Συνεχίζουμε την ανάλυση του εξής προγράμματος από το 2ο άρθρο της σειράς Συναρτήσεις σε Assembly:

```
.section .data
.section .text
.globl _start

_start:
    pushl $14
    pushl $29
    call get_max_number
    popl %edx
    popl %edx

    movl %eax, %ebx
    movl $1, %eax

    int $0x80

.type get_max_number, @function
get_max_number:
    pushl %ebp
    movl %esp, %ebp

    .equ first_number, 8
    .equ second_number, 12

    movl first_number(%ebp), %ebx

    cmpl %ebx, second_number(%ebp)
    jg second_number_is_max

    movl first_number(%ebp), %eax
    jmp return_from_function

    second_number_is_max:
    movl second_number(%ebp), %eax

    return_from_function:
    movl %ebp, %esp
    popl %ebp
    ret
```

Αυτό που κάνει το παραπάνω πρόγραμμα είναι να βρίσκει τον μεγαλύτερο από δύο αριθμούς (στην προκειμένη περίπτωση τον 14 και τον 29) μέσω της συνάρτησης `get_max_number` και να επιστρέφει στο λειτουργικό μέσω της κλήσης συστήματος `exit` τον μέγιστο αριθμό αυτό. Αποφεύγω την χρήση συναρτήσεων που εκτυπώνουν μηνύματα στην οθόνη, προς το παρόν, για να αποφύγω την πολυπλοκότητα.

Στο προηγούμενο μέρος είχαμε αναλύσει μέχρι και τις πρώτες γραμμές της συνάρτησης:

```
pushl %ebp
movl %esp, %ebp
```

οι οποίες θεωρούνται στάνταρ (δηλαδή δεν αλλάζουν μορφή) σε συναρτήσεις που ακολουθούν το cdecl και γι' αυτό πολλές φορές ονομάζονται πρόλογος της συνάρτησης. Σε αυτό το σημείο η στοίβα μας έχει την εξής μορφή:

```
old %ebp value      <= (%esp) και (%ebp)
return address      <= 4(%esp) και 4(%ebp)
παράμετρος 29       <= 8(%esp) και 8(%ebp)
παράμετρος 14       <= 12(%esp) και 12(%ebp)
__                  <= 16(%esp) και 16(%ebp)
```

Όπως είχαμε πει, ο ebp χρησιμοποιείται για να δημιουργήσουμε το stack frame της συνάρτησής μας, από το οποίο θα μπορούμε να αναφερόμαστε σε τοπικές μεταβλητές και παραμέτρους με σταθερά offsets. Αυτό σημαίνει πως αν υλοποιήσουμε σωστά το cdecl τότε μπορούμε να είμαστε σίγουροι πως ο αριθμός 29 που περάσαμε σαν παράμετρο, για παράδειγμα, βρίσκεται στάνταρ 8 bytes πάνω από τον ebp, καθ' όλη την εκτέλεση της συνάρτησης. Ακόμα και αν κάνουμε κλήσεις άλλων συναρτήσεων μέσα από την συνάρτησή μας και ο ebp αλλάζει θέση για να δείχνει στα καινούργια stack frame, οφείλουμε με βάση το cdecl να τον επαναφέρουμε έτσι ώστε να δείχνει στο stack frame της συνάρτησής μας.

Αυτή είναι και η κύρια λογική του cdecl. Η δημιουργία stack frame μέσω της στοίβας και του ebp, με το που αρχίζει να τρέχει η συνάρτηση (ο πρόλογος από πάνω) και η επαναφορά του ebp ώστε να δείχνει στην θέση του παλιού stack frame πριν επιστρέψουμε από την συνάρτηση (επίλογος που θα τον δούμε σε λίγο).

Στις επόμενες δύο σειρές

```
.equ first_number, 8
.equ second_number, 12
```

αναθέτω συμβολικά ονόματα στα offsets των παραμέτρων μου ώστε να μπορώ να τα ελέγχω πιο εύκολα και να καταλαβαίνω πιο είναι πιο. Όλες οι εντολές που ξεκινάνε με `.` (τελεία) δεν μεταφράζονται σε κώδικα μηχανής. Είναι απλά εντολές που περνάμε στον linker και στον assembler ώστε να τους καθοδηγήσουμε με διάφορους τρόπους. Η παραπάνω εντολή `.equ` απλά αναθέτει ένα συμβολικό όνομα στις τιμές 8 και 12. Δεν μεταφράζεται σε γλώσσα μηχανής. Αυτό που γίνεται είναι πως κάθε φορά που ο assembler βρίσκει τις λέξεις `first_number` και `second_number` απλά τις αντικαθιστά με τις τιμές 8 και 12 αντίστοιχα. Κάτι σαν τις μακροεντολές με #define των C/C++. Έτσι κάθε φορά που γράφουμε `first_number(%ebp)` εννοούμε

`8(%ebp)` δηλαδή προσθέτουμε 8 bytes στον ebp και μετά διαβάζουμε την τιμή που βρίσκεται σε αυτή τη θέση μνήμης (δηλαδή τη θέση μνήμης `%ebp + 8`).

