
  while(rqueue.s!=rqueue.e){
    struct proc *p=rpop();

    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){

      //skip page table if accessed. chances are high, not every page table was accessed.
      if(pd[i]&PTE_A)
        continue;
      //else
      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
      for(int j=0;j<NPTENTRIES;j++){

        //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;
        int virt = ((1<<22)*i)+((1<<12)*j);

        //file name
        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);

        // file management
        int fd=proc_open(c, O_CREATE | O_RDWR);
        if(fd<0){
          cprintf("error creating or opening file: %s\n", c);
          panic("swap_out_process");
        }

        if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
          cprintf("error writing to file: %s\n", c);
          panic("swap_out_process");
        }
        proc_close(fd);

        kfree((char*)pte);
        memset(&pgtab[j],0,sizeof(pgtab[j]));

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

        break;
      }
```

```
296
297             break;
298           }
299         }
300
301       }
302
303       release(&rqueue.lock);
304
305       struct proc *p;
306       if((p=myproc())==0)
307         panic("swap out process");
308
309       swap_out_process_exists=0;
310       p->parent = 0;
311       p->name[0] = '*';
312       p->killed = 0;
313       p->state = UNUSED;
314       sched();
315     }
316
```

The process runs a loop until the swap out requests queue(**rqueue1**) is non empty. A set of instructions are carried out to end the swap out process when the queue is empty . The loop begins by removing the first process from the rqueue, then it searches its page table for a victim page using the **LRU policy**. The physical address for each secondary page table is extracted by repeatedly going through each entry in the process page table (**pgdir**). We iterate through the page table for each secondary page table and check each entry's **accessed bit (A)**, which is the sixth bit from the right. If it is set, we can tell by looking at the **bitwise & of the entry and PTE A**, which is declared as **32 in mm.c.**

```
for(int i=0;i<NPDENTRIES;i++){
  //If PDE was accessed

  if(((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;
  }
}

// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchuvm(p);
```

```
void int_to_string(int x, char *c){
  if(x==0)
  {
    c[0]='0';
    c[1]='\0';
    return;
  }
  int i=0;
  while(x>0){
    c[i]=x%10+'0';
    i++;
    x/=10;
  }
  c[i]='\0';

  for(int j=0;j<i/2;j++){
    char a=c[j];
    c[j]=c[i-j-1];
    c[i-j-1]=a;
  }

}
```

Above shown code, which is found in the scheduler, essentially sets every bit that has been accessed in the process' primary page table and any secondary page tables.

Returning to the swap out process function now. The victim page is selected by the function (using the macros stated in section A report) as soon as it locates a secondary page table entry with the accessed bit unset. The following step is to replace and store this page on drive.

We use the process' pid and virtual address of the page to be eliminated to name the file that stores this page. We have created a new function called **int_to_string** that copies an integer into a given string. We use this function to make the filename using integers **pid** and **virt**. Here is that function (declared in **proc.c**):

```
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);
}
```

```
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;
}
```

We need to write the contents of the victim page to the file with the name <**pid>, <virt>.swp.** But we encounter a problem here. We store the filename in a string called **c.** File system calls cannot be called from proc.c. The solution was that we copied the **open, write, read, close etc.** functions from **sysfile.c** to **proc.c,** modified them since the **sysfile.c** functions used a different way to take arguments and then renamed them to **proc_open, proc_read, proc_write, proc_close etc.** so we can use them in proc.c. Some examples:

Now, we write a page back to storage utilising these functions. We use the proc open command to open a file with the **O_CREATE** and **O_RDWR** permissions (we imported fcntl.h with these macros). If this file doesn't already exist, O_CREATE creates it, and O_RDWR stands for read/write. **fd** is an integer that stores the file descriptor. We use **procwrite** to write the page to this file using this file descriptor. Then, this page is put to the

free page queue using **kfree** so that it is ready for usage (remember, when kfree adds a page to the free queue, we also wake up all processes sleeping on sleeping_channel). The page table entry is then cleared using memset.

For Task 3, we need to know if the page that caused a fault was swapped out or not. In order to mark this page as swapped out, we set the 8th bit from the right (2^7) in the secondary page table entry. We use xor to accomplish this task.

Suspending kernel request when no requests are left:

When the queue is empty, the loop breaks and suspension of the process is initiated. While exiting the kernel processes that are running, we can't clear their **kstack** from within the process because after this, they will not know which process to execute next. We need to clear their kstack from outside the process. For this, we first preempt the process and wait for the scheduler to find this process. When the scheduler finds a kernel process in the **UNUSED** state, it clears this process' kstack and name. The scheduler identifies the kernel process in unused state by checking its name in which the first character was changed to **'*'** when the process ended.

Thus the ending of kernel processes has two parts:

- From within process:

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

  swap_out_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

- From scheduler:

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

  //If the swap out process has 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;
  }
```

Task3:Swapping In Mechanism

To begin with, a swap in request queue needs to be created. We created a swapped request queue called **rqueue2** in proc.c using the same struct (rq) as in Task 2. In defs.h, we additionally declare an extern prototype for rqueue2. We developed the matching rqueue2 functions (**rpop2**() and **rpush2**()) along with the queue and defined their prototypes in defs.h and proc.c, respectively. Additionally, we initialised **pinit's** lock. Additionally, we set its **s and e variables to zero in userinit.**

Next, we add an additional entry to the **struct proc** in **proc.h** called **addr (int).** This entry will tell the swapping in function at which virtual address the page fault occurred:

proc.h (in struct proc):

```
  char name[16];               // Process name (debugging)
  int addr;                    // ADDED: 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():**

trap.c:

