Lab 5

Ai Nguyen

December 12, 2015

Lab to explore RAID storage. Using this technique, it is possible to recover from a failed disk drive by stripping data across 3 drives and storing the xor of those 3 values into the 4th (parity) drive

1 C algorithm

C source code written to file lab6.c

```
#include <stdio.h>
#include <math.h>
#define N 48 //Symbolic constant vs Creates memory location
char dataIn[] = "This is some test data for a RAID simulator";
char dataOut[N]; //Partial initialization of array
char disk1[N/3]; // expect to store "Tsso sdaoaA mar"
char disk2[N/3];
char disk3[N/3];
char disk4[N/3];
int main()
{
  printf("dataIn<%s>\n", dataIn);
```

Stripe the data across the drives and store parity in disk4

C source code appended to file lab6.c

```
  int i =0, j =0;
  for(i =0, j =0; i < N-2; i+=3, j++)
  {       disk1[j] = dataIn[i];
          disk2[j] = dataIn[i+1];
          disk3[j] = dataIn[i+2];
          disk4[j] = disk1[j] ^ disk2[j] ^ disk3[j];
  }
```

C source code appended to file lab6.c

```
  printf("dataOut<%s>\n", dataOut);
  printf("disk1<%s>\n", disk1);
  printf("disk2<%s>\n", disk2);
  printf("disk3<%s>\n", disk3);
  printf("disk4<%s>\n", disk4);
```

Suppose disk2 is corrupted(like politicians)

C source code appended to file lab6.c

```
for (i=0; i < N/3; i++)
{
  disk2[i] = '.';
}
for (i=0 , j =0; i < N - 2; i+=3, j++)
{
  dataOut[i] = disk1[j];
  dataOut[i+1] = disk2[j];
  dataOut[i+2] = disk3[j];
}
printf("dataOut<%s>\n", dataOut);
```

Fix/rebuild disk2

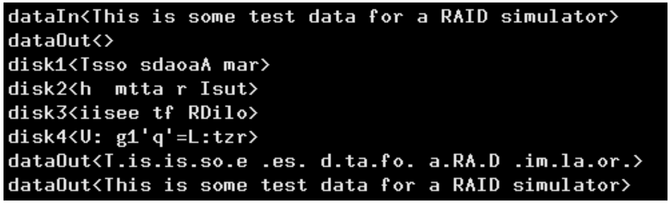
C source code appended to file lab6.c

```
for (i =0; i < N/3; i++)
{
  disk2[i] = disk1[i] ^ disk3[i] ^ disk4[i];
}
```

unstripe data across drives and store msg in dataOut

C source code appended to file lab6.c

```
for(i =0, j=0; j < N/3; i+=3, j++)
{
  dataOut[i] = disk1[j];
  dataOut[i+1] = disk2[j];
  dataOut[i+2] = disk3[j];
}
printf("dataOut<%s>\n",dataOut);
  return 0;
}
```



```
dataIn<This is some test data for a RAID simulator>
dataOut<>
disk1<Tsso sdaoaA mar>
disk2<h  mtta r Isut>
disk3<iisee tf RDilo>
disk4<U: g1'q'=L:tzr>
dataOut<T.is.is.so.e .es. d.ta.fo. a.RA.D .im.la.or.>
dataOut<This is some test data for a RAID simulator>
```

2 ASM Code

2.1 Main

First we create some constants for the size of our (size of string). Use *.equ* to assign names to numbers.

N size of *dataIn* and *dataOut* without counting terminated character.

