### Chapter 3:

**Make assembly file:**

*as --32 maximum.s -o maximum.o*

*ld -melf\_i386 maximum.o -o maximum*

*./maximum*

*echo $?*

**edit exe.sh and in open in terminal by:** *bash exe.sh*

**Linker** - program that is responsible for putting the object files together and adding

information to it so that the kernel knows how to load and run it

*.section .data* - program storage

*.section .text* - instructions are in here

*.globl \_start* - start of the program

*\_start:* - label, useful for assembler

*movl $1, %eax* - move number 1 (not address) to register eax, movl <source>, <destination>

*int $0x80* - interrupt, transfers control to Linux kernel to do a system call

**List of registers:**

• %eax

• %ebx

• %ecx

• %edx

• %edi

• %esi

**Special-purpose registers:**

• %ebp

• %esp

• %eip

• %eflags

**Conditional jump** - changes paths based on the results of a previous comparison or calculation

**Unconditional jump** - just goes directly to a different path no matter what

**Types of memory locations:**

• .byte - bytes take up one storage location for each number (0 - 255)

• .int - take up two storage locations for each number (0 - 65535)

• .long - take up four storage locations for each number (0 - 4294967295)

• .ascii - characters each take up one storage location, for example *.ascii "Hello there\0"* takes up 12 bytes

*movl data\_items(,%edi,4), %eax*

*movl BEGINNING\_ADDRESS(,%INDEX\_REGISTER,WORD\_SIZE)*

**Types of jumps:**

• je - jump if the values were equal

• jg - jump if the second value was greater than the first value

• jge - jump if the second value was greater than or equal to the first value

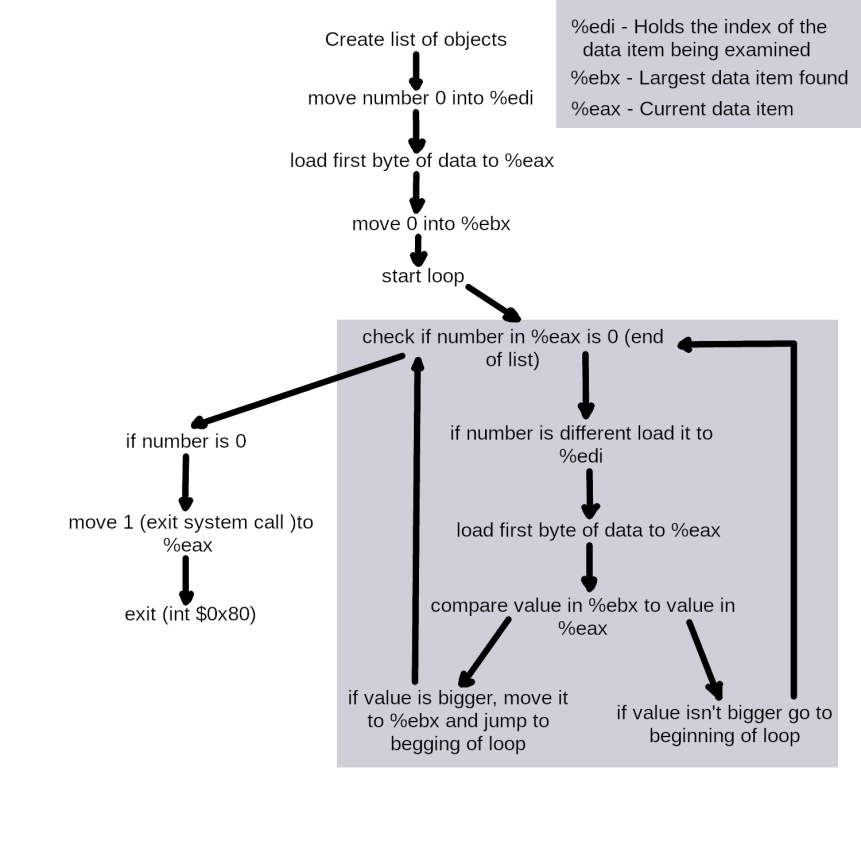
• jl - jump if the second value was less than the first value

