OS lab assignment 3

Aadi Aarya Chandra – 200101002

Prakhar Pandey – 200101082

Pratham Pekamwar – 200101087

Part A

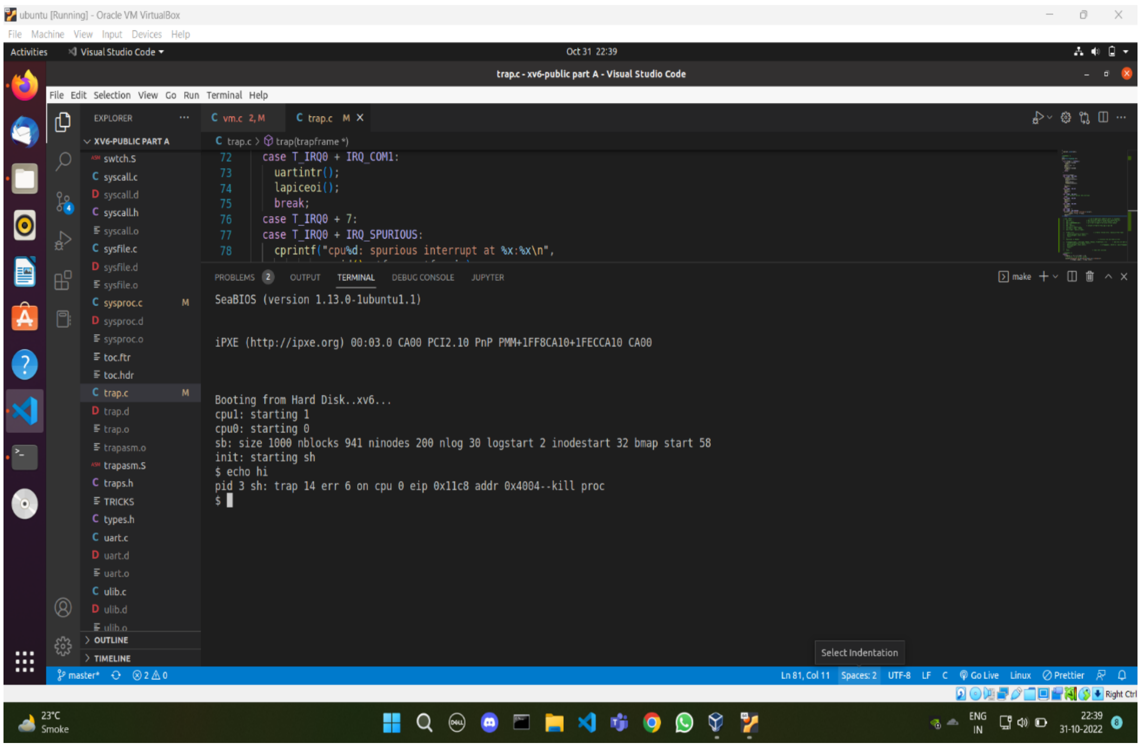
Files modified –

1. Sysproc.c (already modified in the given patch): It modifies the sys_sbrk() function so that growproc(), which allocates phy mem for a proc is not called when sbrk is called (ie when some function calls malloc and asks for memory), but we update the size of process in its struct to make it feel like it has more memory. This would cause a page fault when it tries to access the newly "allocated" memory.
2. Trap.c – here we update the switch case which handles traps by including the case of page faults, and we handle it by finding the virtual address which caused trap fault (rcr2()) then allocate 1 frame corresponding to that address's page. And we don't kill the process like before, so after this allocation (which is done using the same code that allocuvm uses, but with mappages1() used instead of mappages() as mappages was static int in vm.c)
3. vm.c – created a function mappages1(), which is a copy of mappages() except that it returns an int instead of static int.

Testing –

No test file is made as testing for this part is very simple, as it is given that we don't need to do rigorous testing. We just try typing commands like echo hi or ls in the qemu shell, it should run without any trap errors.