M size of *msg*

M1 size of *msg*1

M2 size of *msg*2

DISK_SIZE size for disks

asm source code written to file lab6.s

```
.equ N,53
.equ M,20
.equ M1,29
.equ M2,37
.equ DISK_SIZE, 23
```

Data section will include:

dataIn Storing test data for our program

dataOut Storing the result from our program. Since we need *dataOut* the same with *dataIn*, I initialized it the same with *dataIn* to have the same size.

disk1,disk2,disk3,disk4 Array for each disk. I initialized each array with their names first and all spaces following ($numberofchar = DISK\_SIZE$)

msg,msg1,msg2, endline These string variables is just for making the output more readable and clearer asm source code appended to file lab6.s

```
.data
  dataIn: .string "DataIn: This is some test data for a RAID simulator  "
  dataOut: .string "DataOut: This is some test data for a RAID simulator  "
  disk1: .string "Disk1:                    "
  disk2: .string "Disk2:                    "
  disk3: .string "Disk3:                    "
  disk4: .string "Disk4:                    "
  msg: .string "ASM RAID simulator\n"
  msg1: .string "After one disk is corrupted\n"
  msg2: .string "After rebuilding the corrupted disk\n"
  endline: .string "\n"
```

We won't use *printf* from C library to print what we need but the system call technique from ASM

First print out *msg*, *dataIn*, *endline*

asm source code appended to file lab6.s

```
.text
.globl _start
_start:

mov $M, %edx
mov $msg, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

asm source code appended to file lab6.s

```
mov $N, %edx
mov $dataIn, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

3 blocks of code above have the same syntax in the format:

- Copy size of buffer to *%edx*
- Copy address of buffer to *%ecx*
- Copy file descriptor to *%ebx*. In this case, we need 1 for STOUT
- Copy number of system call to *%eax*. We need 4 for system write.

Then we call *load_data* to store data into disks. This function doesn't take any parameter.

asm source code appended to file lab6.s

```
# store data to disks
call load_data
```

After loading data. Print it out with the same syntax we use before for system writing. For readable output, we print *endline* after each disk priting.

asm source code appended to file lab6.s

```
mov $DISK_SIZE, %edx
mov $disk1, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

asm source code appended to file lab6.s

```
mov $DISK_SIZE, %edx
mov $disk2, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

asm source code appended to file lab6.s

```
mov $DISK_SIZE, %edx
mov $disk3, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

asm source code appended to file lab6.s

```
mov $DISK_SIZE, %edx
mov $disk4, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

We will make one of four disks corrupt by calling function *make_corrupt*. This function will take one parameter, which is the address of the disk we want to be corrupted (in the code, we use $*disk*2)

asm source code appended to file lab6.s

```
push $disk2 #disk2 is corrupted
call make_corrupt
add $4, %esp
```

Print dataOut when one disk is corrupted

asm source code appended to file lab6.s

```
mov $M1, %edx
mov $msg1, %ecx # print msg1
mov $1, %ebx
mov $4, %eax
int $0x80
```

asm source code appended to file lab6.s

```
call print_dataOut

# print msg2
mov $M2, %edx
mov $msg2, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

After make one disk corrupted, we rebuild that disk by doing XOR 3 other disks. Calling funtion *rebuild* that take 4 parameter. First 3 disks is to rebuild the corrupted one, and the fourth disk is the one needed to rebuild.

asm source code appended to file lab6.s

```
#Rebuild corrupted disk
push $disk1
push $disk3
push $disk4
push $disk2      # this for fixing
call rebuild
add $12, %esp
```

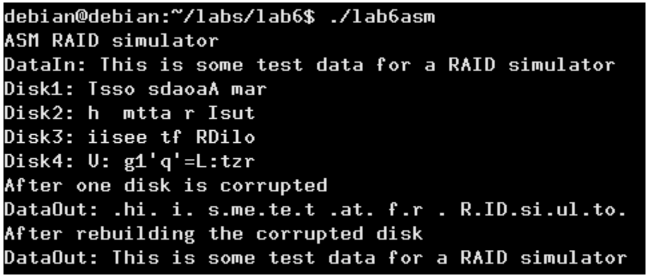
Then print dataOut to test our program

asm source code appended to file lab6.s