• jle - jump if the second value was less than or equal to the first value

• jmp - jump no matter what, this does not need to be preceded by a comparison

**Understanding program *maximum.s:***

1. create list of objects (*data\_items*)
2. move number 0 into index register (*movl $0, %edi*)
3. load the first byte of data and assign 4 storage locations to it, then store number to %eax (*movl data\_items(,%edi,4), %eax*)
4. copy 0 to %ebx (*movl %eax, %ebx*)
5. create loop (*start\_loop:*)
6. check if number in %eax is 0 (*cmpl $0, %eax*)
7. if number in %eax is 0 (that means end of list) exit loop (*je end\_loop*)
8. otherwise if element wasn’t 0, load nex value (*incl %edi*)
9. same as 3. (*movl data\_items(,%edi,4), %eax*)
10. compare current value (from %eax) to biggest value so far (in %ebx) (*cmpl %ebx, %eax*)
11. if value is less or equal go to beginning of loop (*jle start\_loop*)
12. if value is bigger, store in %ebx (*movl %eax, %ebx*)
13. store number 1 in %eax, which is exit call (*movl $1, %eax*)
14. load exit call (*int 0x80*)



**General form of address references:**

ADDRESS\_OR\_OFFSET(%BASE\_OR\_OFFSET,%INDEX,MULTIPLIER)

To calculate address:

FINAL ADDRESS = ADDRESS\_OR\_OFFSET + %BASE\_OR\_OFFSET + MULTIPLIER \* %INDEX

ADDRESS\_OR\_OFFSET and MULTIPLIER must both be constants, while the other

two must be registers

**Addressing modes: (p. 42)**

• direct addressing mode - this loads %eax with the value at memory address ADDRESS e.g. *movl ADDRESS, %eax*

• indexed addressing mode - this starts at string\_start, and adds 1 \* %ecx to that address, and loads the value into %eax e.g. *movl string\_start(,%ecx,1), %eax*

• indirect addressing mode - loads a value from the address indicated by a

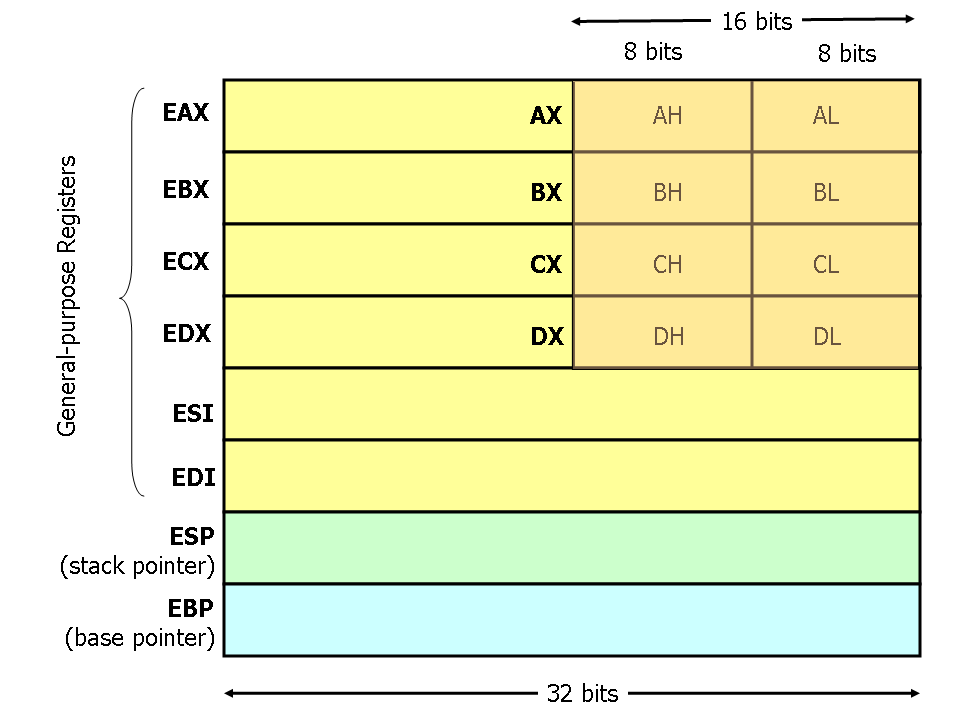
register e.g. *movl (%eax), %ebx*

• immediate mode - load direct values into registers or

memory locations e.g. *movl $12, %eax*

• register addressing mode - register mode simply moves data in or out of a register

**x86 register layout:**



**Chapter 4:**

**Functions** - separate part of code

Parts of functions:

• name - label at the start

• parameters - data items given for processing

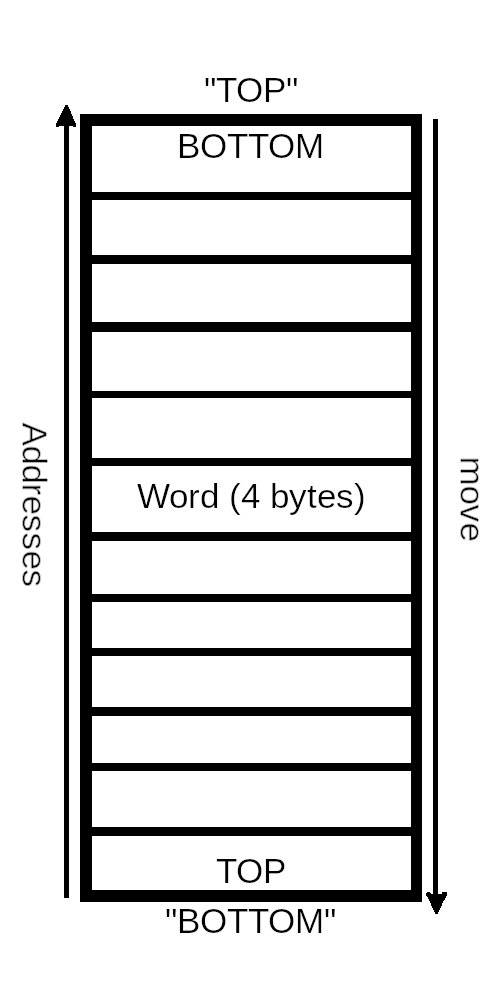
• local variables - temporary data, changing after every executing of program, used only within single function

• static variables - data always same, used rarely within one function

• global variables - data always same, but used within file not single function

• return address - start point of function, used to go back to beginning of processing

• return value - main method of transferring data back to the main program

**Stack:**

*pushl* - push values onto the top of the stack (real bottom)

*popl* - pop values off the top (real top)

*%esp* - current top of the stack

*%ebp* - special register used for accessing function

parameters and local variables

stack frame - includes all of the stack variables used

within function, including parameters, local variables

and the return address

*pushl, %esp* - %esp gets subtracted by 4 (word) and its

grow downward

*popl, %esp* - remove something from the stack, it adds

4 (word) to %esp and puts the previous top value in

specified register

*call <function>* - pushes the address of the next instruction,

which is return address onto the stack, then it modifies the

instruction pointer (%eip) to point to the start of the fuction.

If we want to access the value on the top of the stack

without removing it, we can use an indirect addressing

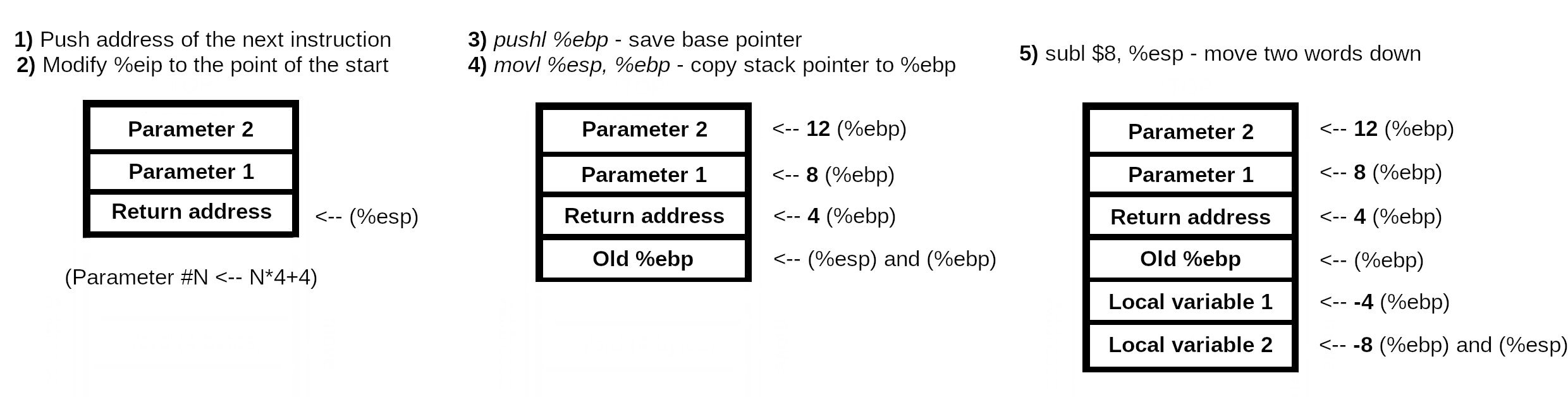
mode e. g. *movl (%esp), %eax*

If we want to access value below top of the stack

we can use e. g. *movl 4(%esp), %eax* (add 4 to %esp)

If we want to hold the pointer to the top of the stack

rather than the value at the top e. g. *movl %esp, %eax*

**

When function is done executing:

1. It stores it’s return value in %eax,
2. Resets the stack (deletes current stack frame),
3. Returns control back to wherever it was called from (ret - instruction pops whatever value is at the top of the stack and sets %eip) that value