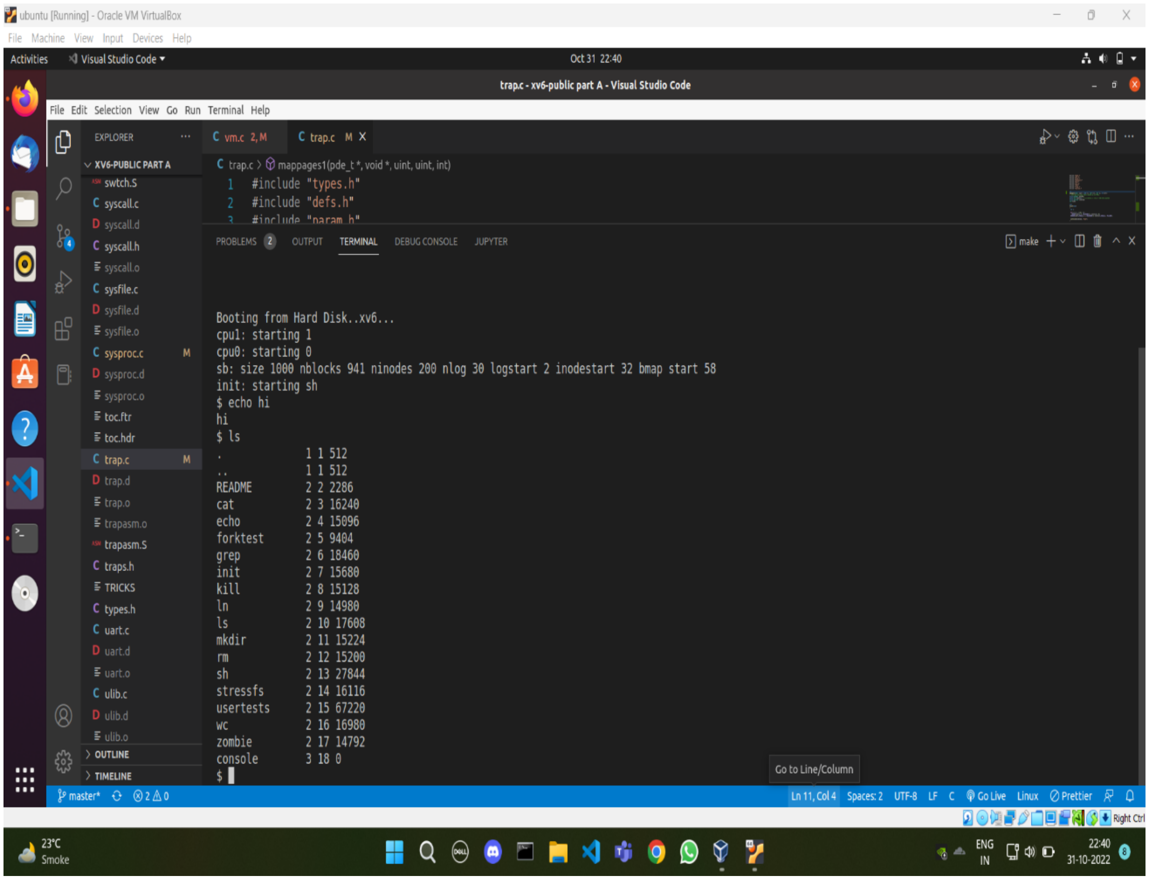
Running the emulator without making changes to trap.c and vm.c (after applying patch) causes trap error of page fault (trap no. 14) as shown below –



```c
    case T_IRQ0 + IRQ_COM1:
      uartintr();
      lapiceoi();
      break;
    case T_IRQ0 + 7:
    case T_IRQ0 + IRQ_SPURIOUS:
      cprintf("cpu%d: spurious interrupt at %x:%x\n",
```

```
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 0 eip 0x11c8 addr 0x4004--kill proc
$ 
```

But after making the required changes, the commands like ls and echo run well -



```c
#include "types.h"
#include "defs.h"
#include "param.h"
```

```
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
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16240
echo           2 4 15096
forktest       2 5 9404
grep           2 6 18460
init           2 7 15680
kill           2 8 15128
ln             2 9 14980
ls             2 10 17608
mkdir          2 11 15224
rm             2 12 15200
sh             2 13 27844
stressfs       2 14 16116
usertests      2 15 67220
wc             2 16 16980
zombie         2 17 14792
console        3 18 0
$ 
```

Part B

Answers to questions asked in problem statement –

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

Xv6 maintains a linked list called kmem of free frames in kalloc.c

➔ What data structures are used to answer this question?