Οι επόμενες γραμμές απλά κάνουν την σύγκριση και αποθηκεύουν τον μεγαλύτερο αριθμό στον `eax` register:

```
movl first_number(%ebp), %ebx

cmpl %ebx, second_number(%ebp)
jg second_number_is_max

movl first_number(%ebp), %eax
jmp return_from_function

second_number_is_max:
movl second_number(%ebp), %eax
```

Το μόνο αξιοσημείωτο για τον παραπάνω κώδικα είναι μάλλον το γεγονός ότι πρέπει να μετακινήσω μία παράμετρο πρώτα σε έναν register (τον `ebx` στην προκειμένη):

```
movl first_number(%ebp), %ebx
```

πριν κάνω την σύγκριση:

```
cmpl %ebx, second_number(%ebp)
```

γιατί το instruction `movl` θέλει τουλάχιστον ένα operand να είναι register. Άρα κάτι τέτοιο δεν θα δούλευε:

```
cmpl first_number(%ebp), second_number(%ebp)
```

Το αποτέλεσμα της σύγκρισης (στην πραγματικότητα η «σύγκριση» είναι απλά η αφαίρεση των δύο αριθμών) αποθηκεύεται στον `eflags` register αλλά εμείς δεν τον διαβάζουμε απ’ ευθείας μιας και κάτι τέτοιο δεν συνηθίζεται. Αυτό που κάνουμε είναι να χρησιμοποιήσουμε τις εντολές της οικογένειας `jmp` οι οποίες διαβάζουν τον `eflags` για εμάς και μεταφέρουν την ροή του προγράμματος στο αντίστοιχο label ανάλογα με το αποτέλεσμα.

Στην συνέχεια ο μεγαλύτερος αριθμός αποθηκεύεται στον `eax` register και προετοιμαζόμαστε να βγούμε από την συνάρτηση:

```
movl %ebp, %esp
popl %ebp
ret
```

Ας δούμε τι γίνεται στην στοίβα όταν τρέξουμε τις πρώτες δύο εντολές (που

ονομάζονται και επίλογος της συνάρτησης). Πριν τις εντολές, ας θυμηθούμε πως η στοίβα μας έμοιαζε έτσι:

```
old %ebp value      <= 8(%esp) και (%ebp)
return address      <= 12(%esp) και 4(%ebp)
παράμετρος 29       <= 16(%esp) και 8(%ebp)
παράμετρος 14       <= 20(%esp) και 12(%ebp)
__                  <= 24(%esp) και 16(%ebp)
```

Αρχικά αντιγράφουμε την θέση μνήμης του ebp μέσα στον esp register με την εντολή

```
movl %ebp, %esp
```

Αυτό που κάνουμε είναι να αφαιρούμε τις τοπικές μεταβλητές που προσθέσαμε (αν προσθέσαμε) πάνω στην στοίβα. Στο συγκεκριμένο παράδειγμα δεν έχουμε τοπικές μεταβλητές, μιας και η συνάρτησή μας κάνει τόσο απλή δουλειά που δεν τις χρειάζεται, αλλά αν υποθέσουμε πως είχαμε 2 τοπικές μεταβλητές και η στοίβα μας είχε αυτήν την μορφή:

```
τοπική μεταβλητή 1  <= (%esp) και -8(%ebp)
τοπική μεταβλητή 2  <= 4(%esp) και -4(%ebp)
old %ebp value      <= 8(%esp) και (%ebp)
return address      <= 12(%esp) και 4(%ebp)
παράμετρος 29       <= 16(%esp) και 8(%ebp)
παράμετρος 14       <= 20(%esp) και 12(%ebp)
__                  <= 24(%esp) και 16(%ebp)
```

με την παραπάνω εντολή, η στοίβα θα γινόταν έτσι:

```
τοπική μεταβλητή 1  <= -8(%esp) και -8(%ebp)
τοπική μεταβλητή 2  <= -4(%esp) και -4(%ebp)
old %ebp value      <= (%esp) και (%ebp)
return address      <= 4(%esp) και 4(%ebp)
παράμετρος 29       <= 8(%esp) και 8(%ebp)
παράμετρος 14       <= 12(%esp) και 12(%ebp)
__                  <= 16(%esp) και 16(%ebp)
```

