CS344
Lab Assignment-3
Group No- M23

Group members –

1- Harshul Gupta (200123023)
2- Arti Sahu (200123011)
3- Sunny Narzary(200123062)

Part A: Lazy Memory Allocation

After applying the patch file, and running echo hi, the following output was obtained as expected.

```
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
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$
```

Whenever the current process needs extra memory than its assigned value, it indicates this requirement to the xv6 OS using sbrk system call. sbrk uses growproc() defined inside proc.c to cater to this requirement.

Lazy allocation simply means not allocating a resource until it is actually needed. We do this by commenting out the call to growproc() inside the sbrk system call. When this process tries to access the page (which it thinks hasbeen already brought inside memory), it encounters a PAGE FAULT, thus generating a T_PGFLT trap to the kernel. This is handled in trap.c by calling handlePageFault().

- rcr2(): gives the virtual address at which the page fault occurs.
- rounded__addr: points to the starting address to the page where this virtual address resides. Then we call kalloc() which returns a free page from a linked list of free pages (freelist inside kmem) in the system.
- Now we need to map it to the virtual address rounded__addr which is done using mappages(). To use mappages() in trap.c, we remove the static keyword in front of it in vm.c and declare its prototype in trap.c.
- mappages(): takes the page table of the current process, virtual address of the start of the data, size of the data, physical memory at which the physical page resides and permissions corresponding to thepage table entry as parameters.

```
  // trap 14
case T_PGFLT:
  if(handlePageFault()<0){
    cprintf("Could not allocate page. Sorry.\n");
    panic("trap");
  }
```

```
int handlePageFault(){
  int addr=rcr2();   //rcr2() gives the virtual address at which the page fault occurs
  int rounded_addr = PGROUNDDOWN(addr); //rounded _addr points to the starting address to the page
                                        //where this virtual address resides
  char *mem=kalloc(); //kalloc() which returns a free page
                      //from a linked list of free pages (freelist inside kmem) in the system.
  if(mem!=0){
    memset(mem, 0, PGSIZE);
    if(mappages(myproc()->pgdir, (char*)rounded_addr, PGSIZE, V2P(mem), PTE_W|PTE_U)<0)
      return -1;
    return 0;
  } else
    return -1;
}
```

In vm.c -

```
int
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 = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

- In this, ‘a’ denotes the first page and ‘last’ denotes the last page of the data that has to be loaded. It then runs a loop 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().

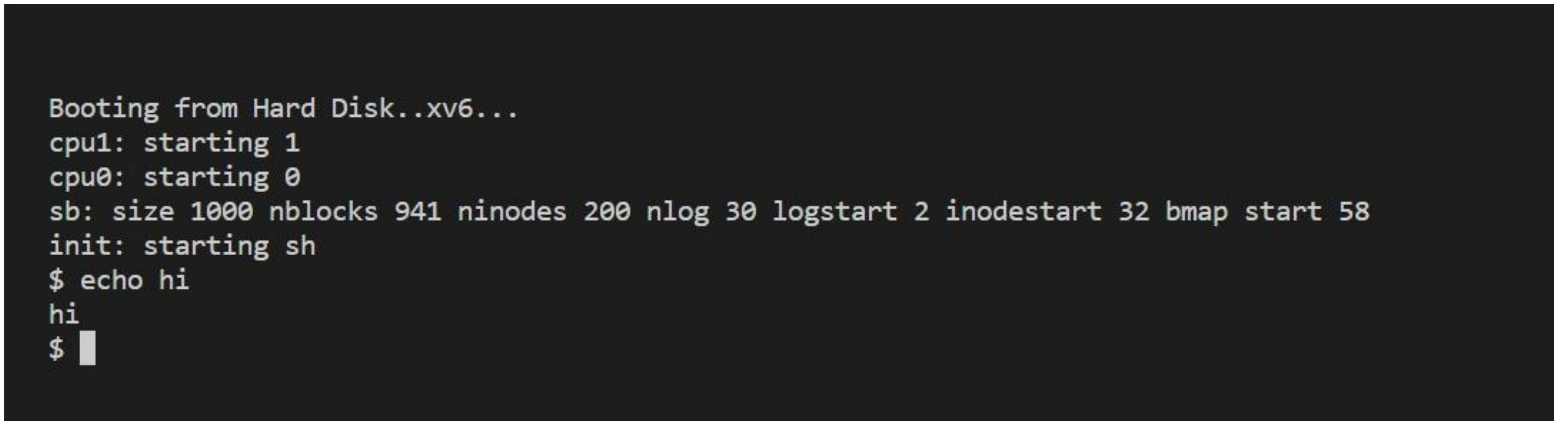
```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

- walkpgdir() takes a page table and a virtual address as input and returns the page table entry corresponding to that virtual address inside the page table. After this, we return apointer to the page table entry corresponding to the virtual address.