A linked list is used whose elements are struct run*

➔ Where do these reside?

In kalloc.c, the linked list is called kmem.

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

No, but in param.h there is a parameter called NPROC, which sets the size of ptable and hence the maximum number of total processes.

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

1, since there cannot be 0 processes after booting, since all user interactions need to be done using user processes which are forked from initproc.sh

Files modified/added

1. Proc.c –
   a. create_kernel_process function was created in proc.c, works like fork but trapframe is not initialised as these are kernel processes, and eip is set to entrypoint.
   b. Process request queues are created for swap out and swap in processes, with functionalities like spinlock, push and pop operations.
   c. Various functions are created like proc_read, proc_write etc. which write the swapped out process to a file when it is swapped out, and read it back when it is swapped in.
   d. The swap out and swap in processes are also implemented here.
   e. The scheduler is modified slightly to detect whether a page was swapped out in last iteration of the process.
2. Proc.h – virtual address of page fault cause is added as a property in struct proc itself for convenience.
3. Defs.h – prototypes for various functions and data structures implemented in proc.c, vm.c, trap.c
4. Vm.c – in the allocuvm function, we let the process go to sleeping mode, and make it sleep on a channel that we declare within vm.c. then we create the swap out kernel process if not already created and push it to its request queue.
5. Kalloc.c – in kfree function, we run a for loop and wakeup all the processes that were in the swap out request queue.

6. Makefile – the test file memtest.c in written in UPROGS and EXTRAS and random.c is written in EXTRAS for qemu to recognise these files as user programs.
7. Memtest.c – test file, explained below
8. Random.c – used to calculate a random number in a range, as this functionality is not available inside qemu. This is mostly standard code of implementation of this function.
9. User.h – the randomrange function, implemented in random.c, is added to user.h as a protoype so we can use it in memtest.c
10. Mmu.h – a new page table control flag called PTE_A is defined to denote when a page has been accessed, used in swap in and swap out processes. This flag helps us to use LRU policy to find the page to be swapped out, by making suitable changes to the scheduler in proc.c
11. Trap.c – whenever a page fault occurs, we check if it is due to swapping out of that page, if so we call the swap in process.
12. exec.c is also modified very slightly (including some header files) done during debugging.
13. Param.h is also changed slightly as explained below, but only for testing purposes, so it is not included in the submission.

Explanation of swapping out process

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 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. Whenever the process is being context switched into by the scheduler, all accessed bits are unset. Since we are doing this, the accessed bit seen by swap_out_process_function will indicate whether the entry was accessed in the last iteration of the process, thus giving us the LRU page. This code resides in the scheduler and it basically unsets every accessed bit in the process' page table and its secondary page tables. This page is then swapped out and stored to drive.
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.
The termination of swap out process is handled similarly to the kill function in proc.c