Return from function:

1. *movl %ebp, %esp*
2. *popl %ebp*
3. *ret*

**Control flow of func1:**

1. push second argument (power - 3) - *pushl $3*
2. push first argument (base number - 2) - *pushl $2*
3. call power function - *call power*

power:

1. save old base pointer - *pushl %ebp*
2. make stack pointer the base pointer - *movl %esp, %ebp*
3. get room for a local storage - *subl $4, %esp*
4. put first argument to %ebx - *movl 8(%ebp), %ebx*
5. put second argument to %ecx - *movl 12(%ebp), %ebx*
6. store current result (base number - 2 to -4(%ebp)) - *movl %ebx, -4(%ebp)*

power\_loop\_start:

1. if value of power is 1 go to end\_power - *cmpl $1, %ecx*
2. otherwise continue - *je end\_power*
3. move result to %eax - *movl -4(%ebp), %eax*
4. multiply result by base number - *imull %ebx, %eax*
5. store current result in -4(%ebp) - *movl %eax, -4(%ebp)*
6. decrease the power - *decl %ecx*
7. repeat for next power - *jmp power\_loop\_start*

end\_power:

1. return value to %eax - *movl -4(%ebp), %eax*
2. restore stack pointer - *movl %ebp, %esp*
3. restore base pointer - *popl %ebp*
4. return - *ret*
5. move stack pointer back - *addl $8, %esp*
6. save first answer in %eax - *pushl %eax*
7. push second argument- *pushl $2*
8. push first argument - *pushl $5*
9. call power function, do it all over again - *call power*
10. move stack pointer back - *addl $8, %esp*
11. pop second value to %ebx - *popl %ebx*
12. add them together to %ebx - *addl %eax, %ebx*
13. exit - *movl $1, %eax / int $0x80*

**Control flow of factorial:**

1. push factorial number at the top of the stack - *pushl $4*
2. call factorial function - *call factorial*

\_factorial:

1. restore %ebp, create stack frame - *pushl %ebp*
2. use %ebp instead of %esp - *movl %esp, %ebp*
3. move first parameter into %eax - *movl 8(%ebp), %eax*
4. check if we've hit base case - *cmpl $1, %eax*
5. decrease %eax by one - *decl %eax*
6. push %eax onto the stack - *pushl %eax*
7. call factorial again - *call factorial*
8. reload our parameter into %ebx - *movl 8(%ebp), %ebx*
9. multiply that by the result from %eax and store in %eax - *imull %ebx, %eax*
10. if we've hit base case go to end\_factorial: - *je end\_factorial*

end\_factorial:

1. restore %ebp and %esp to where they were before the function started - *movl %ebp, %esp / popl %ebp*
2. return - *ret*
3. clean up the stack - *addl $4, %esp*
4. move return value from %eax to %ebx - *movl %eax, %ebx*
5. exit - *movl $1, %eax / int $0x80*

**Chapter 5:**

**Buffer** - place where OS stores data from file to read, buffers are a fixed size, set by the programmer

*.section .bss*

*.lcomm my\_buffer, 500* - to reserve storage space in .bss section (this section can reserve storage, but it can’t initialize it), this directive, .lcomm, will create a symbol, my\_buffer, that refers to a 500-byte storage location that we can use as a buffer

*movl $my\_buffer, %ecx*

*movl 500, %edx*

*movl 3, %eax*

*int $0x80* - this will read up to 500 bytes into our buffer

**Default files/file descriptors (number referring to file):**

• STDIN - this is the standard input. It is a read-only file, and usually represents keyboard, this is always file descriptor 0

• STDOUT - This is the standard output. It is a write-only file and usually represents screen display, this is always file descriptor 1

• STDERR - This is your standard error. It is a write-only file, and usually represents your screen display, this is always file descriptor 2.

*.equ* - allows you to assign names to numbers e. g. *.equ LINUX\_SYSCALL, 0x80* - any time after that you wrote *LINUX\_SYSCALL* , the assembler would substitue 0x80 for that. So now, you can write: *int $LINUX\_SYSCALL*