```
call print_dataOut
#End
mov $1, %eax
mov $0, %ebx
int $0x80
```

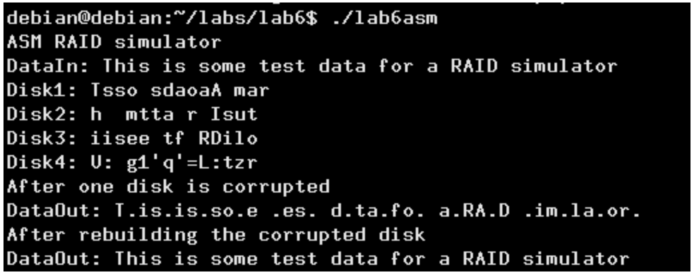
2.2 Testing

when disk 1 is corrupted



```
debian@debian:~/labs/lab6$ ./lab6asm
ASM RAID simulator
DataIn: This is some test data for a RAID simulator
Disk1: Tsso sdaoaA mar
Disk2: h  mtta r Isut
Disk3: iisee tf RDilo
Disk4: U: g1'q'=L:tzr
After one disk is corrupted
DataOut: .hi. i. s.me.te.t .at. f.r . R.ID.si.ul.to.
After rebuilding the corrupted disk
DataOut: This is some test data for a RAID simulator
```

when disk 2 is corrupted



```
debian@debian:~/labs/lab6$ ./lab6asm
ASM RAID simulator
DataIn: This is some test data for a RAID simulator
Disk1: Tsso sdaoaA mar
Disk2: h  mtta r Isut
Disk3: iisee tf RDilo
Disk4: U: g1'q'=L:tzr
After one disk is corrupted
DataOut: T.is.is.so.e .es. d.ta.fo. a.RA.D .im.la.or.
After rebuilding the corrupted disk
DataOut: This is some test data for a RAID simulator
```

2.3 load_data

This function is for loading data into 4 buffers (disks). For each group of 3 characters in *dataIn* (the first group is from the beginning):

disk1 store 1^{st} letter of that group

disk2 store 2^{nd} letter of that group

disk3 store 3^{rd} letter of that group

disk4 doing XOR of $disk1$, $disk2$ and $disk3$

Use $\%esi$ for i (index of $dataIn$) and $\%edi$ for j (index of disks).
Since I initialized 4 buffers (disks) in the format : "Disk1: [16 spaces]", the data read from $dataIn$ will load to each disk beginning at 8^{th} character, or at index $i = 7$. For $dataIn$, the data we need is from index $j = 8$

asm source code appended to file lab6.s

```
load_data:
push %ebp
mov %esp, %ebp
mov $8, %esi            #i = 8
mov $7, %edi            #j = 7
```

Begin of the loop ($\_L1$) The loop will stop when $i < N - 2$. Since $N = 53$, we will compare $\%esi$ with 51. If $\%esi$ is greater than 51, break the loop by jumping to ${}_L2$, the end of function.

asm source code appended to file lab6.s

```
_L1:
cmp $51, %esi           # N-2
jge _L2                 # if i >= 51
```

Body of the loop

asm source code appended to file lab6.s

```
movb dataIn(%esi), %cl
movb %cl, disk1(%edi)
inc %esi
movb dataIn(%esi), %cl
movb %cl, disk2(%edi)
inc %esi
movb dataIn(%esi), %cl
movb %cl, disk3(%edi)
xor disk1(%edi),%cl
xor disk2(%edi),%cl
movb %cl, disk4(%edi)
inc %esi
inc %edi
jmp _L1
```

End of the loop

asm source code appended to file lab6.s

```
_L2:
mov %ebp, %esp
pop %ebp
ret
```

In this lab, we use *indexed addressing mode* to access the data from $dataIn$ or load data to disks. The syntax will be:

$$location(index)$$

It is similar to the syntax of array. If we need element with index i in an array, we use $array[i]$; for ASM, if we want to access first character of $dataIn$ (there is no array in ASM, it is actually just a buffer), we use $dataIn(\%esi)$ with $\%esi = 0$

Since we move a character (8-bit or 1 byte) from $dataIn$ to each disk at a time, we use register $\%cl$ instead of $\%eax$ or $\%ebx$, which are 32-bit. $\%cl$ is first part of register $\%ecx$, and its size is only 8-bit, which is good for our purpose. Moreover, we need to use *movb* instruction also for copying each byte at a time.

2.4 make_corrupt

asm source code appended to file lab6.s