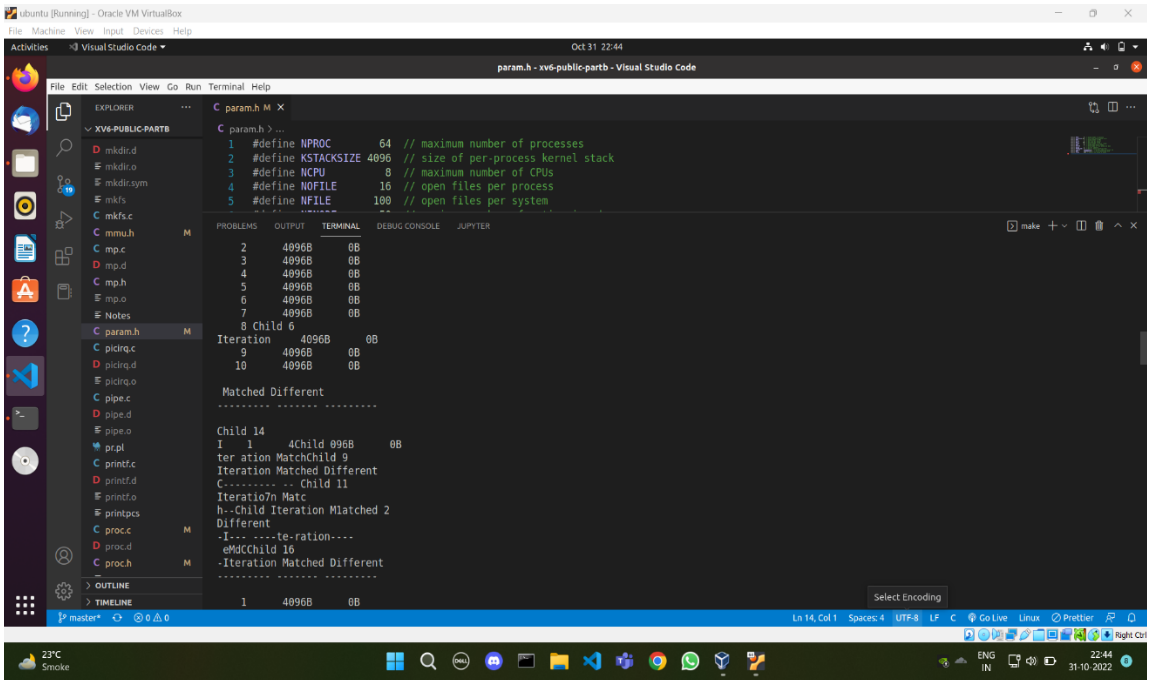
Explanation of swap in process

It is mostly similar to the swap out process in terms of data structures and functions used. 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. 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 m.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.

Testing and sanity checks

Note: we make number of processors (NCPU in param.h) as 1 for testing purposes. This is because keeping NCPU at a higher value results in jumbled output due to context switches.
The main process forks 20 child processes, and each child process runs a loop with 10 iterations. In each iteration, malloc() is used to allocate 4096 bytes (4KB) of memory) to the process, which is then treated like an int array, such that index i has the value i*i – 4i +1.
Then we go thru this array again, matching each value stored in the array with i*i – 4i + 1 and incrementing the no. of bytes matched by 4 each time the integer matches.

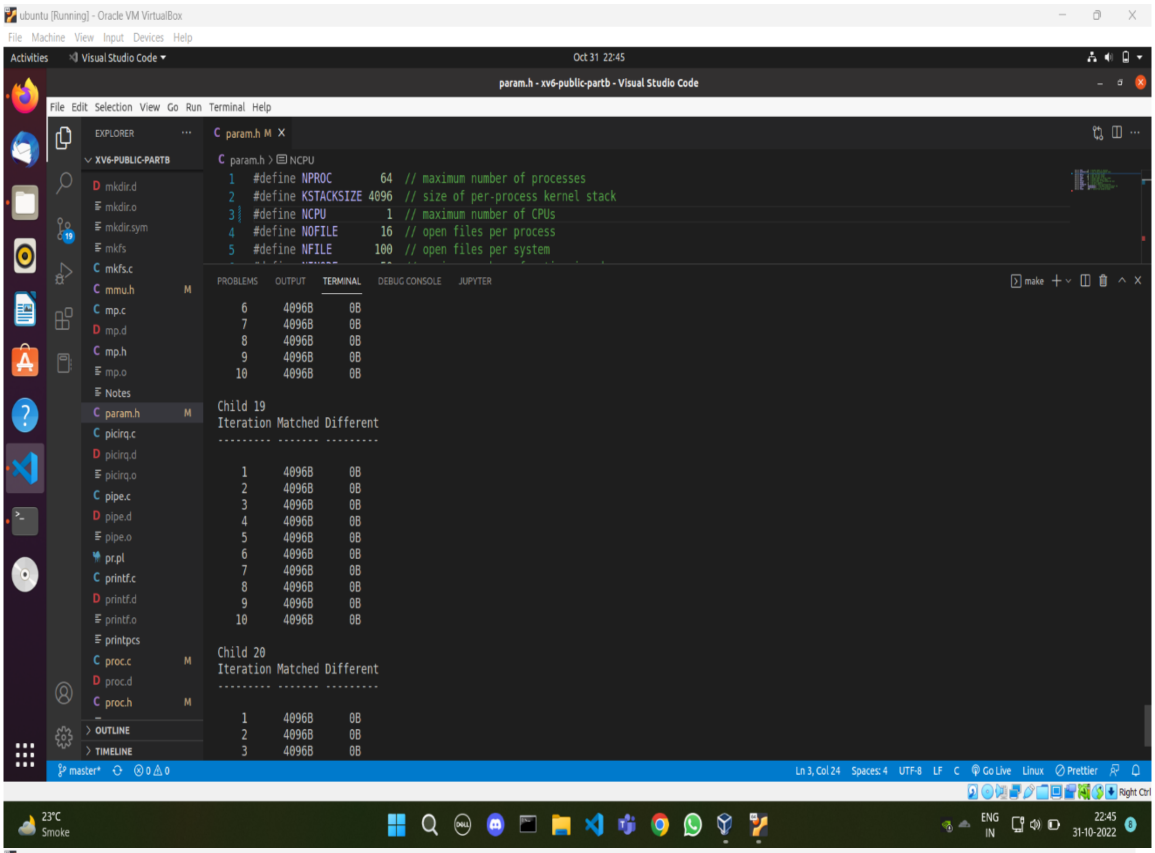
NCPU = 8



```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
```

