

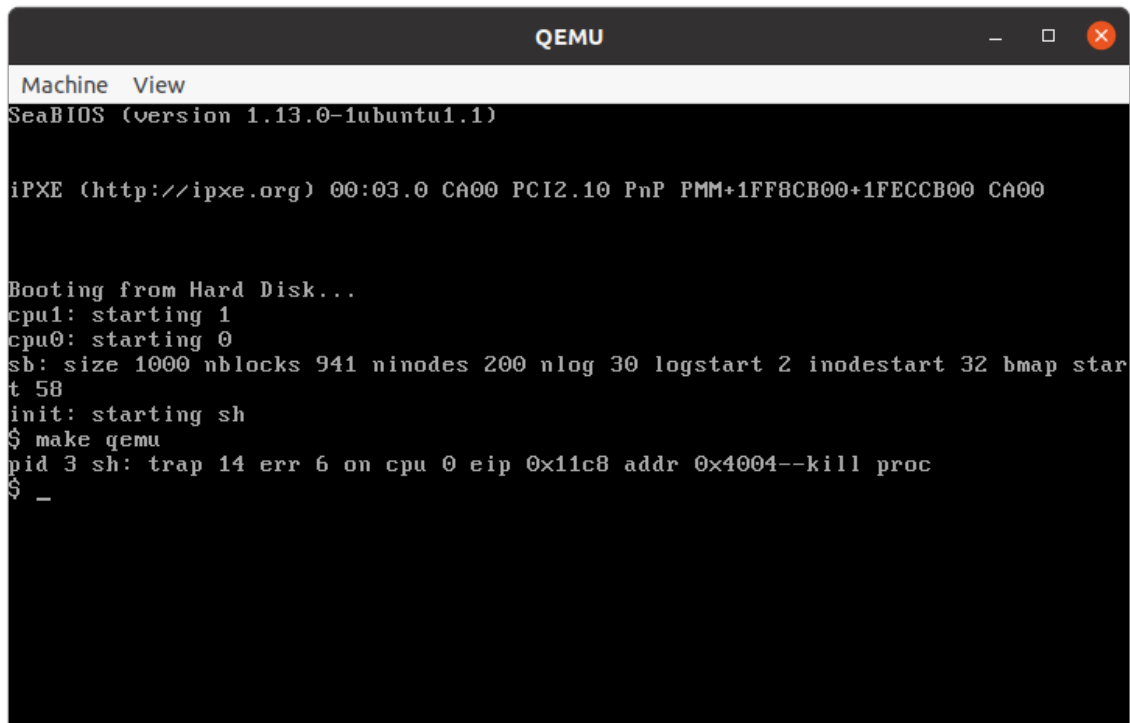
Assignment 3

Group C6:

Vani Krishna Barla	200101101
Sidharth Bansal	200101096
Mahek Vora	200101062

Part A

1. Eliminate allocation from sbrk()



```
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
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
$ make qemu
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x11c8 addr 0x4004--kill proc
$ -
```

When `sbrk(n)` is called the function only increments the process size by `n` and return the old size. It does not allocate memory, because the call to `growproc()` is commented out.

2. Lazy Allocation

To implement Lazy Allocation, the process is not assigned memory by the `sbrk` call and is rather allocated memory when the process triggers a page fault which is denoted by the **trap 14** error in the terminal in the previous part.

