NSWI120 exam 8.2.2022 Pavlo Shvaiko

Tuesday, 8 February 2022 2:24 PM

Pavlo Shvaiko

① Information about the number of Inodes in the filesystem is located in the Super Block.
Super Block is on offset of 1 block (1024 bytes)
$1024_{dec} \rightarrow ?_{hex}$
$1024 = 256 \times 4 \qquad 256 = 16^2$
$\therefore 1024_{dec} \rightarrow 400_{hex}$
So this information is located on offset 400_{hex} in hexdump.
According to specification it is the first 2 bytes.

400: | 00 | 01 |
|---|---|
| c0 | 02 |

little endian

a - 10
b - 11
c - 12

$02c0_{hex}$: $0 \times 16^3 + 2 \times 16^2 + 12 \times 16^1 + 0 \times 16^0$
$= 2 \times 256 + 12 \times 16$
$= 512 + 192$
$= 704$ inodes

16 × 12 = 32 + 160 = 192

② first.txt file is stored on offset 6860_{hex}, seen in hex dump.
First 2 bytes of the file contain the inode number that contains information about this file
6860: 03 | 00 little endian
$\therefore 0003_{hex} = 3_{dec}$
So info about this file is stored in inode number 3.

Filesystem:	size (blocks)	(counting starts from 0)
Blank blocks:	1	
Super block:	1	
Inode map:	1	
Zone map:	1	
Inode table:		

Super block:

Super block:

400 : c0 02 00 08 01 00 01 00

size of Imap (blocks) | size of Zmap (blocks)

Inode offset : 4 blocks = $1024 \times 4 = 4096_{dec}$ bytes

$16 \times 16 \times 16 = 16^3$

offset : 1000_{hex}

Each inode is 32 bytes

So 3rd inode is offset : $1000_{hex} + 60_{hex} = 1060_{hex}$

Size is offset 4 into the inode entry: 0e 00 00 00 (little endian)

$\therefore$ size = $00\,00\,00\,0e_{hex} = 14_{dec}$

size = 14 bytes

Contents: zone pointers are offset 15_{dec} into the inode entry

There is only 1 zone pointer : 1c 00 (little endian)

zone pointer = 001c = 1x16 + 12

= 28th block

$28_{dec} = 1c_{hex}$

$4 \times C = 4 \times 12$

$= 48$

$= 16 \times 3$

$1024 \times 28 = 28672$

$1024 \times 1c = 7000_{hex}$

offset 28672_{dec} bytes

offset 7000_{hex}

Contents of first.txt is "I'll be back!"

(5)

a		b		c	
LsB	MsB	LsB	MsB	LsB	MsB
0xE000	0xE001	0xE002	0xE003	0xE004	0xE005

a = ((b & 0x1E09) | c) << 1

1) b & 0x1E09

```
LDA 0xE002
AND #0x09
TAX
```
LsB of b
store result in register X

```
LDA 0xE003
AND #0x1E
TAY
```
Msb of b
store result in register Y

2) a = ... | c

```
TXA
ORA 0xE004
STA 0xE000
```
LsB or with c
store result in LsB of a

```
TYA
ORA 0xE005
STA 0xE001
```
MsB or with c
store result in MsB of a

3) a = a << 1

```
LDA 0x E000
CLC
ROL A
STA 0xE000
```
LsB of a shift (rol) left.
clear carry so 0 go do the Lsb.
Store result in LsB of a

```
LDA 0x E001
ROL A
STA 0xE001
```
MsB of a shift (rol) left
no clear carry so the bit
rolled out during the previous
operation goes do the Lsb.
Store result in MsB of a.

(6) a) x = 65533

$x = 2^{16} - 3 = (2^{16} - 1) - 2$

⑥ a) $x = 65533$

$x = 2^{16} - 3 = (2^{16} - 1) - 2$

Python added → 0 1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 0 1

15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0

$x = \sim x$

1 0000 0000 0000 0010

print(x, format(x, "04x"))

↳ negative number

→ we need to 'convert' to positive using two's complement

NOT(1 0000 0000 0000 0010) + 1

= 0 1111 1111 1111 1101 + 1

= 0 1111 1111 1111 1110 $8+4+2=14$

= - FFFE

b) $x = 65533$

$x = 2^{16} - 3 = (2^{16} - 1) - 2$

Python added → 0 1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 0 1

15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0

$x = \sim x$

1 0000 0000 0000 0010

x = uint16(x)

↳ truncates x to 16 bits

0000 0000 0000 0010

print(x, format(x, "04x"))

0002 (in hex)

⑦ Subtraction with borrow:

Subtraction with borrow is implemented like subtraction taught in primary schools.

