

# Εθνικό Μετσόβιο Πολυτεχνείο

## Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

### 1η Σειρά Ασκήσεων

**Μάθημα:** Λειτουργικά Συστήματα (Τμήμα 1°)

**Εξάμηνο:** 6°

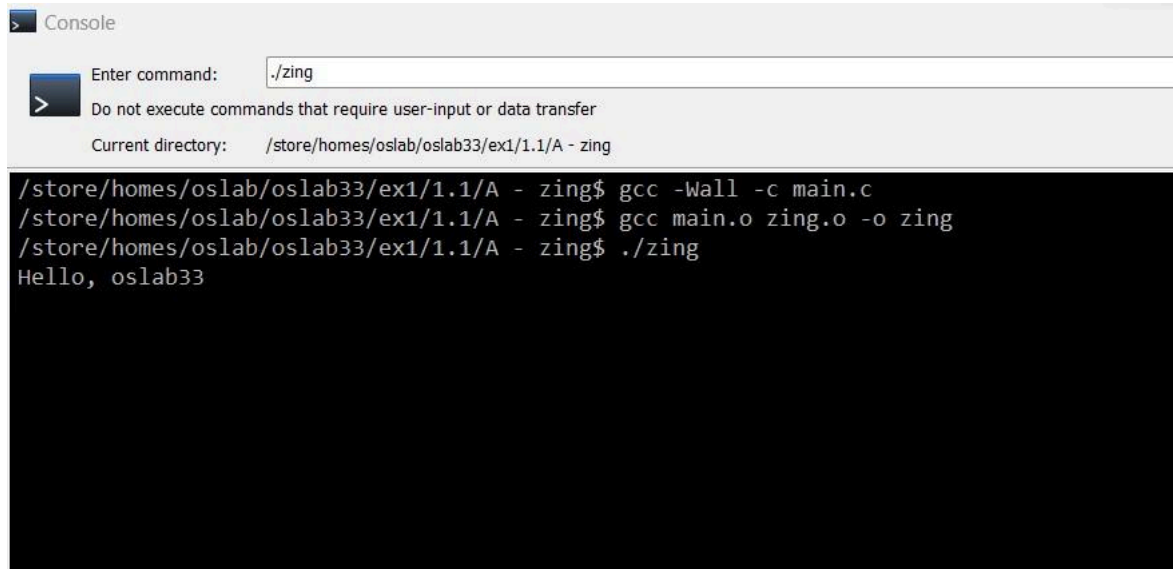
**Ονοματεπώνυμο:** Αλεξοπούλου Γεωργία, Γκενάκου Ζωή

#### *Άσκηση 1.1:*

Για την επίλυση της άσκησης, αρχικά αντιγράφουμε τα αρχεία `zing.h` και `zing.o` από το αρχικό τους directory (`/store/homes/oslab/codes/zing`) στο directory που έχουμε δημιουργήσει για το ερώτημα 1 (`/store/homes/oslab/oslab33/ex1/1.1/A-zing`). Από τις δοθείσες διαφάνειες και τα ζητούμενα της άσκησης, η `main` συνάρτησή μας καλεί τη συνάρτηση `zing.h`, όπως φαίνεται παρακάτω:

```
#include "zing.h"
int main(int argc, char **argv) {
    zing();
    return 0; }
```

Για την εκτέλεση του αρχείου `zing`, αρχικά μεταγλωττίζουμε τη `main` συνάρτησή μας με την εντολή `gcc -Wall -c main.c`, κι έπειτα, για τη δημιουργία του εκτελέσιμου αρχείου `zing` αξιοποιούμε την εντολή `gcc main.o zing.o -o zing`. Τέλος, καλούμε τη συνάρτηση `zing` με το command `./zing`. Τα παραπάνω φαίνονται στην επόμενη φωτογραφία:



```
> Console
Enter command: ./zing
Do not execute commands that require user-input or data transfer
Current directory: /store/homes/oslab/oslab33/ex1/1.1/A - zing

/store/homes/oslab/oslab33/ex1/1.1/A - zing$ gcc -Wall -c main.c
/store/homes/oslab/oslab33/ex1/1.1/A - zing$ gcc main.o zing.o -o zing
/store/homes/oslab/oslab33/ex1/1.1/A - zing$ ./zing
Hello, oslab33
```