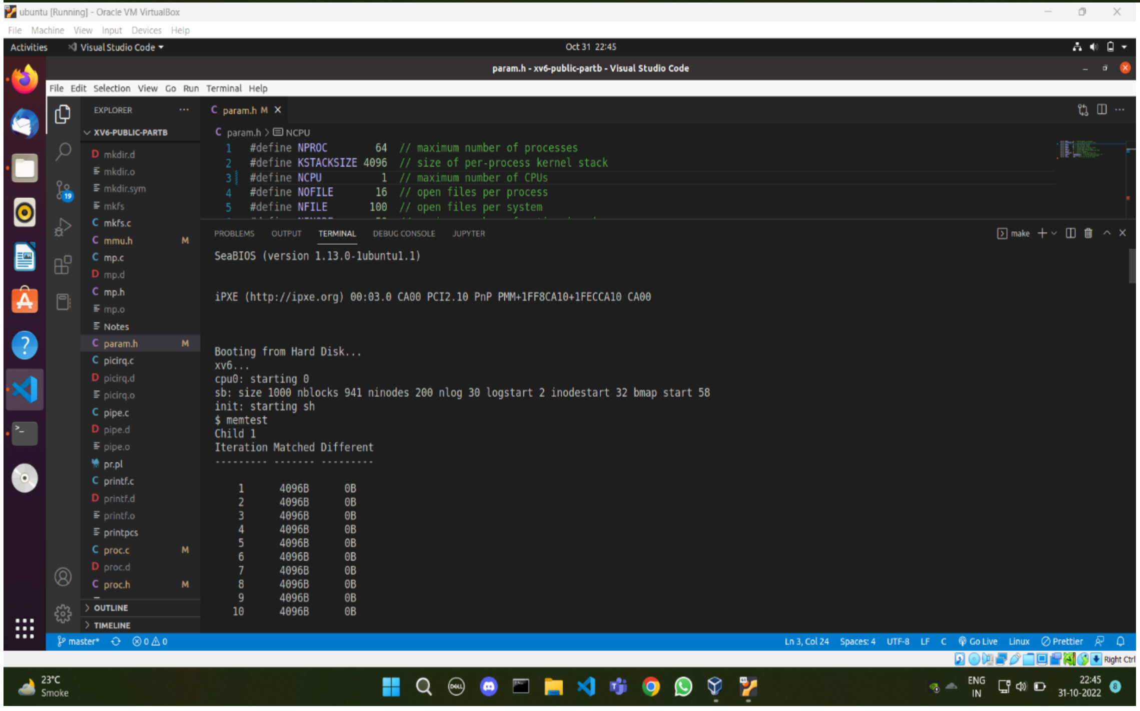
```
    2      4096B      0B
    3      4096B      0B
    4      4096B      0B
    5      4096B      0B
    6      4096B      0B
    7      4096B      0B
    8 Child 6
Iteration     4096B      0B
    9      4096B      0B
   10      4096B      0B

 Matched Different
--------- ------- ---------

Child 14
I    1      4Child 096B      0B
ter ation MatchChild 9
Iteration Matched Different
C--------- -- Child 11
Iteratio7n Matc
h--Child Iteration M1atched 2
Different
-I--- ----te-ration----
 eMdCChild 16
-Iteration Matched Different
--------- ------- ---------

    1      4096B      0B
```