Whenever, subtraction of 2 digits of 2 numbers will result in a negative answer, 1 is borrowed

Whenever, subtraction of 2 digits of 2 numbers will result in a negative answer, 1 is borrowed from the digit with a higher power.
In CPU this is implemented by having a borrow flag which is set (=1) when borrow is needed and cleared (=0) when borrow is not needed.
SBB = A − B − borrow flag

Subtraction with carry:
Similar to subtraction with borrow but uses the carry flag in reverse logic.
carry flag = 1 means borrow not needed
carry flag = 0 means borrow is needed

SBC = A − B − borrow with 8-bit CPU
= A − B − not(carry)
= A − B − (1−C) + 256 ← result does not change
= A − B − 1 + C + 256
= A − B + C + 255
= A + (255 − B) + C
= A + C + NOT(B)

NOT(B) because

| 1 1 1 1 1 1 1 1 |
| − |
| B |
| 1 whenever $B^i = 0$ |
| 0 whenever $B^i = 1$ |

Advantage: addition with carry instruction can be reused.

(8) "BAG"
A - 41h
B - 42h
C - 43h
D - 44h
E - 45h

Lsb-first

F - 46h

G - 47h

B = 0100 0010

A = 0100 0001

G = 0100 0111

Transmission: 1 (start) 01000010 0(stop)

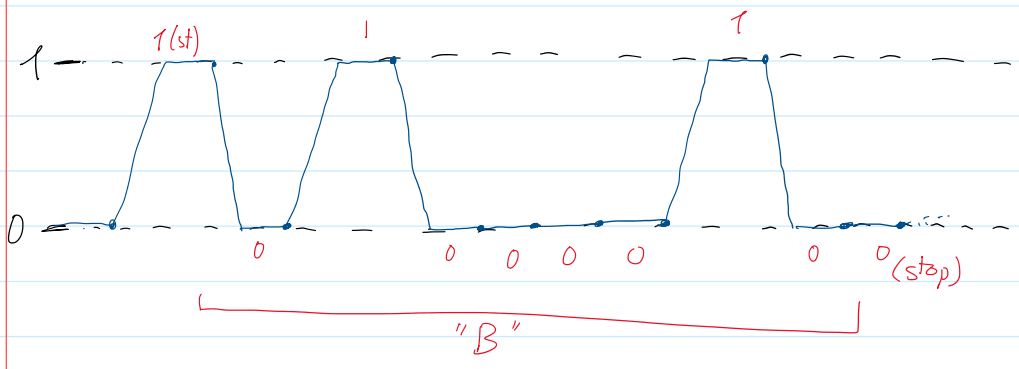
"B" (Lsb-first)

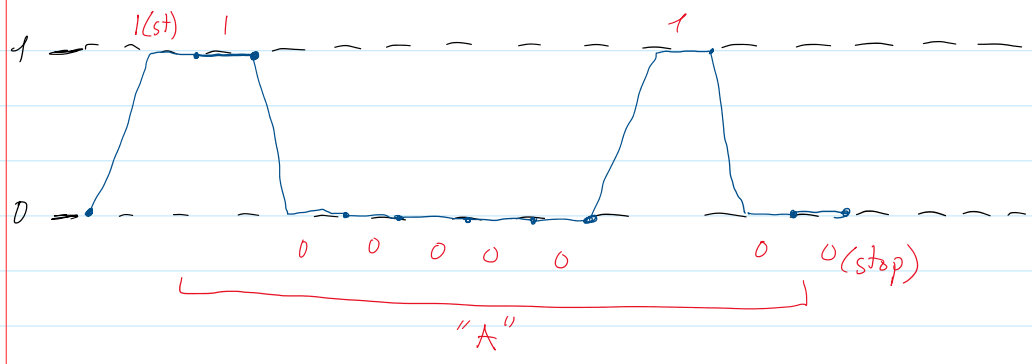
1(start) 10000010 0(stop)

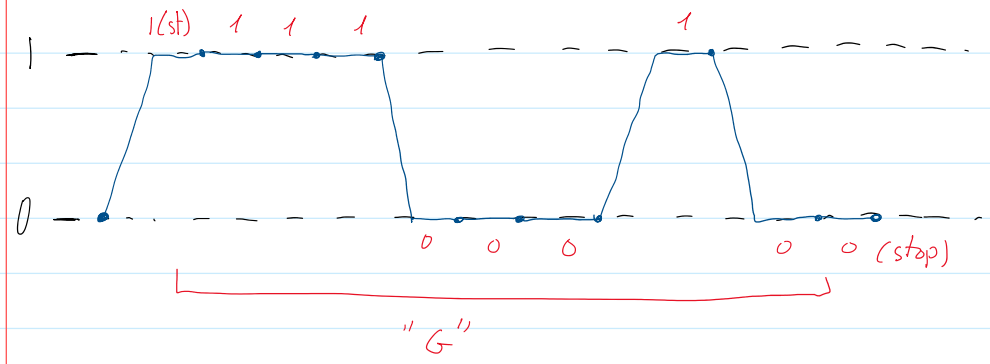
"A" (Lsb-first)