**Constant** - value that is assigned when a program assembles or compiles, and is never changed

**General control flow of toupper.s:**

1. CONSTANTS - place where constants are
2. BUFFERS - this is where the data is loaded from the data file and written into the output file
3. STACK POSITIONS - show stack positions and constants
4. convert\_to\_upper function - save the stack pointer, move the function parameters into the appropriate registers for use, load zero into %edi and check to make sure that noone gave us a buffer of zero size
5. convert loop - move a byte into %cl, check to see if that value is in the range of lower-case a to lower-case z, then add the uppercase conversion and store it back into the buffer, check to see if we are at the end of the buffer
6. open files - *open* system call, next the file descriptor of the newly-opened file is stored, then number of arguments are stored at 8(%esp), the name of the program is stored at 12(%esp) and the arguments are stored from 16(%esp) on
7. save the current stack position in %ebp and then reserve some space on the stack to store the file descriptors and open the files
8. open input, then output file
9. read/write loop - read data, write to a buffer, after reading a block, check for end-of-file marker and exit or continue
10. convert\_to\_upper - function is called with a buffer and read all characters
11. issue write system call - move data from buffer to a file and capitalize it
12. clean everything and exit

Registers in this program:

1. *%eax* - contains the system call numbers
2. *%ebx* - contains a pointer to a string that is the name of the file to open
3. *%ecx* - contains the options used for opening the file
4. *%edx* - contains the permissions that are used to open the file
5. *%edi* - handles locations
6. *%cl* - it stores bytes of data
7. *%esp, %ebp* - stack

**Precise control flow of toupper.s:**

CONSTANTS

1. assign system call numbers to names
2. assign file descriptors numbers to names
3. assign call interrupts to names

*.sections .bss*

1. assign buffer parameters

*.sections .text*

1. assign stack positions

*\_start:*

1. save stack pointer - *movl %esp, %ebp*
2. allocate space for our file descriptors on the stack - *subl $ST\_SIZE\_RESERVE, %esp*
3. open input file - *movl $SYS\_OPEN, %eax*
4. input filename into %ebx - *movl ST\_ARGV\_1(%ebp), %ebx*
5. open it in read-only - *movl $O\_RDONLY, %ecx*
6. set permissions to file - *movl $0666, %edx*
7. exit from file - *int $LINUX\_SYSCALL*

*store\_fd\_in:*

1. save the given file descriptor - *movl %eax, ST\_FD\_IN(%ebp)*
2. open the output file - *movl $SYS\_OPEN, %eax*
3. output filename into %ebx - *movl ST\_ARGV\_2(%ebp), %ebx*
4. open it in write mode - *movl $O\_WRONLY\_TRUNC, %ecx*
5. set permissions to file - *movl $0666, %edx*
6. exit from file - *int $LINUX\_SYSCALL*

*store\_fd\_out:*

1. store the file descript - *movl %eax, ST\_FD\_OUT(%ebp)*

*read\_loop\_begin:* <-- main loop

1. read in a block from the input file - *movl $SYS\_READ, %eax*
2. get the input file descriptor - *mocl ST\_FD\_IN(%ebp), %ebx*
3. set the output location - *movl $BUFFER\_DATA, %ecx*
4. set the buffer size - *movl $BUFFER\_SIZE, %edx*
5. exit from file - *int $LINUX\_SYSCALL*
6. check for end-of-file marker - *cmpl $END\_OF\_FILE, %eax*
7. if found or an error, go to the end - *jle end\_loop*

*continue\_read\_loop:*

1. push location of buffer - *pushl $BUFFER\_DATA*
2. push size of the buffer - *pushl %eax*
3. call *convert\_to\_upper*

*CONSTANTS*

1. set lower boundary of our search - *.equ LOWERCASE\_A, ‘a’*
2. set upper boundary of our search - *.equ LOWERCASE\_Z ‘z’*
3. set conversion between upper and lower case - *.equ UPPER\_CONVERSION, ‘A’ - ‘a’*

*STACK STUFF*

1. allocate length of buffer - *.equ ST\_BUFFER\_LEN, 8*
2. allocate buffer - *.equ ST\_BUFFER, 12*

*conver\_to\_upper:*

1. save base pointer - *pushl %ebp*
2. copy stack pointer to %ebp - *movl %esp, %ebp*

*SET UP VARIABLES*

1. set buffer - *movl ST\_BUFFER(%ebp), %ebx*
2. set buffer length - *movl ST\_BUFFER\_LEN(%ebp), %ebx*
3. move 0 to %edi - *movl $0, %edi*
4. if a buffer with zero length was given, just leave - *cmpl $0, %ebx / je end\_convert\_loop*

*convert\_loop:*

1. get the current byte - *movb (%eax,%edi,1), %cl*
2. go to next byte, unless it’s between ‘a’ and ‘z’
3. otherwise convert the byte to uppercase - *addb $UPPER\_CONVERSION, %cl*
4. and store it back - *movl %cl, (%eax,%edi,1)*

*next\_byte:*

1. load next byte - *incl %edi*
2. continue unless we’ve reached the end - *cmpl %edi, %ebx / jne convert\_loop*

