# COMP8051 – Operating System Engineering - Assignment/lab 2

**Completion Date: 3rd April 2023**

**Value: 15 marks**

**On completion please zip up your files and upload to Canvas.**

## Question 1 – ps command in xv6

**Given the code in ps-code.txt on Canvas create a ps command in xv6.**

The process table and struct proc in proc.h are used to maintain information on the current processes that are running in the XV6 kernel. Since ps is a user space program, it cannot access the process table in the kernel. So we'll add a new system call. The ps command should print:

- process id
- parent process id
- state
- size
- name

The system call you need to add to xv6 has the following interface:

int getprocs(int max, struct uproc table[]);

struct uproc is defined as (add it to a new file uproc.h):

```
struct uproc {
int pid;
int ppid;
int state;
uint sz;
char name[16];
};
```

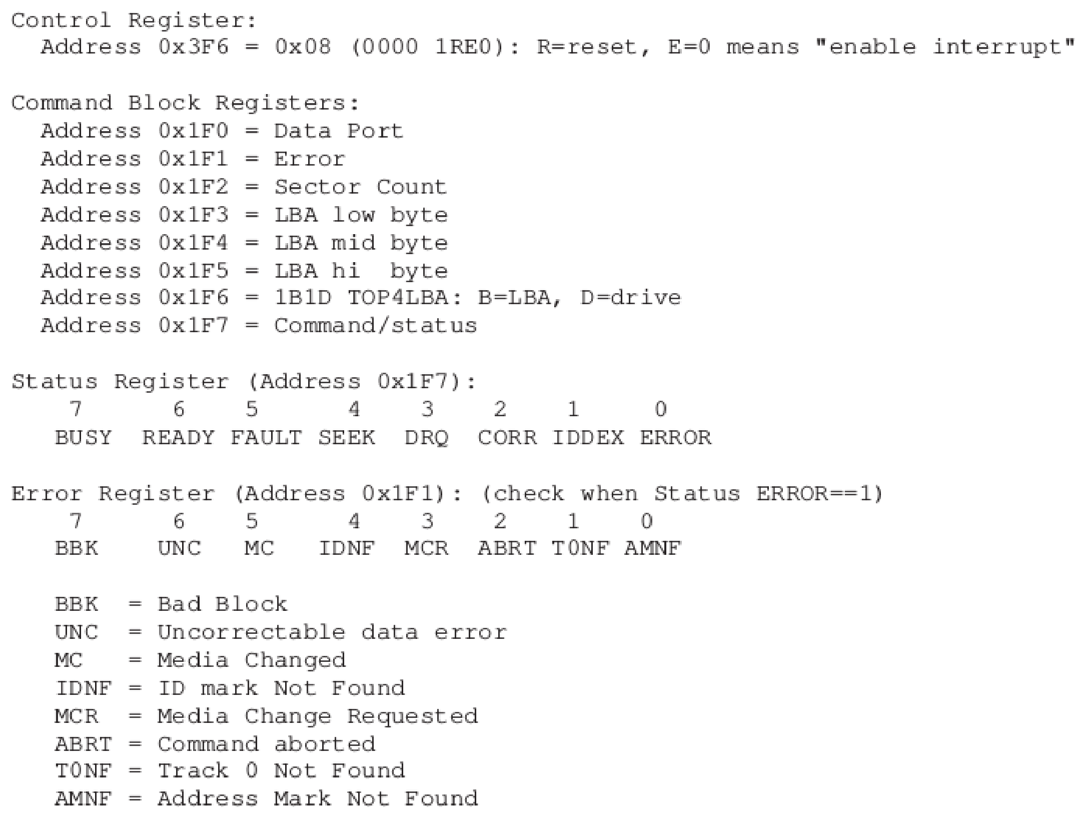
Your ps program calls getprocs with an array of struct proc objects and sets max to the size of that array (measured in struct uproc objects). Your kernel code copies up to max entries into your array, starting at the first slot of the array and filling it consecutively. The kernel returns the actual number of processes in existence at that point in time, or -1 if there was an error.