```
make_corrupt:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
mov $0, %edi
_L3:
cmp $DISK_SIZE, %edi
jg _L4
movb $'.', (%eax,%edi,1)
inc %edi
jmp _L3
_L4:
mov %ebp, %esp
pop %ebp
ret
```

This function is for making the disk we want corrupted.”Corrupted” in this case just simply means that we change all the characters in that disk into the character **'.'**. The function will take one parameter, which is the address of the disk need to be corrupted.

We will need the loop for doing so. The loop begins from $\_L3$ and end at $\_L4$, with the condition $\%edi > DISK\_SIZE$ to break out the loop.

Since I create this function for making any disk corrupted, not specific one, I will not use the syntax of *indexed addressing mode* I mentioned before ($disk2(\%edi)$). Instead, I use *indexed indirect addressing mode*:

$$(\%eax, \%edi, 1)$$

which means start at $\%eax$, go $\%edi$ location forward, which each location being 1 byte big.

2.5 print_dataOut

This function is for loading data into *dataOut* (for result) and then print it out. We will use the same technique in function $load_data$ before. The data we need to store in *dataOut* begins from index $i = 9$. Again, the data from each disk we need to access begins from index $j = 7$.

asm source code appended to file lab6.s

```
print_dataOut:
push %ebp
mov %esp, %ebp
mov $9, %esi   # i = 9
mov $7, %edi   # j = 7
_L5:
cmp $DISK_SIZE, %edi
jge _L6
```

Body of the loop

asm source code appended to file lab6.s

```
movb disk1(%edi), %cl  #disk1[j]
movb %cl, dataOut(%esi)          #dataOut[i]
inc %esi
movb disk2(%edi), %cl  #disk2[j]
movb %cl, dataOut(%esi)          #dataOut[i+1]
inc %esi
movb disk3(%edi), %cl  #disk3[j]
movb %cl, dataOut(%esi)          #dataOut[i+2]
inc %esi
inc %edi
jmp _L5
```

asm source code appended to file lab6.s

```
_L6:
```

End of loop and print *dataOut*

asm source code appended to file lab6.s

```
mov $N, %edx
mov $dataOut, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov $1, %edx
mov $endline, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80

mov %ebp, %esp
pop %ebp
ret
```

2.6 rebuild

asm source code appended to file lab6.s

```
rebuild:
push %ebp
mov %esp, %ebp
mov $7, %edi   # j = 7
mov $DISK_SIZE, %edx
_L7:
cmp $DISK_SIZE, %edi
jge _L8
mov 20(%ebp), %eax
movb (%eax,%edi,1), %cl
mov 16(%ebp), %eax
xor (%eax,%edi,1),%cl
mov 12(%ebp), %eax
xor (%eax,%edi,1),%cl
mov 8(%ebp), %eax      #this need rebuilding
movb %cl, (%eax,%edi,1)
inc %edi
jmp _L7
_L8:
mov %ebp, %esp
pop %ebp
ret
```

This function is for fixing/rebuilding the corrupted disk from 3 other disks.
Since I make *make_corrupt* for any disk I want to be corrupted, I need to make this one also general, which means not for any specific disk.
First 3 parameters (20(%*ebp*), 16(%*ebp*), 12(%*ebp*)) are those normal disks (order is not important), and last parameter (8(%*ebp*)) is for the corrupted disk.

We use the same technique with function *make_corrupt*, using the *indexed indirect addressing mode*, and of course, index %*edi* of each disk starts from 7.
The corrupted disk is fixed by doing XOR of 3 other disks.