*end\_convert\_loop:*

1. leave without return address

*continue\_read\_loop:* <-- continuation

1. get the size back - *popl %eax*
2. restore %esp - *addl $4, %esp*
3. write size of the buffer
4. write file to use
5. write location of the buffer
6. CONTINUE THE LOOP - *jmp read\_loop\_begin*
7. if we’ve hit end, go to the end - *jle end\_loop*

*end\_loop:*

1. close the files
2. exit

**Chapter 6:**

**Database** - a program which handles persistent structured data for you

**Control flow of write-records.s:**

1. include data files (linux.s and record-def.s)
2. make records (make string, repeat to the end of the field and add null character)

*file\_name:*

1. name of the file - *.ascii “test.dat\0”*

*\_start:*

1. assign file descriptor to -4 - *.equ ST\_FILE\_DESCRIPTOR, -4*
2. copy stack pointer to %ebp - *movl %esp, %ebp*
3. allocate space to hold the file descriptor - *subl $4, %esp*
4. open the file - *movl $SYS\_OPEN, $eax / mov file\_name, %ebx*
5. create if it doesn’t exist and open for writing - *movl $0101, %ecx*
6. set permissions for the file - *movl %eax, ST\_FILE\_DESCRIPTOR(%ebp)*
7. exit - *int $LINUX\_SYSCALL*
8. store the file descriptor - *movl %eax, ST\_FILE\_DESCRIPTOR(%ebp)*
9. write records (call write-record function from write-record.s)
10. close the file descriptor
11. exit the program

**Control flow of count-chars.s:**

1. make stack - *pushl %ebp / movl %esp, %ebp*
2. start counting at zero - *movl $0, %ecx*
3. allocate starting address of data - *movl ST\_STRING\_START\_ADDRESS(%ebp), %edx*

*control\_loop\_begin:*

1. grab the current character - *movb (%ebp), %al*
2. check if it’s null - *cmpb $0, %al*
3. if yes, we’re done - *je count\_loop\_end*
4. otherwise increment the counter and the pointer - *incl %ecx / incl %edx*
5. go back to beginning of the loop - *jmp count\_loop\_begin*

*count\_loop\_end:*

1. move the count into %eax - *movl %ecx, %eax*
2. return - *popl %ebp / ret*

**Control flow of read-records.s:**

1. include data files (linux.s and record-def.s)

*file\_name:*

1. load test.dat - *.ascii “test.dat\0”*
2. set up buffer

*\_start:*

1. location of input descriptor - *.equ ST\_INPUT\_DESCRIPTOR, -4*
2. location of output descriptor - *.equ ST\_OUTPUT\_DESCRIPTOR, -8*
3. copy stack pointer to %ebp - *movl %esp, %ebp*
4. allocate space to hold the file descriptor - *subl $8, %esp*
5. open the file in read-only mode
6. save the file descriptor - *movl %eax, ST\_INPUT\_DESCRIPTOR(%ebp)*
7. save the output file descriptor - *movl $STDOUT, ST\_OUTPUT\_DESCRIPTOR(%ebp)*

*record\_read\_loop:*

1. return the number of bytes read
2. if it isn’t the same number we as requested - *cmpl $RECORD\_SIZE, %eax*
3. quit - *jne finished\_reading*
4. otherwise print out first name, but first count it’s size
5. exit from the program

**Control flow of add-year.s:**

1. include data files (linux.s and record-def.s)
2. load test.dat - *.ascii “test.dat\0”*
3. load testout.dat - *.ascii “testout.dat\0”*
4. set up the buffer size
5. set up stack positions of file descriptors
6. copy stack pointer and make room for local variables
7. open file for reading
8. open file for writing
9. return the number of bytes read
10. if it isn’t the same number as we requested - *cmpl $RECORD\_SIZE, %eax*
11. increment the age - *incl record\_buffer + RECORD\_AGE*
12. write the record out
13. exit

**Chapter 7:**

**Robust program** - not crashing program, able on handling errors and continuing

**Driver** - function that, loads your function, supply it with data and check the results

**Chapter 8:**

**Data types for reading functions:**

1. *int* - integer number (4 bytes on x86 processor)
2. *long* - integer number (4 bytes on an x86 processor)
3. *long long* - integer number that’s larger than a long (8 bytes on an x86 processor)
4. *short* - integer number that’s shorter than an int (2 bytes on an x86 processor)
5. *char* - single-byte integer number. This is mostly used for storing character data, since ASCII strings usually are represented with one byte per character
6. *float* - floating-point number (4 bytes on an x86 processor)
7. *double* - floating-point number that is larger than a float (8 bytes on an x86 processor)
8. *unsigned* - modifier used for any of the above types which keeps them from being used as signed quantities
9. *\** - is used to denote that the data isn’t an actual value, but instead is a pointer to a location holding the given value (4 bytes on an x86 processor)
10. *struct* - set of data items that have been put together under a name
11. *typedef* - basically allows you to rename a type