```
    break;
  case T_PGFLT:
    handlePageFault();
  break;
  //PAGEBREAK: 13
```

```
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();
  }
}
```

Similar to Part A, **handlePageFault** uses **rcr2** to identify the virtual address at which the page fault occurred (). Then, using a new lock named swap in lock (initialised in trap.c and with extern in defs.h), we put the active process to sleep. The page table entry for this location is then obtained (the rationale is the same as **walkpgdir**). We must now determine if this page was switched out. When changing a page in Task 2, we set the page table entry's bit of the seventh order (2^7).

Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using **bitwise & with 0x080.** If it is set, we initiate **swap_in_process (if it doesn't already exist - check using swap_in_process_exists).** Otherwise, we safely suspend the process using exit() as the assignment asked us to do.

Now, we go through the **swapping_in_process.** The entry point for the swapping out process is **swap_in_process_function (declared in proc.c)** as you can see in handlePageFault

Note: swap_in_process_function is shown on the next page since it is long. Refer to the next page for the actual function.

In the Task 2 section of the report, I've previously discussed how we introduced file management features in proc.c. Here, I'll just list the functions I used and how I applied them. The loop in the function continues until rqueue2 is not empty. In the loop, a process is selected from the queue, and its pid and addr values are extracted to get the file name. After that, it uses int to string to generate the filename as a string called "c." It then opened this file in read-only mode (O_RDONLY) using proc open and file descriptor fd. Then, using kalloc, we give this process access to a free frame (mem).

We read from the file with the fd file descriptor into this free frame using proc_read. 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, we run the kernel process termination instructions.

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

```
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: Sanity Tests

memtest.c is the user program which will test the above implemented features in xv6. We will also modify the Makefile to add memtest in UPROGS (User Programs). The code is as follows :

memtest.c > ...

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

#define CRCT(x) 4096 - x

int numGenerator(int num){
    int smallNum = num%10;
    return (5*smallNum + 11*smallNum*smallNum + 17*smallNum*smallNum*smallNum + 43*smallNum*smallNum*smallNum*smallNum);
}

void validator(int ind, int num, int* correct){
    if (numGenerator(ind) == num) *correct+=4;
    return;
}

int main(int argc, char *argv[])
{
    for (int i=1; i<=20; i++)
        if (fork() == 0){
            int *ptr[10];
            printf(1, "Child ID: %d\n", i);
            for (int j=0; j<10; j++) {
                ptr[j] = (int *)malloc(4096);
                for (int k=0; k<1024; k++)
                    ptr[j][k] = numGenerator(k);
            }

            for (int j=0; j<10; j++){
                int correctBytes=0;
                for (int k=0; k<1024; k++){
                    validator(k,ptr[j][k],&correctBytes);
                }
                printf(1,"Iteration Numebr : %d,     Incorrect Bytes: %d\n",j+1,CRCT(correctBytes));
            }
            printf(1,"\n");
            exit();
        }

    for (int i=0; i<20; i++){
        wait();
    }
    exit();
}
```

Output (shown for the last 2 children, same for all 20):

```
Child ID: 19
Iteration Numebr : 1,     Incorrect Bytes: 0
Iteration Numebr : 2,     Incorrect Bytes: 0
Iteration Numebr : 3,     Incorrect Bytes: 0
Iteration Numebr : 4,     Incorrect Bytes: 0
Iteration Numebr : 5,     Incorrect Bytes: 0
Iteration Numebr : 6,     Incorrect Bytes: 0
Iteration Numebr : 7,     Incorrect Bytes: 0
Iteration Numebr : 8,     Incorrect Bytes: 0
Iteration Numebr : 9,     Incorrect Bytes: 0
Iteration Numebr : 10,     Incorrect Bytes: 0

Child ID: 20
Iteration Numebr : 1,     Incorrect Bytes: 0
Iteration Numebr : 2,     Incorrect Bytes: 0
Iteration Numebr : 3,     Incorrect Bytes: 0
Iteration Numebr : 4,     Incorrect Bytes: 0
Iteration Numebr : 5,     Incorrect Bytes: 0
Iteration Numebr : 6,     Incorrect Bytes: 0
Iteration Numebr : 7,     Incorrect Bytes: 0
Iteration Numebr : 8,     Incorrect Bytes: 0
Iteration Numebr : 9,     Incorrect Bytes: 0
Iteration Numebr : 10,     Incorrect Bytes: 0

$ akshat@4KxH4T:~/OSLab/Assignment 3/sub/xv6-public$
```

Explanation :

We test our code as follows: 1. The parent process memtest forks 20 identical child processes, where each of them run a loop of iterations. In each iteration, The child calls malloc(4096) to request additional memory and uses it as an int array. We iterate over this array and fill it with an integer calculated as a function of the index i using `numGenerator(i).` 2. After all the iterations are completed, meaning 10 arrays allocated, we then run a validation loop. The validation loop iterates again over the array, and matches the array value stored at index i with the value assigned earlier (numGenerator(i)). 3. We maintain a counter variable correctBytes for each array to track the number of bytes correctly matched. Finally, we report the number of incorrectly matched bytes (4096 - correctBytes). We also ran the code against lower values of PHYSTOP. In particular, we also set it to 0x0400000 (meaning 4MB of memory), which is the least needed by xv6 to execute kinit1 (as seen in main.c and xv6 documentation). In all cases, the output obtained is identical as shown above. The number of mismatched bytes are 0 in all iterations for all children, which means our implementation works correctly