1(start) 11100010 0(stop)

"G" (Lsb-first)







(9) Msb-first bytes are sent

perform humidity measurement

$S_{RH} = 3048_{dec}$

Device address: '1000 000" + R: "1" or W: "0"

Trigger RH measurement, no hold master: '1111 0101'

Start | 0x80 | ACK | 0xF5 | Stop

↓ device address + write

↓ command to trigger RH measurment

$3048_{dec} \rightarrow ?_{hex}$

3048 = 2048 + 1000

= 2048 + 1024 − 24

$= 2^{11} + (2^{10} - 1) - 23$

23 = 16 + 7

Start | 0x81 | ACK | 0x0B | ACK | 0xE8 | NACK | Stop

↓ device address + read

↓ data MSB

↓ data LSB

0 11 (b) 14 (E) 8

0 0 0 0 | 1 0 1 1 | 1 1 1 0 | 1 0 0 0

15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0

So $3048_{dec} = 0x0BE8$

* sent MSB first (from specification example on page 8)

(10) 65.625_{dec}

0 1 0 0 0 0 0 1 . 1 0 1 0 _ _ _ _

7 6 5 4 3 2 1 0 . -1 -2 -3 -4

$2^{-1} = 0.5$

$2^{-2} = 0.25$

$65 = 2^6 + 1$

$.625 = 2^{-1} + 2^{-3}$

→ little endian

$2^{-1} = 0.5$ $65 = 2^6 + 1$

$2^{-2} = 0.25$ $.625 = 2^{-1} + 2^{-3}$

$2^{-3} = 0.125$

Little endian

8.24: 0x 41. A0 00 00

12.20: 0x 04 1A 00 00

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
$00AC1400:	00	00	00	00	A0	41	00	00	00	00	00	00	1A	04	00	00

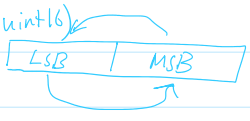
8.24 little-endian

12.20 little-endian

③

```
def PrintBlocks(name: str, inodeOffset: int):
    f = open(name, "rb")  # open file in binary read format
    # seek to Zone 0 of that inode
    f.seek(inodeOffset + 15)
    # Read and print direct zone pointers
    for _ in range(7):
        buffer = f.read(2)  # read 2 bytes into buffer
        a = frombuffer(buffer[0], dtype = uint16) # store LsB
        b = frombuffer(buffer[1], dtype = uint16) # store MsB
        b = b << 8  # SHL by 8 bits to move it to MsB
        c = a | b  # combine MsB and LsB
        if c != 0x00:
            print(c)
        # Another approach with ROL
        # a = frombuffer(buffer, dtype=uint16)
        # a = a ROL 8

    # Read zone 7
    buffer = f.read(2) # read 2 bytes
    a = frombuffer(buffer[0], dtype = uint16) # store LsB
    b = frombuffer(buffer[1], dtype = uint16) # store MsB
```



```
a = frombuffer(buffer[0], dtype = uint16) # store LsB
b = frombuffer(buffer[1], dtype = uint16) # store MsB
b = b << 8   # SHL by 8 bits to move it to MsB
block_num = a | b   # combine MsB and LsB
f.close()  # close opened file
f = open(name, "rb")  # open it again to start seeking
                                          from 0

f.seek(block_num x 1024): # seek to indirect block
for _ in range(512):
    buffer = f.read(2) # read 2 bytes
    a = frombuffer(buffer[0], dtype = uint16) # store LsB
    b = frombuffer(buffer[1], dtype = uint16) # store MsB
    b = b << 8   # SHL by 8 bits to move it to MsB
    c = a | b
    if  c != 0x00:
        print(c)


# I did not know if I could return when I encountered
  the first block pointer which is 0x00
  so this code checks all block pointers


(4) def GetLongestFreeBlock(bitmap: bytes):
    longest_free = 0  # stores longest free sequence
    longest_free_block = None  # pointer to the start of longest free block
    counter = 0     temp_pointer = None # points to current free block start
    for i in range(1024):
        b = bitmap[i]  # save 1 byte from bitmap
        mask = 0x01   # 0000 0001
        for j in range(8):
            if  b & mask == 0x00:  # free
                counter ++
                if  temp_pointer  == None:
                    temp-pointer  = (i x 8) + j
                mask = mask << 1  #eg. 0000 0001 --> 0000 0010
            else:  # b & mask == 1 --> not free
```

```
        if  counter > longest_free:
                longest_free_block = temp_pointer
                longest_free = counter
        temp_pointer = None
        counter = 0

return  longest_free_block
```