- Now, the mappages() knows the entry to which the current virtual address has to be mapped. It checks if the PRESENT bit of that entry is already set indicating that it is already mapped to some virtual address. If yes, it generates an error telling that remap has occurred. If no error takes place, it associates the page table entry to the virtual address, sets the permission bits and sets its PRESENT bit indicating that the current page table entry has been mapped to a virtual address.

Final Output -



ANSWERS TO PART-B QUESTIONS

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

Ans) xv6 maintains a linked list of free pages in kalloc.c called kmem.

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

<Q2> What data structures are used to answer this question?

Ans) A linked list named freelist. Every node of the linked list is a structure defined in kalloc.c namely struct run (pagesare typecast to (struct run *) when inserting into freelist in kfree(char*v)).

<Q3> Where do these reside?

Ans) This linked list is declared inside kalloc.c inside a structure kmem. Every node is of the type struct run which is also defined inside kalloc.c.

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

Ans) Due to a limit on the size of ptable (a max. of NPROC elements which isset to 64 by default), the number of user processes are limited in xv6. NPROC is defined in param.h.

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

Ans) When the xv6 operating system boots up, there is only one process named initproc. Also, since a process can have a virtual address space of 2GB (KERNBASE) and the assumed maximum physical memory is 240MB (PHYSTOP), one process can take up all of the physical memory . Hence, the answer is 1.

Part B:

Task 1: Kernel Processes

The **create_kernel_process()** function was created in **proc.c**. The kernel process will remain in kernel mode the whole time.

The **eip** register of the process' context stores the address ofthe next instruction. We want the process to start executing at the entry point. Thus, we set the eip value of the context to entry point.

- allocproc assigns the process a spot in ptable.
- setupkvm sets up the kernel part of the process' page table that maps virtual addresses above KERNBASE to physical addresses between 0 and PHYSTOP.
- Name of the new process is set as name using safestrcpy (np->name, name, size of(name));

proc.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);

}
```

Task 2: Swapping out mechanism

We need a **process queue** that keeps track of the processes that were refused additional memory since there were no free pages available. We created a **circular queue struct** called **rq.** And the specific queue that holds processes with **swap out requests** is **rqueue.**

We have also created the functions corresponding to rq, namely **rpush()** and **rpop().** The queue needs to be accessed with a lock that we have initialized in **pinit**. We have also initialized the initial values of **s** and **e** to zero in **userinit.**

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;

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;
}

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;
}
```

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

```
void
userinit(void)
{
  acquire(&rqueue.lock);
  rqueue.s=0;
  rqueue.e=0;
  release(&rqueue.lock);
```

Now, whenever kalloc is not able to allocate pages to a process, it returns zero. This notifies allocuvm that the requested memory wasn't allocated (mem=0). Here, we first need to change the process state to sleeping. Then, we need to add the current process to the swap out request queue, rqueue:

- Global declarations (vm.c):

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

- 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 lackof 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);

}
```

- **swap_out_process_function**

The process runs a loop until the swap outrequests queue (**rqueue1**) is non empty. When the queue is empty, a set of instructions are executed for the termination of **swap_out_process .** The loop starts by popping the first process from **rqueue** and uses the LRU policy to determine a victim page in its page table. We iterate through each entry in the process' page table (**pgdir**) and extracts the physical address for each secondary page table.

For each secondary page table, we iterate through the page table and look at the **accessed bit (A)** on each of the entries (The accessed bit is the sixth bit from the right. We check if it is set by checking the **bitwise &** of the entry and **PTE_A (which we defined as 32 in mmu.c)**).
This code resides in the scheduler and it basically unsetsevery accessed bit in the process' page table and its secondary page tables.

```
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;
      }
    }
  }
  release(&rqueue.lock);

  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();
}
```

As soon as the function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number as the victim page. This page is then swapped out and stored to 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**.