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

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

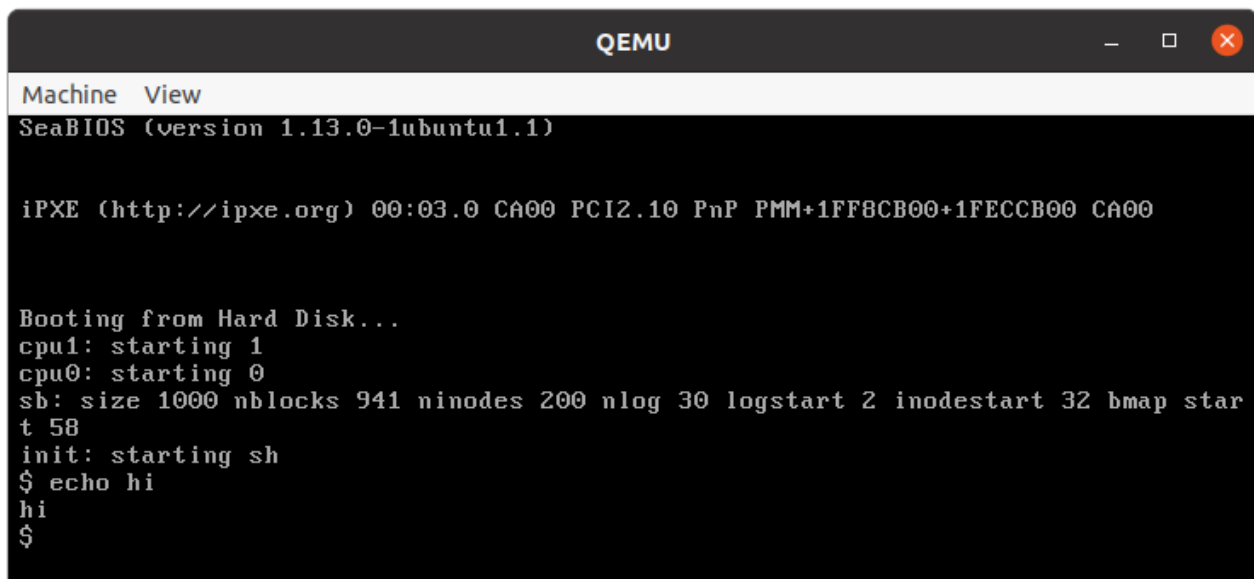
    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;

    myproc()->sz += n;
    // if(growproc(n) < 0)
    //     return -1;
    return addr;
}
```

The `tf->trapno` when equal to `T_PGFLT` indicates the occurrence of page fault.

The `handlePageFault()` function first finds the virtual address that causes the page fault, which is returned by `rcr2()`. `PGROUNDDOWN()` rounds the VA to the start of the page boundary. `kalloc()` returns a pointer to memory and `memset` sets the size to `PageSize` of 4096 bytes. `mappages()` enters the mapping of virtual address to corresponding physical address into the page directory.

```
int handlePageFault() {
    int addr=rcr2();
    int rounded_addr = PGROUNDDOWN(addr);
    char *mem=kalloc();
    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;
    }
    return -1;
}
```



```
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
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
$
```

Part B

Questions:

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

It keeps a linked list of free physical pages; xv6 deletes newly allocated pages from the list, and adds freed pages back to the list. xv6 calls `kinit1` through `main()` which adds 4MB of free pages to the list initially.

→ What data structures are used to answer this question?

A linked list named `freelist` where ever node of the linked list is a structure defined in `kalloc.c` namely `struct run` (pages are typecast to `struct run *` when inserting into the `freelist` in `kfree(char *v)`).

→ Where do these reside?

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`

→ Does xv6 memory mechanism limit the number of user processes?

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

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

When the v operating system boots up there is only one process named `initproc` this process forks the `sh` process which forks other user processes. 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

Task 1: Kernel processes

```
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: out of memory?");  
    //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);  
}
```

The `create_kernel_process()` follows the code of `userinit()` function in `proc.c` by calling `allocproc()` and updating the page table & directory and mapping the virtual address above `KERNBASE` to physical addresses between 0 and `PHYSTOP` by calling `setupkvm()`. Since the process will remain in the kernel the whole time we do not need to initialize its trapframe (trapframes store userspace register values), user space and the user section of its page table. The `eip` register of the process context stores the address of the next instruction. We want the process to start executing at the entry point (which is a function pointer), thus setting the `eip` to the entry point. Then, acquire process table lock to update the state of the newly created process to `RUNNABLE`.

Task 2: swapping out mechanism

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** and initialized the initial values of `s` & `e` to zero in `userinit`.

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

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