**Useful functions from the C library:**

• *size\_t strlen (const char \*s)* - calculates the size of null-terminated strings

• *int strcmp (const char \*s1, const char \*s2)* - compares two strings alphabetically

• *char \* strdup (const char \*s)* - takes the pointer to a string, creates a new copy in a new location, and returns the new location

• *FILE \* fopen (const char \*filename, const char \*opentype)* - opens a managed and buffered file (allows easier reading and writing than using file descriptors directly)

• *int fclose (FILE \*stream)* - closes a file opened with fopen

• *char \* fgets (char \*s, int count, FILE \*stream)* - fetches a line of characters into string \*s

• *int fputs (const char \*s, FILE \*stream)* - writes a string to the given opened file

• *int fprintf (FILE \*stream, const char \*template, ...)* - is just like printf, but it uses an open file rather than defaulting to using standard output

**Chapter 9:**

**Byte** - the size of a storage location, on x86 processors a byte can hold numbers between 0 and 255

**Word** - the size of a normal register, on x86 processors a word is four bytes long. Most computer operations handle a word at a time

**Address** - a number that refers to a byte in memory

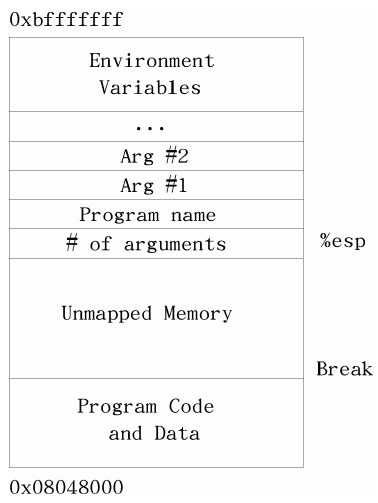
**Pointer** - a register or memory word whose value is an address

**Mapping** - the process of assigning virtual addresses to physical addresses

In order to make the process more efficient, memory is separated out into

groups called **pages**

**Memory Layout of a Linux Program at Startup:**



**Handling of memory accesses in Linux:**

1. The program tries to load memory from a virtual address.

2. The processor, using tables supplied by Linux, transforms the virtual memory address into a physical memory address on the fly.

3. If the processor does not have a physical address listed for the memory address, it sends a request to Linux to load it.

4. Linux looks at the address. If it is mapped to a disk location, it continues on to the next step. Otherwise, it terminates the program with a segmentation fault error.

5. If there is not enough room to load the memory from disk, Linux will move another part of the program or another program onto disk to make room.

6. Linux then moves the data into a free physical memory address.

7. Linux updates the processor’s virtual-to-physical memory mapping tables to reflect the changes.

8. Linux restores control to the program, causing it to re-issue the instruction which caused this process to happen.

9. The processor can now handle the instruction using the newly-loaded memory and translation tables.

The section of memory being managed is commonly referred to as the **Heap**

**Control flow of alloc.s:**

1. reserve space (fill with 0) of heap and break addresses
2. set constants to define the structure of the heap

allocate\_init:

1. standard function stuff
2. find out where the break is
3. set up addresses of heap and break

allocate:

1. standard function stuff
2. set up registers

alloc\_loop\_begin:

1. find out if need more memory
2. grab the size of this memory
3. if the space is available, compare the size to the needed size, if its big enough, go to allocate\_here

next\_location:

1. repeat 9. and 10.

allocate\_here:

1. mark space as unavailable

move\_break:

1. all memory is exhausted and need for more
2. add space for headers structure and break
3. save needed registers
4. reset the break (%ebx has the requested break point)
5. check for an error conditions
6. restore saved registers
7. mark the memory as unavailable
8. record the size of the memory and make sure %eax points to the start of usable memory
9. store the new program break and return the pointer to the allocated memory

deallocate:

1. mark the current memory region as available and allocate will find it next time it is called

**Chapter 10:**

**Boolean operations:**

1. **AND** - takes two bits and returns one bit. AND will return a 1 only if both bits are 1 and a 0 otherwise. For example, 1 AND 1 is 1, but 1 AND 0 is 0, 0 AND 1 is 0, and 0 AND 0 is 0.
2. **OR** - takes two bits and returns one bit. It will return 1 if either of the original bits is. For example, 1 OR 1 is 1, 1 OR 0 is 1, 0 OR 1 is 1, but 0 OR 0 is 0.
3. **NOT** - only takes one bit and returns it’s opposite NOT 1 is 0 and NOT 0 is 1.
4. **XOR** is like OR, except it returns 0 if both bits are 1.
5. **Shift** - a left shift moves each digit of a binary number one space to the left, puts a zero in the ones spot and chops off the furthest digit to the left e. g. 10010111 = 00101110.
6. **Rotate** - a left rotate does the same thing, but takes the furthest digit to the left and puts it in the ones spot e. g. 10010111 = 00101111.