```
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;
  }

}
```

We need to write the contents of the victim page to the filewith 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.

.

After this, we do something important: 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 process 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 firstpreempt 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:

1. from within process:

```
  release(&rqueue.lock);

  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();
}
```

2. From scheduler

```
//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;
}
```

Task 3: Swapping in mechanism

We first need to create a swap in request queue. We used the same struct (rq) as in Task 2 to create a swap in request queue called rqueue2 in proc.c. We also declare an extern prototype for rqueue2 in defs.h. Along with declaring the queue, we also created the corresponding functions for rqueue2 (rpop2() and rpush2()) in proc.c and declared their prototype in defs.h. We also initialised its lock in pinit. We also initialised its s and e variables in userinit.

```
//circular request queue for swapping in requests
struct rq rqueue2;

struct proc* rpop2(){

        acquire(&rqueue2.lock);
        if(rqueue2.s==rqueue2.e){
                release(&rqueue2.lock);
                return 0;
        }
        struct proc* p=rqueue2.queue[rqueue2.s];
        (rqueue2.s)++;
        (rqueue2.s)%=NPROC;
        release(&rqueue2.lock);
        return p;
}

int rpush2(struct proc* p){
        acquire(&rqueue2.lock);
        if((rqueue2.e+1)%NPROC==rqueue2.s){
                release(&rqueue2.lock);
                return 0;
        }
        rqueue2.queue[rqueue2.e]=p;
        (rqueue2.e)++;
        (rqueue2.e)%=NPROC;

        release(&rqueue2.lock);
        return 1;
}
```

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 :

```
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)
  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():**

In handlePageFault, we find the virtual address at which the page fault occurred by using rcr2(). We then put the current process to sleep with a new lock called swap_in_lock We then obtain the page table entry corresponding to this address. Now, we need to check whether this page was swapped out.

trap.c:

```
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();
  }
}
```

In Task 2, whenever we swapped out a page, we set its page table entry's bit of 7th 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_ Otherwise, we safely suspend the process using exit() .

Now, we go through the swapping in process. The entry point for the swapping out process is swap_in_process_function as you can see in handlePageFault.

```
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();

}
```

The function runs a loop until rqueue2 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 int_to_string. Then, it used proc_open to open this file in read only mode (O_RDONLY) with file descriptor fd. We then 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 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);
```

Task 4: Sanity Test

In this part, our aim is to create a testing mechanism in order to test the functionalities created by us in the previous parts. We will implement a user-space program named memtest that will do this job for us. The implementation of memtest is given below.

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

int math_func(int num){
        return num*num - 4*num + 1;
}

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

        for(int i=0;i<20;i++){
                if(!fork()){
                        printf(1, "Child %d\n", i+1);
                        printf(1, "Iteration Matched Different\n");
                        printf(1, "--------- ------- ---------\n\n");

                        for(int j=0;j<10;j++){
                                int *arr = malloc(4096);
                                for(int k=0;k<1024;k++){
                                        arr[k] = math_func(k);
                                }
                                int matched=0;
                                for(int k=0;k<1024;k++){
                                        if(arr[k] == math_func(k))
                                                matched+=4;
                                }

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

                        }
                        printf(1, "\n");

                        exit();
                }
        }

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

}
```

We can make the following observations by looking at the implementation:

- The main process creates 20 child processes using fork() system call.
- Each child process executes a loop with 10 iterations
- At each iteration, 4096B (4KB) of memory is being allocated using malloc()
- The value stored at index i of the array is given by the mathematicalexpression $i^2 - 4i + 1$ which is computed using math_func().
- A counter named matched is maintained which stores the number of bytes that contain the right values. This is done by checking the value stored atevery index with the value returned by the function for that index.

In order to run memtest, we need to include it in the Makefile under UPROGS and EXTRA to make it accessible to the xv6 user.
On running memtest, we obtain the following output:

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ memtest
Child 1
Iteration Matched Different
--------- ------- ---------

    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
Iteration Matched Different
--------- ------- ---------

    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
Iteration Matched Different
--------- ------- ---------

    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
Iteration Matched Different
--------- ------- ---------

    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 5
Iteration Matched Different
--------- ------- ---------

    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 6

Iteration	Matched	Different
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 7

Iteration	Matched	Different
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 8

Iteration	Matched	Different
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 9

Iteration	Matched	Different
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 10

Iteration	Matched	Different
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 11

Iteration	Matched	Different
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 12

Iteration	Matched	Different
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 13

Iteration	Matched	Different
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 14

Iteration	Matched	Different
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 15