```
//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;
}
```

```
197 int rpush(struct proc *p){
198
199     acquire(&rqueue.lock);
200     if((rqueue.e+1)%NPROC==rqueue.s){
201         release(&rqueue.lock);
202         return 0;
203     }
204     rqueue.queue[rqueue.e]=p;
205     rqueue.e++;
206     (rqueue.e)%=NPROC;
207     release(&rqueue.lock);
208
209     return 1;
210 }
211
212
213
```

Firstly, whenever `kalloc` is not able to allocate pages to a process, it returns zero. This notifies `allocvm` that the requested memory wasn't allocated (`mem=0`). Here the process state needs to be changed to sleeping. The process sleeps on a special sleeping channel called `sleeping_channel` that is secured by a lock called `sleeping_channel_lock`. `sleeping_channel_count` is used for corner cases when the system boots. Then, we need to add the current process to the swap out request queue, `rqueue`:

In `vm.c`, `allocvm()` :

```
if(mem == 0){
    // cprintf("allocvm out of memory\n");
    deallocvm(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;
}

```

create_kernel_process here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. When the swap out process ends, the swap_out_process_exists variable is set to 0. When it is created, it is set to 1 (as seen above). This is done so multiple swap out processes are not created.

```

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);
}

```

In kalloc.c :

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: Thus, 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.

Now, the swap_out_process function() : The process runs a loop until the swap out requests queue (rqueue1) is non empty. When the queue is empty, a set of instructions are executed for the termination of this 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 extract the physical address for each secondary page table.

In mmu.h:

```
#define PTE_A 0x020 //Accessed
```

```

for(int i=0;i<NPENTRIES;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;
    }
}

```

```

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<NPENTRIES;i++){

            //skip page table if accessed. chances are high, not even
            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);
                strncpy(c+strlen(c),".swp",5);

                // file management
                int fd=proc_open(c, 0_CREATE | 0_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);
}

```

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.

It is reset in scheduler() and it basically unsets every accessed bit in the process page table and its secondary page tables.

In swap_out_process_function, as soon as the

function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number (using macros mentioned in part A report) as the victim page. This page is then swapped out and stored to drive.

We use the process pid (pid, line 267 in image) and virtual address of the page to be eliminated to name the file that stores this page.

```

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();
}

int proc_read(int fd, int n, char *p)
{
    struct file *f;
    if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
        return -1;
    return fileread(f, p, n);
}

```

For suspending the 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 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 has changed to '*' when the process ended.

Thus the ending of kernel processes has two parts:

1. from within process
2. from scheduler

```
if(p->state==UNUSED && p->name[0]=='*'){
    kfree(p->kstack);
    p->kstack=0;
    p->name[0]=0;
    p->pid=0;
}
```

Task 3: Swapping in process

We used the same struct (rq) as in Task 2 to create a swap in request queue called rqueue2 in proc.c. Along with declaring the queue, we also created the corresponding functions for rqueue2 (rpop2() and rpush2()).

Next we need to handle page fault (T_PGFLT) traps raised in trap.c. We do it in a function called handlePageFault().

In trap.c:

```
case T_PGFLT:
    handlePageFault();
break;
```

```
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 `handlePageFault`, just like Part A, 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 (the logic is identical to `walkpgmdir`). Now, we need to check whether this page was swapped out. In Task 2, whenever we swapped out a page, we set its page table entries 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_process`. Otherwise, we safely suspend the process using `exit()`.

In 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`.

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 we 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*.

```

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:

Our aim in this part is to make a program memtest, to test the functionalities developed in the previous parts.

The program memtest creates 20 processes using fork(). In each child process, we allocate 4096 bytes of memory using malloc 10 times. In each page of memory, we assign the value $i^2 - 4*i + 1$, computed by math_func to the index i (stored as integers).

We then check the values at each page by comparing against the values calculated by math_func and store the number of correctly stored bytes in matched.

We then print the number of bytes matched for every page in every child.

```

#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();
}

```

```

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

```

We run memtest for the usual value of PHYSTOP (0xE000000) and also for a smaller value 0x0400000. All the bytes match in both cases indicating that our implementation is correct.