**Masking** - the process of eliminating everything you don’t want, it is accomplished by doing an AND with a number that has the bits we are interested in set to 1.

**Flags** - the individual true/false elements

Some of the *open* system call flags (second parameters):

1. O\_WRONLY - this flag is 0b<31 zeros>1 in binary, or 01 in octal (or any number system for that matter). This says to open the file in write-only mode.
2. O\_RDWR - this flag is 0b<30 zeros>10 in binary, or 02 in octal. This says to open the file for both reading and writing.
3. O\_CREAT - this flag is 0b<25 zeros>1000000 in binary, or 0100 in octal. It means to create the file if it doesn’t already exist.
4. O\_TRUNC - this flag is 0b<22 zeros>1000000000 in binary, or 01000 in octal. It means to erase the contents of the file if the file already exists.
5. O\_APPEND - this flag is 0b<21 zeros>10000000000 in binary, or 02000 in octal. It means to start writing at the end of the file rather than at the beginning.

To use these flags, you simply OR them together in the combination that you want.

For example, to open a file in write-only mode and have it create the file if it

doesn’t exist, I would use O\_WRONLY (01) and O\_CREAT (0100). OR’d together, I

would have 0101.

**Program status register** - this register holds a lot of information about what happens in a computation, it can store results of comparisons or error flags related to operations.

The way a computer handles decimals is by storing them at a fixed precision

(number of significant bits). A computer stores decimal numbers in two parts - the

**exponent** and the **mantissa**. The **mantissa** contains the actual digits that will be

used and the **exponent** is what magnitude the number is. For example, 12345.2 is

stored as 1.23452 \* 10^4. The mantissa is 1.23452 and the exponent is 4. All

numbers are stored as X.XXXXX \* 10^XXXX. The number 1 is stored as

1.00000 \* 10^0.

The x86 processor is a little-endian processor, which means that it stores the "little end", or least-significant byte of its words first. E. g. in number 254 it first stores number 4, then 5 and 2.

**Control flow of integer-to-string.s:**

1. set up stack positions

iteger2string:

1. normal function stuff
2. set up the character count - movl $0, %ecx
3. move the value into %eax - movl ST\_VALUE(%ebp), %eax
4. set up the base number - movl $10, %edi

conversion\_loop:

1. clear out %edx - movl $0, %edx
2. divide value by base number - divl %edi
3. retrieve character for the number stored in %edx - addl $’0’, %edx
4. push this value on the stack - pushl %edx
5. increment the digit count - incl %ecx
6. check if %eax is zero yet - cmpl $0, %eax
7. go to the end\_conversion\_loop if so - je end\_conversion\_loop
8. if %eax isn’t zero go to the conversion\_loop - jmp conversion\_loop

end\_conversion\_loop:

1. get the pointer to the buffer in %edx - movl ST\_BUFFER(%ebp), %edx

copy\_reversing\_loop:

1. pop off to the entire %eax register, but then only move the small part (%al) into the character string - popl %eax / movb %al, (%edx)
2. decreasing %ecx so we know when we are finished - decl %ecx
3. increasing %edx so that it will be pointing to the next byte - incl %edx
4. check to see if we are finished - cmpl $0, %ecx
5. if so, jump to the end of the function - je end\_copy\_reversing\_loop
6. otherwise, repeat the loop - jmp copy\_reversing\_loop

end\_copy\_reversing\_loop:

1. write a null byte and return - movb $0, (%edx)
2. exit from the function

**Control flow of conversion-program.s:**

1. include data file - .include "linux.s"

tmp\_buffer:

1. set up storage location - .ascii "\0\0\0\0\0\0\0\0\0\0\0"

\_start:

1. normal function stuff
2. set up storage for the result - pushl $tmp\_buffer
3. set up number to convert - pushl $824
4. call integer2string
5. get the character count for our system call - pushl $tmp\_buffer
6. call count\_chars
7. the count goes in %edx for SYS\_WRITE - movl %eax, %edx
8. make the system call
9. write a carriage return
10. call write\_newline
11. exit