Assignment 3

G26 Members:

Abhishek Agrahari

Vivek Kumar

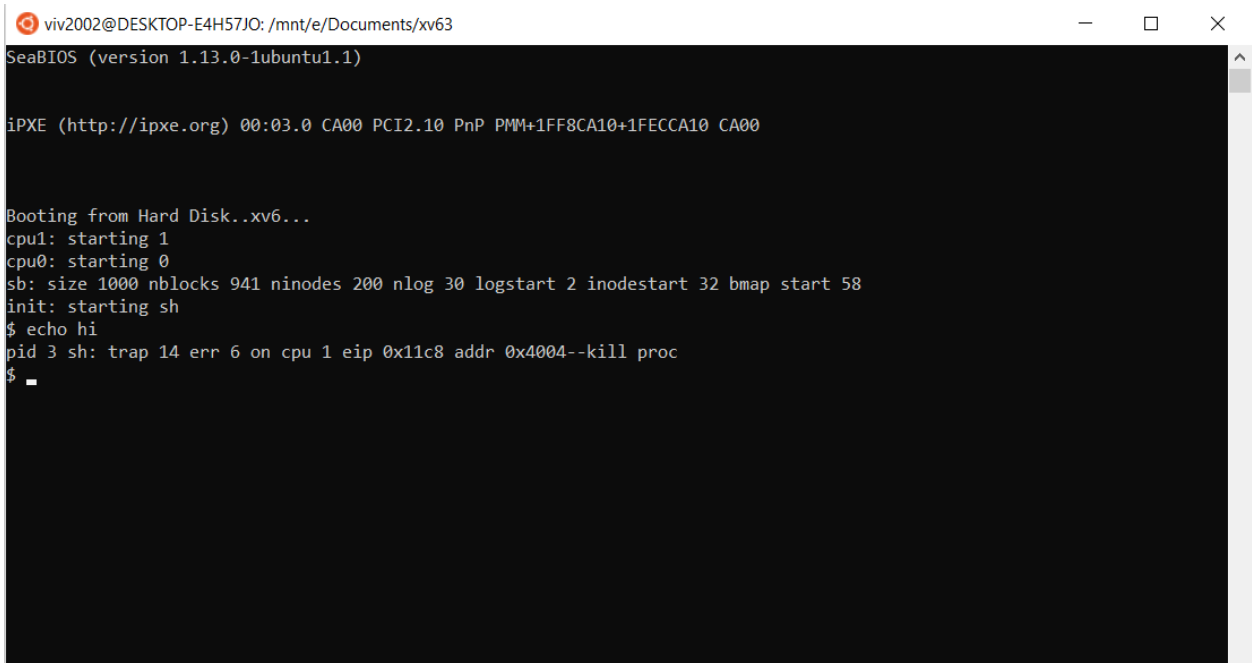
Manish Kumar

Anubhav Bajaj

B.K. Uday Singh

Part A: Lazy Memory Allocation

At first, we applied the provided patch which eliminates the allocation of pages from sbrk.
After it we run $echo hi command and the output shown was:

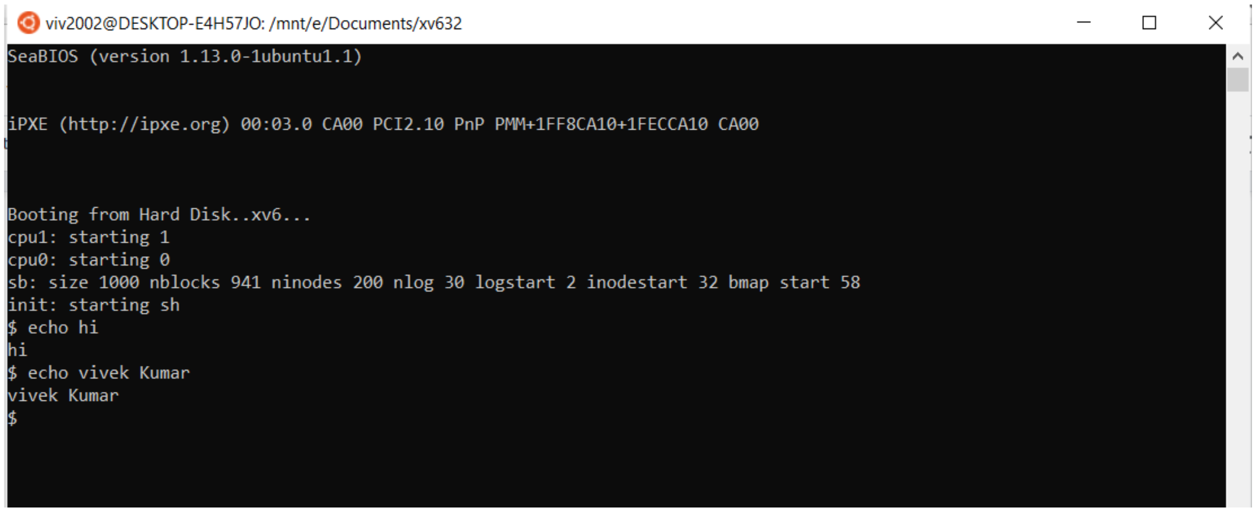


viv2002@DESKTOP-E4H57JO: /mnt/e/Documents/xv63

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


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



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
$ 
```

This error was because no memory allocation was done so it was a page fault.
To handle this we implemented handlePageFault function in trap.c which responds to a page fault by assigning newly allocated page of physical memory at the virtual address where page fault occurred. This information was provided to the function by rcr2().
After implementing the function we put the prototype of mappages function in trap.c.
In trap.c we checked whether the fault was a page fault by checking tf->trapno and then in case of page fault we called handlePageFault function which performed the required assignments.

And then we again run $echo hi command and now the output was:



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


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



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
hi
$ echo vivek Kumar
vivek Kumar
$
```

This shows we have correctly implemented lazy memory allocation.

Part B: Disk Swapping of User Process Pages

Answers of questions given in Part B-

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

Xv6 maintains a linked list of free pages in kalloc.c called kmem. Freelist in this struct is the head of this linked list and each node is of struct run.

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

The list of free pages are maintained as a linked list freelist as shown in the above image.

Q3. Where do these reside?

Each free page's run structure is stored within the page itself, since there is

nothing else stored there.

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

Yes, in xv6 number of user processes that can be created is limited by NPROC = 64

defined in param.h

Q5. 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 xv6 operating system boots up, there is only one process named **initproc** (this process forks the sh process which forks other user processes). Hence the answer is 1.

Task 1:

The **create_kernel_process()** function was created in **proc.c**. The kernel process will remain in kernel mode the whole time. Thus, **we do not need to initialise 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, **we set the eip value of the context to entry point** (Since entry point is the address of a function). **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.**

Task 2

- In swap out mechanism whenever a request to a page which isn't in the main memory is made and physical memory is full, we select a victim by using Least Recently Used algorithm.
- An additional parameter is maintained with all the page table entries which stores the timestamp of the instant when the page was last referenced.

- We iterate through the page tables of all the processes in the process table and select the page table entry having the least time stamp as the victim.

- The victim is then swapped out of the memory, page table entry for the victim is deleted and the contents of the page are copied to a file named '<pid>_<VA[20:0].swap' which is stored in the secondary storage so that we can fetch the memory back into the physical memory whenever required.

Task 3

- Whenever a page is requested which isn't in the page table of the process a call is made to the swap in mechanism.

- Swap in mechanism maintains a queue of all the requests and changes the process state to SLEEPING till the page is brought into memory.

- If there is already a free page in memory, swap_in simply loads the contents of file '<pid>_<VA[20:0].swap' into the free page in physical memory and creates the page table entry.