### **Ερωτήσεις:**

#### **1. Ποιο σκοπό εξυπηρετεί η επικεφαλίδα;**

Ο σκοπός των επικεφαλίδων (header) `#include <...>` είναι να διευκολύνουν τους προγραμματιστές να συμπεριλάβουν εξωτερικό κώδικα στα προγράμματά τους χωρίς να χρειάζεται να ανησυχούν για τις λεπτομέρειες υλοποίησης. Τα αρχεία κεφαλίδας συχνά περιέχουν οδηγίες προεπεξεργαστή και ορισμούς μακροεντολών που μπορούν να απλοποιήσουν την κωδικοποίηση και να επιτρέψουν πιο αποτελεσματικό και οργανωμένο προγραμματισμό. Οι επικεφαλίδες επίσης αποσκοπούν είτε στην κλήση των κατάλληλων βιβλιοθηκών συναρτήσεων για την εκτέλεση του κώδικα (όπως στο `zing2.c`), είτε στην κλήση ενός αρχείου κώδικα (όπως στο `main.c`). Χωρίς αυτή, ο κώδικας δεν είναι εκτελέσιμος.

#### **2. Ζητείται κατάλληλο Makefile για τη δημιουργία του εκτελέσιμου της άσκησης.**

Όσον αφορά στο Makefile, έχουμε δημιουργήσει ένα αρχείο με directory `/store/homes/oslab/oslab33/ex1/1.1/A-zing`:

```
zing: main.o zing.o
    gcc -o zing main.o zing.o
main.o: main.c zing.h
    gcc -Wall -c main.c

clean: main.o zing
    rm -f main.o zing
```

Σκοπός του αρχείου Makefile είναι η δημιουργία ενός αρχείου που περιλαμβάνει όλες τις εντολές που αναλύσαμε προηγουμένως στην άσκηση. Επομένως είναι λιγότερο χρονοβόρα και πιο αποδοτική διαδικασία.

3. Παράζετε το δικό σας `zing2.o`, το οποίο θα περιέχει `zing()` που θα εμφανίζει διαφορετικό αλλά παρόμοιο μήνυμα με τη `zing()` του `zing.o`. Συμβουλευτείτε το **manual page** της `getlogin(3)`. Αλλάξτε το Makefile ώστε να παράγονται δύο εκτελέσιμα, ένα με το `zing.o`, ένα με το `zing2.o`, επαναχρησιμοποιώντας το κοινό object file `main.o`.

Σε περίπτωση που θέλουμε να δημιουργήσουμε ένα αρχείο `zing2` παρόμοιας φιλοσοφίας με το αρχείο `zing`, δηλαδή να εκτυπώνει π.χ. το μήνυμα “Welcome back, oslab33”, θα δημιουργήσουμε ένα νέο directory στο οποίο αντιγράψουμε εκ νέου τα αρχεία `zing.h` και `zing.o` (/store/homes/oslab/oslab33/B-zing2). Ο πηγαίος κώδικας της `zing2.c` έχει ως εξής:

```
#include <stdio.h>
#include <unistd.h>
void zing(void) {
    printf("Welcome back, %s\n", getlogin() ) ; }
```