***(2 marks)***

## **Question 2** IDE Disk Driver

### **Explain how the xv6 operating system uses interrupts to schedule I/O requests to the disk?**

Read the sections on Drivers, Code: Drivers in chapter 3 and section 36.8 of file-devices.pdf, which gives a summary of the IDE disk controller protocol. See in particular the code in **ide.c**. See image below



```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5      4     3     2     1      0
   BUSY  READY FAULT SEEK  DRQ   CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6     5      4     3     2     1     0
   BBK    UNC   MC    IDNF  MCR   ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

The xv6 source code includes a working IDE driver in ide.c. For example the piece of code in *idestart*

*outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b→sector>>24)&0x0f));*

is associated with:

I/O Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

The 0xe is to set 1B1 in bits 5,6,7 B=1 indicates that we are going to have the low 4 bits at address 0x1f6 set the top 4 bits of the Logical Block Address (LBA).

((b→dev&1)<<4) sets the D bit.

((b→sector>>24)&0x0f)) sets the top 4 bits of LBA to low four bits at address 0x1f6.

An IDE disk presents a simple interface to the Disk system, consisting of four types of register: control, command block, status, and error. These registers are available by reading or writing to specific “I/O addresses” (such as 0x3F6 ) using (on x86) the in and out I/O instructions.

***(4 marks)***

## Question 3: Big Files in xv6

Currently xv6 files are limited to 140 sectors, or 71,680 bytes. This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 128 more block numbers, for a total of 12+128=140. You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 128 addresses of singly-indirect blocks, each of which can contain up to 128 addresses of data blocks. The result will be that a file will be able to consist of up to 16523 sectors (or about 8.5 megabytes).

See big-notes.txt for outline code.

See   Appendix 1 for more details.

***(4 marks)***

## Question 4: FUSE file system

The  fuse-mem-tree-lab.zip contains a simple in memory file system that will integrate into Linux using FUSE.

FUSE (Filesystem in Userspace) is an interface for userspace programs to export a filesystem to the Linux kernel.

sudo apt-get install libfuse-dev
download and extract the fuse-mem-tree-lab.zip
run **make** to build the executable
./memfs -f tes1

You should see the tes1 directory in the file explorer with data files and directories in it.  Note:- all the files have dummy contents in them.

You are to:

**a)** modify the do_read function in the mem_filesystem.c so that it reads the actual contents of the requested file. See the TODO comments in the code.

**b)** Give a brief overview of FUSE and a distributed file system such as glusterfs or ceph which is built on it.

See here:-

https://www.kernel.org/doc/html/next/filesystems/fuse.html

https://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/

https://docs.gluster.org/en/main/Quick-Start-Guide/Architecture/

https://www.gluster.org/

https://youtu.be/7I9uxoEhUdY

***(5 marks)***

## Appendix 1 – Big Files xv6

Modify your Makefile's `CPUS` definition so that it reads:

CPUS := 1

Add

QEMUEXTRA = -snapshot

right before `QEMUOPTS`

The above two steps speed up qemu tremendously when xv6 creates large files.

`mkfs` initializes the file system to have fewer than 1000 free data blocks, too few to show off the changes you'll make. Modify `param.h` to set `FSSIZE` to:

```
#define FSSIZE       20000  // size of file system in blocks
```

Download big.c into your xv6 directory, add it to the UPROGS list, start up xv6, and run `big`. It creates as big a file as xv6 will let it, and reports the resulting size. It should say 140 sectors.

### What to Look At

The format of an on-disk inode is defined by `struct dinode` in `fs.h`. You're particularly interested in `NDIRECT`, `NINDIRECT`, `MAXFILE`, and the `addrs[]` element of `struct dinode`. Look here for a diagram of the standard xv6 inode.

The code that finds a file's data on disk is in `bmap()` in `fs.c`. Have a look at it and make sure you understand what it's doing. `bmap()` is called both when reading and writing a file. When writing, `bmap()` allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

`bmap()` deals with two kinds of block numbers. The `bn` argument is a "logical block" -- a block number relative to the start of the file. The block numbers in `ip->addrs[]`, and the argument to `bread()`, are disk block numbers. You can view `bmap()` as mapping a file's logical block numbers into disk block numbers.

### Your Job

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block.

You don't have to modify xv6 to handle deletion of files with doubly-indirect blocks.

If all goes well, `big` will now report that it can write 16523 sectors. It will take `big` a few dozen seconds to finish.

## Hints

Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 16,384 blocks (really 16383, since you have to decrease the number of direct blocks by one).

Think about how you'll index the doubly-indirect block, and the indirect blocks it points to, with the logical block number.

If you change the definition of `NDIRECT`, you'll probably have to change the size of `addrs[]` in `struct inode` in `file.h`. Make sure that `struct inode` and `struct dinode` have the same number of elements in their `addrs[]` arrays.

If you change the definition of `NDIRECT`, make sure to create a new `fs.img`, since `mkfs` uses `NDIRECT` too to build the initial file systems. If you delete `fs.img`, `make` on Unix (not xv6) will build a new one for you.

If your file system gets into a bad state, perhaps by crashing, delete `fs.img` (do this from Unix, not xv6). `make` will build a new clean file system image for you.

Don't forget to `brelse()` each block that you `bread()`.

You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.

***(5 marks)***

**sudo apt-get install libfuse-dev**