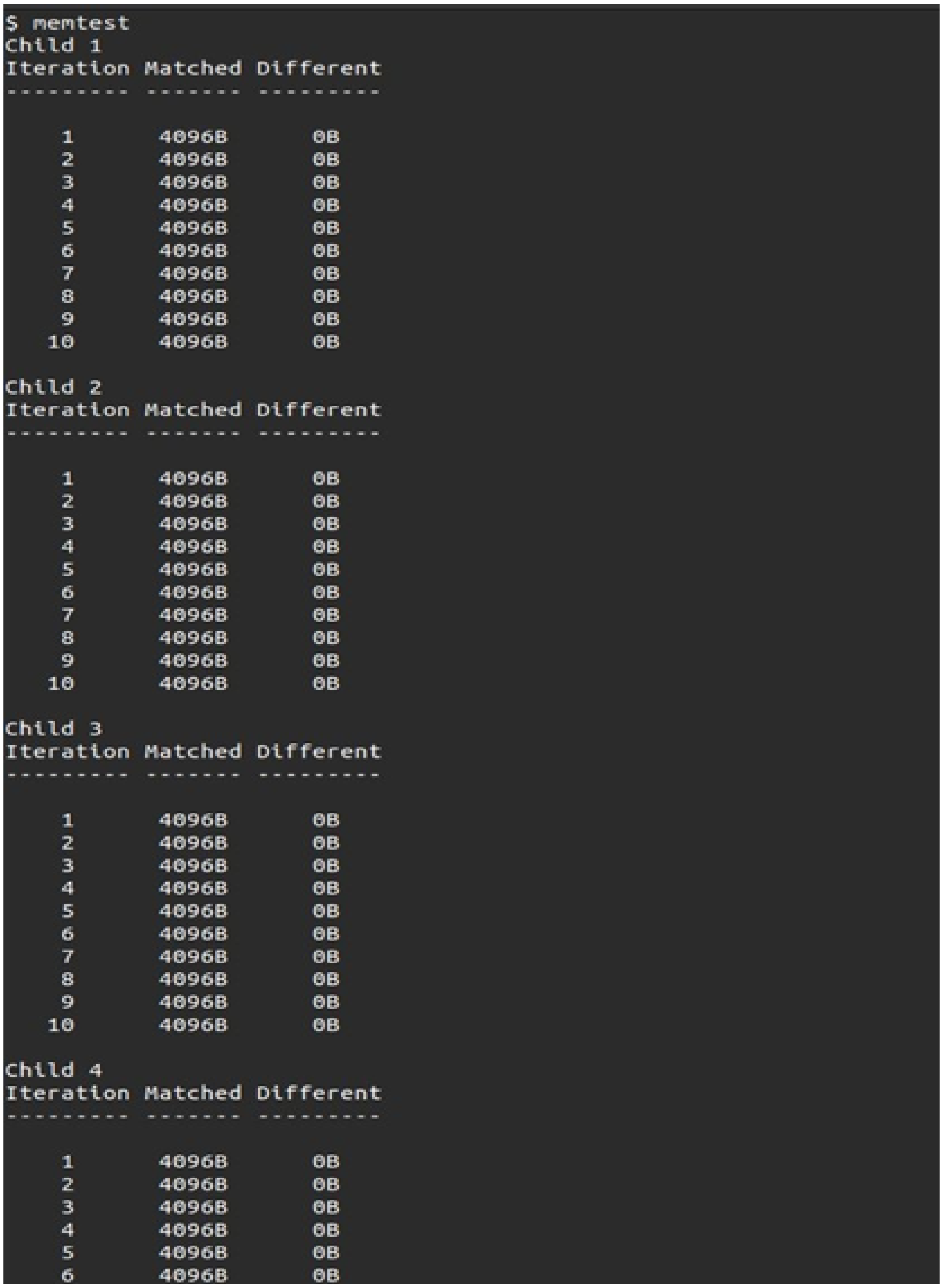
● If there is no free page, then a call to swap_out is made and as soon as a slot becomes free in memory, the required page is loaded in memory and page table entry is updated and the process is put back into RUNNABLE state.

Task4

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.

● 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, 4O96B (4KB) of memory is being allocated using malloc()
● The value stored at index i of the array is given by the mathematical expression 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 at every index with the value returned by the function for that index.

● On running memtest, we obtain the following output:



-
  ```
$ 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
```