Επαναλαμβάνουμε όλα τα προηγούμενα βήματα, μεταγλωττίζοντας επιπλέον το αρχείο `zing2.c` με την εντολή `gcc -Wall -c zing2.c` και μετατρέποντάς το σε εκτελέσιμο αρχείο μέσω της `gcc main.o zing2.o -o zing2`. Τέλος, καλούμε τη συνάρτηση `zing2` με το command `./zing2`:

```
Console
> Enter command: ./zing2
Do not execute commands that require user-input or data transfer
Current directory: /store/homes/oslab/oslab33/ex1/1.1/B - zing2

/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ ls
main.c
zing2.c
zing.h
zing.o
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ gcc -Wall -c main.c
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ ls
main.c
main.o
zing2.c
zing.h
zing.o
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ gcc main.o zing.o -o zing
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ ls
main.c
main.o
zing
zing2.c
zing.h
zing.o
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ gcc -Wall -c zing2.c
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ ls
main.c
main.o
zing
zing2.c
zing2.o
zing.h
zing.o
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ gcc main.o zing2.o -o zing2
/store/homes/oslab/oslab33/ex1/1.1/B - zing2$ ./zing2
Welcome back, oslab33
```

Το Makefile αυτής της διαδικασίας θα πρέπει να μπορεί να μεταγλωττίζει τόσο το αρχείο zing, όσο και το αρχείο zing2. Επομένως, το περιεχόμενο του αρχείου είναι το εξής:

```
all: zing2 zing
zing2: zing2.o main.o
    gcc -o zing2 zing2.o main.o
zing: zing.o main.o
    gcc -o zing zing.o main.o
zing2.o: zing2.c
    gcc -Wall -c zing2.c
main.o: main.c
    gcc -Wall -c main.c
clean: rm -f main.o zing zing2 zing2.o
```

4. Έστω ότι έχετε γράψει το πρόγραμμά σας σε ένα αρχείο που περιέχει 500 συναρτήσεις. Αυτή τη στιγμή κάνετε αλλαγές μόνο σε μία συνάρτηση. Ο κύκλος εργασίας είναι: αλλαγές στον κώδικα, μεταγλώττιση, εκτέλεση, αλλαγές στον κώδικα, κ.ο.κ. Ο χρόνος μεταγλώττισης είναι μεγάλος, γεγονός που σας καθυστερεί. Πώς μπορεί να αντιμετωπισθεί το πρόβλημα αυτό;

Στην περίπτωση εκτέλεσης αρχείου 500 συναρτήσεων, η καθυστέρηση εκτέλεσης οφείλεται στο γεγονός πως οι συναρτήσεις αυτές βρίσκονται σε ένα ενιαίο αρχείο. Προκειμένου να βελτιωθεί η αποδοτικότητα του προγράμματος, κρίνεται σκόπιμος ο κατακερματισμός του κώδικα .c, δημιουργώντας πολλαπλά εκτελέσιμα αρχεία. Με τον τρόπο αυτό σπάει ο κύκλος εξάρτησης, αφού κάθε μια από τις συναρτήσεις βρίσκεται σε ξεχωριστό αρχείο, οπότε η κλήση μίας μόνο συνάρτησης απαιτεί τη μεταγλώττιση κι εκτέλεση ενός μικρότερου αρχείου, κι όχι του αρχικού αρχείου (του οποίου ο όγκος είναι σημαντικά μεγαλύτερος).

5. Ο συνεργάτης σας και εσείς δουλεύατε στο πρόγραμμα `foo.c` όλη την προηγούμενη εβδομάδα. Καθώς κάνατε ένα διάλειμμα και ο συνεργάτης σας δούλεψε στον κώδικα, ακούτε μια απελπισμένη κραυγή. Ρωτάτε τι συνέβει και ο συνεργάτης σας λέει ότι το αρχείο `foo.c` χάθηκε! Κοιτάτε το `history` του φλοιού και η τελευταία εντολή ήταν η: `gcc -Wall -o foo.c foo.c` Τι συνέβη;

Καλώντας την εντολή `gcc -Wall -o foo.c foo.c`, πράγματι παρατηρούμε πως η πηγαίος κώδικας της συνάρτησης `foo` χάνεται από τα αρχεία μας. Αυτό συμβαίνει διότι η παραπάνω εντολή αποσκοπεί στη δημιουργία του εκτελέσιμου αρχείου της `foo.c`, το οποίο όμως ονομάζεται επίσης `foo.c`! Συνεπώς, υπάρχει επικάλυψη, και το εκτελέσιμο αρχείο `foo.c` κάνει `overwrite` στο `source code file foo.c`, γεγονός που οδηγεί σε `execution error`.