Εννοείται πως αφού αλλάξαμε την θέση του esp είναι σαν να αφαιρέσαμε τις μεταβλητές πάνω από την στοίβα, άρα δεν έχει νόημα να μιλάμε για offsets πάνω από την στοίβα. Εμείς πρέπει να υποθέτουμε (ανεξάρτητα με το τι συμβαίνει στην πραγματικότητα) πως με την μετακίνηση του esp προς τα κάτω αφαιρέθηκαν οι τοπικές μεταβλητές και άρα η στοίβα μας πήρε αυτήν την μορφή:

```
old %ebp value      <= (%esp) και (%ebp)
return address      <= 4(%esp) και 4(%ebp)
παράμετρος 29       <= 8(%esp) και 8(%ebp)
```

```
παράμετρος 14       <= 12(%esp) και 12(%ebp)
__                  <= 16(%esp) και 16(%ebp)
```

Ας επανέλθουμε στο πρόγραμμά μας όμως. Η επόμενη εντολή μετά την movl είναι η

```
popl %ebp
```

Δηλαδή βγάζουμε τα πάνω πάνω 4 byte από την κορυφή της στοίβας και τα βάζουμε στον ebp. Μα ποια είναι αυτά τα bytes; Η θέση μνήμης του προηγούμενου stack frame φυσικά! Δηλαδή του stack frame της συνάρτησης που έκανε την κλήση της `get_max_number`. Στο πρόγραμμά μας, αυτή η συνάρτηση είναι η `_start`. Βέβαια εδώ στο παράδειγμά μας δεν είχαμε δημιουργήσει stack frame για την `_start` (γιατί δεν χρησιμοποιήσαμε τοπικές μεταβλητές) και συνεπώς το stack frame στο οποίο δείχνει ο ebp τώρα, δεν περιέχει τίποτα μέσα:

```
return address      <= (%esp) και -12(%ebp)
παράμετρος 29       <= 4(%esp) και -8(%ebp)
παράμετρος 14       <= 8(%esp) και -4(%ebp)
__                  <= 12(%esp) και (%ebp)
```

Η εντολή `ret` κάνει popl το return address από την στοίβα και το αποθηκεύει στον eip register. Αν θυμάστε από το προηγούμενο άρθρο, το return address ήταν απλά η επόμενη εντολή που είναι να εκτελεστεί μετά την call. Δηλαδή η εντολή

```
popl %edx
```

που βρίσκεται κάτω από την εντολή

```
call get_max_number
```

της _start. Το πρόγραμμά μας επιστρέφει στην _start, με άλλα λόγια, και η τιμή επιστροφής βρίσκεται στον eax register. Η εκτέλεση συνεχίζεται από την _start με τις δύο επόμενες εντολές να είναι οι:

```
popl %edx
popl %edx
```

Αυτό που κάνουμε τώρα είναι σημαντικό καθώς καθαρίζουμε την στοίβα από τις παραμέτρους που περάσαμε στην συνάρτηση `get_max_number`. Βέβαια θα μπορούσαμε απλούστερα να προσθέσουμε 8 στον esp για να κάνουμε την ίδια δουλειά, αν δεν θέλαμε να αποθηκεύσουμε κάπου τις τιμές αυτές. Μπορούμε να πούμε, επίσης, πως αυτό θα ήταν και ένα είδος optimization μιας και βλέπουμε ξεκάθαρα πως η εντολή

```
addl $8, %esp
```

είναι μοναδική, μπροστά στις δύο κλήσεις

```
popl %edx
popl %edx
```

που επιφέρουν πρόσθετο overhead στο πρόγραμμά μας. Το πρόγραμμά μας κλείνει την λειτουργία του με τις γραμμές:

```
movl %eax, %ebx
movl $1, %eax
int $0x80
```

οι οποίες απλά καλούν την κλήση συστήματος Linux με αριθμό 1, η οποία είναι η κλήση

```
void exit(int status);
```

που βγαίνει από το πρόγραμμά μας με κωδικό επιστροφής την τιμή `status & 0377`. Η exit ορίζει πως ο προγραμματιστής πρέπει να αποθηκεύσει την παράμετρο `status` στον ebx, και αυτό κάνουμε και εμείς. Στην συνέχεια τοποθετούμε τον αριθμό της κλήσης συστήματός μας στον eax και προκαλούμε ένα interrupt έτσι ώστε ο πυρήνας του Linux να μπορέσει να εκτελέσει την κλήση συστήματος για εμάς.

Οι κλήσεις συστήματος διαφέρουν από τις συναρτήσεις που χρησιμοποιούμε στα προγράμματά μας, γιατί τρέχουν στον πυρήνα και όχι στο desktop του λειτουργικού. Δεν ακολουθούν cdecl και άρα δεν χρειάζεται να βάζουμε τις παραμέτρους τους στο stack. Τις περισσότερες φορές, οι παράμετροι για τις κλήσεις συστήματος περνάνε μέσω συγκεκριμένων registers όπως ορίζεται στο documentation τους. Στην προκειμένη, η παράμετρος για την exit περνάει μέσω του register ebx. Κάθε κλήση συστήματος ορίζει διαφορετικούς κανόνες για τον τρόπο κλήση της. Επίσης, η exit δεν επιστρέφει ακέραιους αριθμούς μεγαλύτερους του 255

Πέρα από αυτήν την παρένθεση, το πρόγραμμά μας επιστρέφει τιμή αλλά δεν την εκτυπώνει πουθενά. Την τιμή που επιστρέψαμε, μπορούμε να την δούμε αν γράψουμε στο τερματικό μας:

```
echo $?
```

αφού εκτελέσουμε το πρόγραμμα.