NCPU = 1



```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          1  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
```

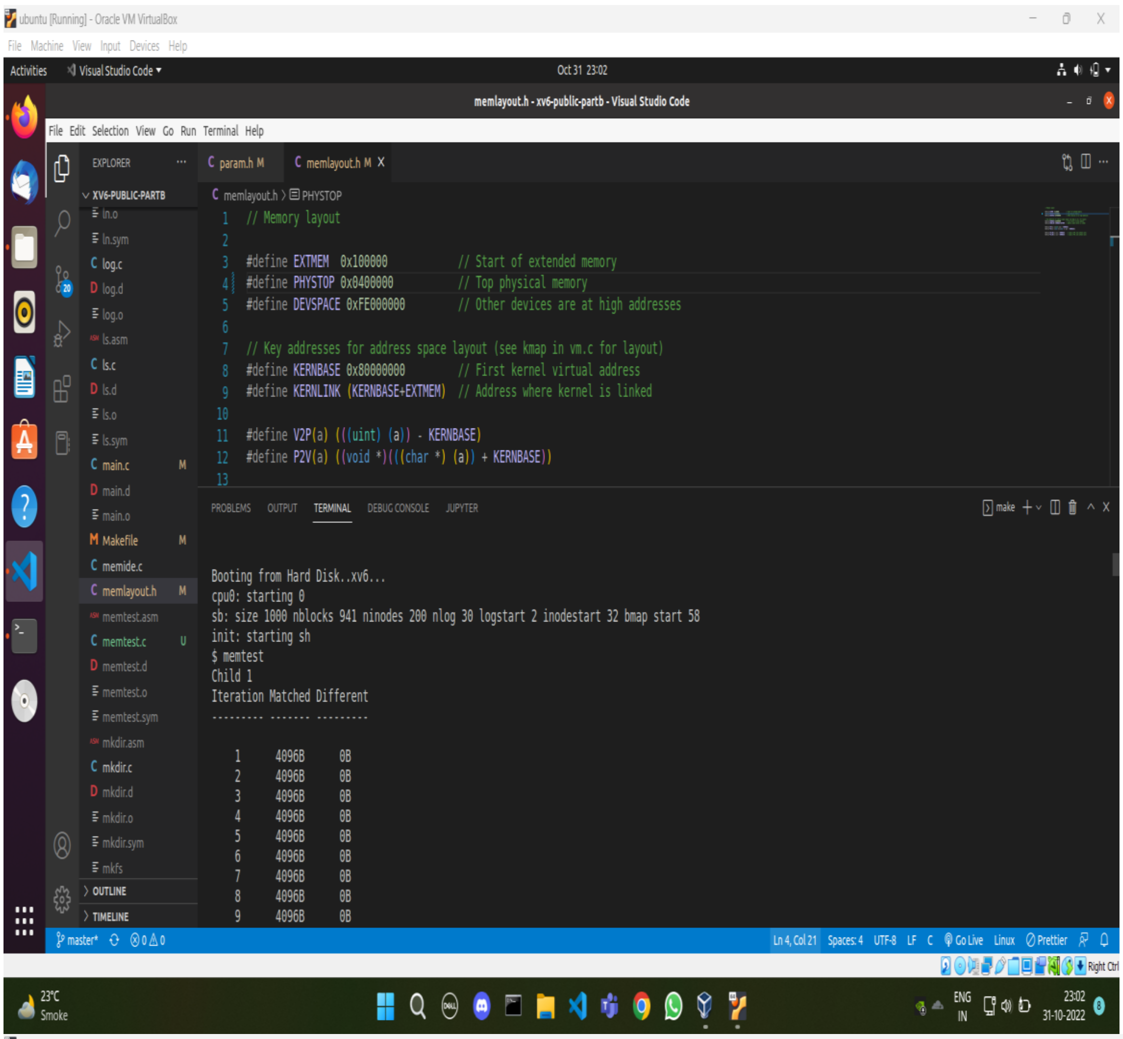
```
    6      4096B      0B
    7      4096B      0B
    8      4096B      0B
    9      4096B      0B
   10      4096B      0B

Child 19
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 20
Iteration Matched Different
--------- ------- ---------

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