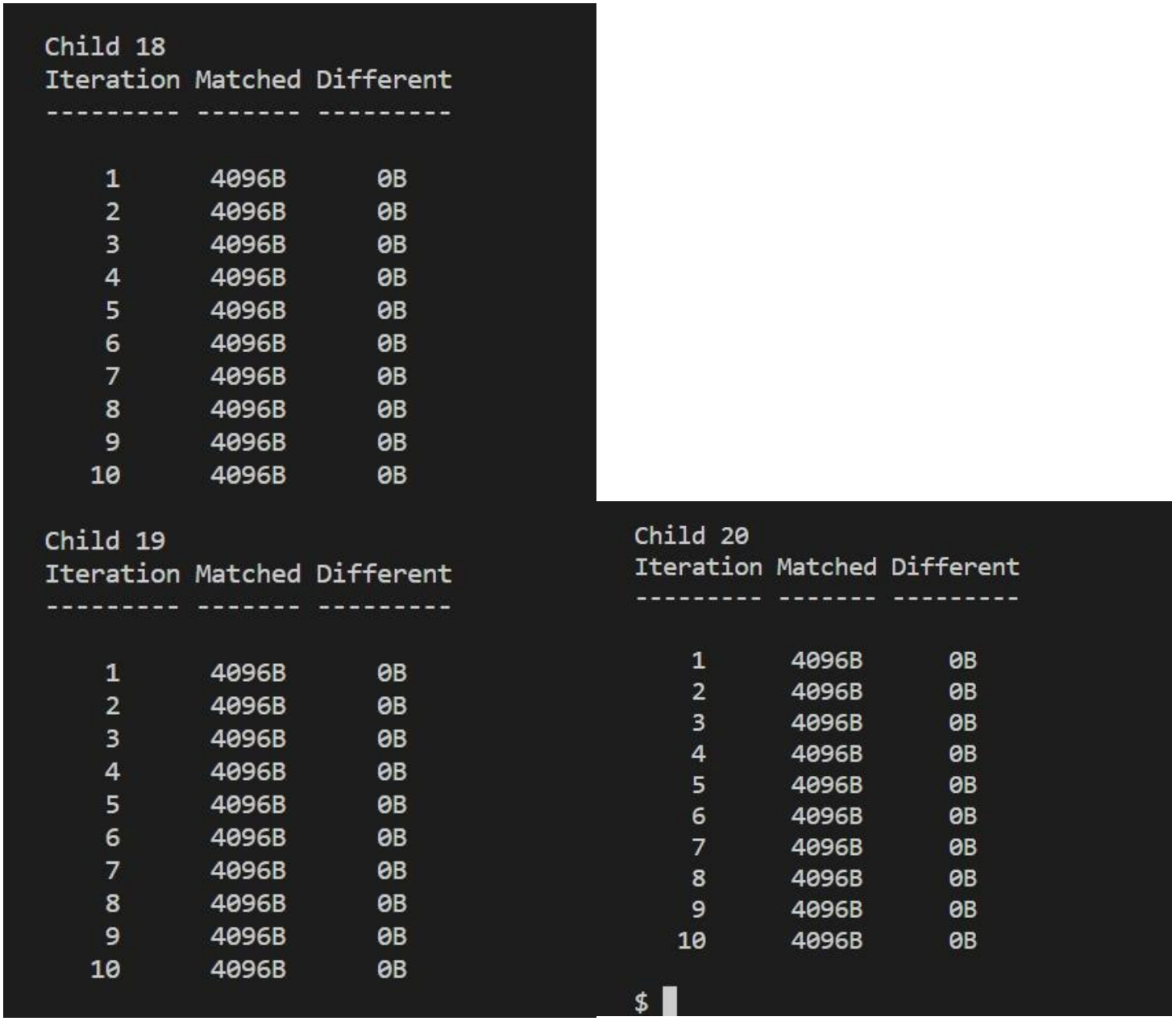
Iteration	Matched	Different
1	4096B	0B
2	4096B	0B
3	4096B	0B
4	4096B	0B
5	4096B	0B
6	4096B	0B
7	4096B	0B
8	4096B	0B
9	4096B	0B
10	4096B	0B

Child 16

Iteration	Matched	Different
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 17

Iteration	Matched	Different
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



As can be seen in the output, our implementation passes the sanity test as all theindices store the correct value.

Now, to test our implementation even further, we run the tests on different values of PHYSTOP (defined in memlayout.h). The default value of PHYSTOP is OxEOOOOOO (224MB). We changed its value to 0x0400000 (4MB). We chose 4MB because this is the minimum memory needed by xv6 to execute kinit1. On running memtest, the obtained output is identical to the previous output indicating that the implementation is correct.