### Άσκηση 1.2:

Ο πηγαίος κώδικας της άσκησης είναι ο παρακάτω και η λειτουργία του φαίνεται αφενός από τα επεξηγηματικά σχόλια και αφετέρου από τις παραγράφους στο τέλος.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

// Set buffer size to 1024
#define BUFFSIZE 1024
```

```

// Helper function to write to file
void doWrite(int fd, const char *buff, int len) {
    int bytes_written = 0;
    while (bytes_written < len) {
        // Write data to file descriptor
        int result = write(fd, buff + bytes_written, len -
bytes_written);
        // Check for error in write
        if (result < 0) {
            perror("write error");
            exit(EXIT_FAILURE);
        }
        bytes_written += result;
    }
}

// Helper function to write data from file to file descriptor
void write_file(int fd, const char *infile) {
    // Open input file
    int infile_fd = open(infile, O_RDONLY);
    if (infile_fd < 0) {
        // Check for error in opening input file
        perror("open error");
        exit(EXIT_FAILURE);
    }
    // Read data from input file and write it to file descriptor
    char buffer[BUFSIZE];
    int bytes_read;
    while ((bytes_read = read(infile_fd, buffer, BUFSIZE)) > 0) {
        doWrite(fd, buffer, bytes_read);
    }
    // Close input file
    close(infile_fd);
}

// Main function
int main(int argc, char *argv[]) {
    const char *infile1, *infile2, *outfile;
    int outfile_fd, i;

```

```

// Check for correct number of arguments
if (argc < 3 || argc > 4 ) {
    printf("Usage: %s infile1 infile2 [outfile]\n", argv[0]);
    exit(EXIT_FAILURE);
}

// Set input and output file names
infile1 = argv[1];
infile2 = argv[2];
outfile = argc > 3 ? argv[3] : "fconc.out";

// Check if input and output files are the same
if (strcmp(infile1, outfile) == 0 || strcmp(infile2, outfile) ==
0) {
    printf("Error: Input and output files must be different.\n");
    exit(EXIT_FAILURE);
}

// Open output file
outfile_fd = open(outfile, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR
| S_IWUSR);
if (outfile_fd < 0) {
    // Check for error in opening output file
    perror("open error");
    exit(EXIT_FAILURE);
}

// Write data from input files to output file
write_file(outfile_fd, infile1);
write_file(outfile_fd, infile2);

// Close output file
close(outfile_fd);