Μία σημείωση για το παραπάνω πρόγραμμα. Η πρόταση

```
.type get_max_number, @function
```

δεν είναι απαραίτητη. Αυτό που κάνει είναι να λέει στον linker πως η

get_max_number είναι συνάρτηση και πως μπορούμε να την καλέσουμε από άλλα αρχεία κώδικα. Αφού το πρόγραμμά μας είναι ένα μόνο αρχείο μπορούμε να το τρέξουμε κανονικά και χωρίς αυτήν. Ωστόσο είναι καλή πρακτική να την βάζουμε πριν από τις συναρτήσεις μας.

Μία εκδοχή του παραπάνω προγράμματος η οποία χρησιμοποιεί την συνάρτηση printf της στανταρ βιβλιοθήκης της C, για να εκτυπώσει το αποτέλεσμα, και κατά συνέπεια βασίζεται σε αυτήν και θυμίζει περισσότερο πρόγραμμα της C, είναι αυτή:

```
.section .data
    max_number_string:
    .string "Between %d and %d, %d is bigger\n"

    number1:
    .long 14

    number2:
    .long 1555
.section .text
.globl main

main:
    pushl %ebp
    movl %esp, %ebp

    pushl number1
    pushl number2
    call get_max_number
    addl $8, %esp

    pushl %eax
    pushl number1
    pushl number2
    pushl $max_number_string
    call printf
    addl $32, %esp

    movl $0, %eax
    leave
    ret

.type get_max_number, @function
get_max_number:
    pushl %ebp
    movl %esp, %ebp

    .equ first_number, 8
    .equ second_number, 12
```

```
    movl first_number(%ebp), %ebx
    cmpl %ebx, second_number(%ebp)
    jg second_number_is_max

    movl first_number(%ebp), %eax
    jmp return_from_function

    second_number_is_max:
    movl second_number(%ebp), %eax

    return_from_function:
    leave
    ret
```

Αν αποθηκεύσετε το παραπάνω πρόγραμμα σε ένα αρχείο με όνομα `max_number.s` τότε για να το μετατρέψετε σε εκτελέσιμο τρέξτε τις εντολές:

```
as max_number.s -o max_number.o
gcc max_number.o -o  max_number
```

κι αυτό γιατί πρέπει να το συνδέσουμε με την βιβλιοθήκη της C.

Μάλλον η μεγαλύτερη διαφορά που θα παρατηρήσατε με αυτό το πρόγραμμα είναι οι δύο εντολές `leave`. Το `leave` instruction πολύ απλά αντικαθιστά τον επίλογο

```
movl %ebp, %esp
popl %ebp
```

Είναι μία συντόμευση των δύο παραπάνω εντολών. Δεν κάνει τίποτα παραπάνω.

Επίσης στο παραπάνω πρόγραμμα, δημιουργούμε stack frame στην main ακόμα κι αν αυτό δεν είναι απαραίτητο. Δεν χρησιμοποιούμε τοπικές μεταβλητές δηλαδή. Και στην C όμως η main και γενικά οποιαδήποτε συνάρτηση, έχει stack frame ακόμα κι αν δεν χρησιμοποιεί τοπικές μεταβλητές. Όντως, αν το σκεφτείτε λίγο, η δομή του παραπάνω προγράμματος θυμίζει πολύ ένα πρόγραμμα C. Ας δούμε ένα αντίστοιχό του στην C:

```
#include <stdio.h>

static char *max_number_string = "Between %d and %d, %d is
bigger\n";

static const int number1 = 14;

static const int number2 = 1555;
```

```
int get_max_number(int first_number, int second_number)
{
    return (first_number > second_number) ? first_number :
second_number;
}

int main()
{
    printf(max_number_string, number1, number2,
get_max_number(number1, number2));
    return 0;
}
```

Θα σας προτρέψω να κάνετε compile σε assembly το παραπάνω, με τον GCC, και να κάνετε σύγκριση μεταξύ των δύο.

Εδώ κλείνει η σειρά των συναρτήσεων σε Assembly. Ελπίζω να ήταν όσο πιο κατανοητή γινόταν. Αν βρείτε κανένα λάθος πουθενά, γράψτε στα σχόλια ή στείλτε μου PM για να το διορθώσω.