```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          1  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
```

```
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...
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
```

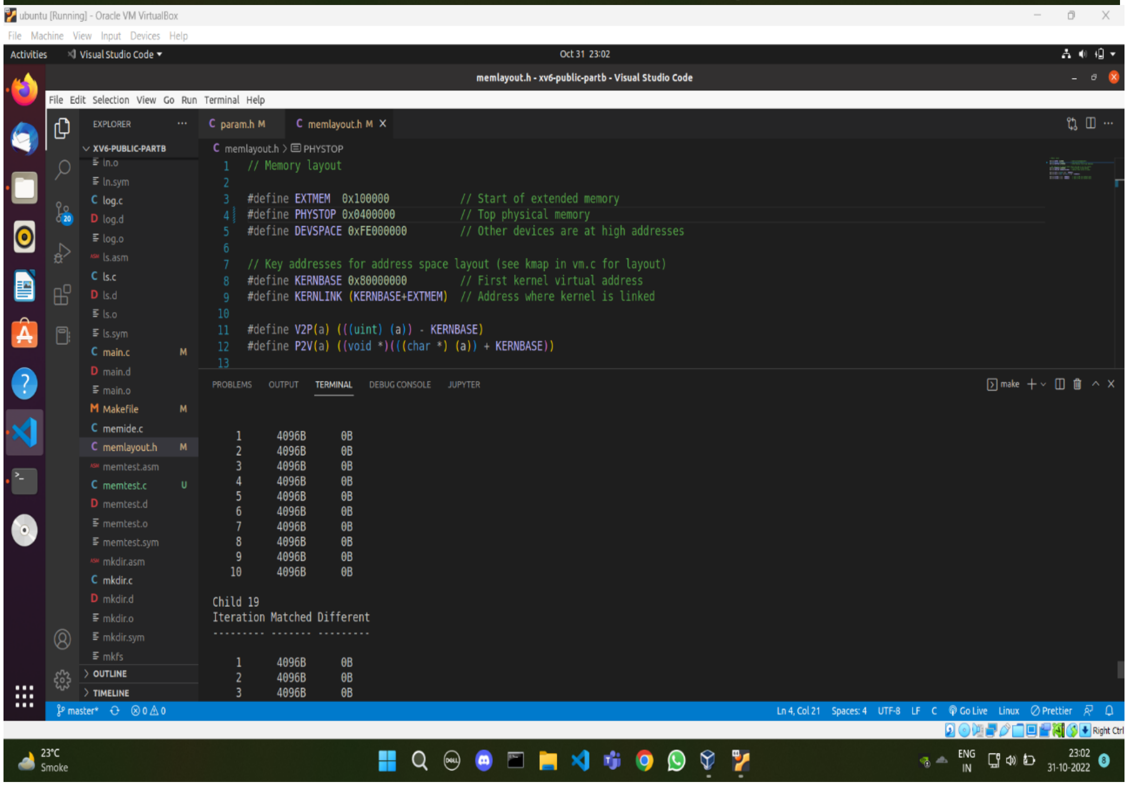
As we can see, the output is correct, ie all integer values match, irrespective of value of NCPU. We also try changing the value of PHYSTOP from its default value of 0xE000000 to 0x0400000. The obtained output is the same as before as shown below. This means that our implementation is correct.



memlayout.h - xv6-public-partb - Visual Studio Code

```c
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0x0400000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

```
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
```



memlayout.h - xv6-public-partb - Visual Studio Code

```c
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0x0400000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

```
    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 19
Iteration Matched Different
--------- ------- ---------

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

Thank you.