// Exit program with success status
exit(EXIT_SUCCESS);
}

```

- Η συνάρτηση "doWrite", λαμβάνει έναν περιγραφητή αρχείου, ένα buffer δεδομένων και το μήκος των δεδομένων που πρόκειται να γραφτούν. Γράφει τα δεδομένα στον περιγραφητή αρχείου σε κομμάτια, χρησιμοποιώντας την κλήση συστήματος "write" εντός ενός βρόχου. Ο βρόχος συνεχίζεται μέχρι να εγγραφούν όλα τα δεδομένα στον περιγραφητή αρχείου. Η συνάρτηση ελέγχει για σφάλματα στην κλήση συστήματος "write" και εάν παρουσιαστεί σφάλμα, εκτυπώνει ένα μήνυμα σφάλματος χρησιμοποιώντας το "printf" και εξέρχεται από το πρόγραμμα με κωδικό κατάστασης αποτυχίας χρησιμοποιώντας "exit".
- Η συνάρτηση "write\_file" λαμβάνει έναν περιγραφητή αρχείου και το όνομα ενός αρχείου εισόδου. Η συνάρτηση επιχειρεί πρώτα να ανοίξει το αρχείο εισόδου χρησιμοποιώντας την κλήση συστήματος "open", χειριζόμενη τυχόν σφάλματα που μπορεί να προκύψουν. Μόλις ανοίξει επιτυχώς το αρχείο εισόδου, η συνάρτηση διαβάζει δεδομένα από αυτό σε κομμάτια χρησιμοποιώντας την κλήση συστήματος "read". Τα δεδομένα εγγράφονται στον καθορισμένο περιγραφητή αρχείου σε αυτά τα κομμάτια ίδιου μεγέθους χρησιμοποιώντας τη συνάρτηση "doWrite". Η λειτουργία συνεχίζει να διαβάζει και να γράφει δεδομένα έως ότου δεν απομένουν άλλα δεδομένα για ανάγνωση από το αρχείο εισόδου. Τέλος, το αρχείο εισόδου κλείνει χρησιμοποιώντας την κλήση συστήματος "close".
- Η "main" λαμβάνει ορίσματα, συγκεκριμένα τα ονόματα δύο αρχείων εισόδου και ένα προαιρετικό όνομα αρχείου εξόδου. Η συνάρτηση ελέγχει πρώτα ότι έχει περάσει ο σωστός αριθμός ορισμάτων. Στη συνέχεια ορίζει τα ονόματα των αρχείων εισόδου και εξόδου ανάλογα και ελέγχει ότι τα αρχεία εισόδου και εξόδου δεν είναι τα ίδια. Η συνάρτηση ανοίγει το αρχείο εξόδου χρησιμοποιώντας την κλήση συστήματος "open" με τα καθορισμένα flags και file permissions. Στη συνέχεια καλεί τη συνάρτηση "write\_file" για κάθε αρχείο εισόδου, γράφοντας τα δεδομένα από κάθε αρχείο εισόδου στο αρχείο εξόδου χρησιμοποιώντας τη συνάρτηση "doWrite". Τέλος, το αρχείο εξόδου κλείνει και το πρόγραμμα εξέρχεται με κωδικό κατάστασης επιτυχίας χρησιμοποιώντας τη λειτουργία «exit». Συνολικά, αυτή η "κύρια" λειτουργία παρέχει έναν τρόπο σύνδεσης των περιεχομένων δύο αρχείων εισόδου και εγγραφής τους σε ένα αρχείο εξόδου, χειριζόμενη τα σφάλματα που μπορεί να προκύψουν κατά τη διάρκεια της διαδικασίας.

### ***Ερωτήσεις:***

1. ***Εκτελέστε ένα παράδειγμα του fclone χρησιμοποιώντας την εντολή strace. Αντιγράψτε το κομμάτι της εξόδου της strace που προκύπτει από τον κώδικα που γράψατε.***

Η εντολή strace είναι ένα εργαλείο που μας επιτρέπει να παρακολουθούμε τα system calls και signals που κάνει ένα πρόγραμμα κατά την εκτέλεσή του. Στο μπλέ πλαίσιο φαίνονται τα system calls που χρησιμοποιήσαμε στον κώδικά μας.



```

/store/homes/oslab/oslab33/ex1/1.2$ strace ./fconc A B
execve("./fconc", ["/fconc", "A", "B"], [/* 16 vars */]) = 0
brk(0) = 0x1da2000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f6fb12000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=29766, ...}) = 0
mmap(NULL, 29766, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9f6fb0a000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0\0\0\1\0\0\0P\34\2\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1738176, ...}) = 0
mmap(NULL, 3844640, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9f6f549000
mprotect(0x7f9f6f6ea000, 2097152, PROT_NONE) = 0
mmap(0x7f9f6f8ea000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a1000) = 0x7f9f6f8ea000
mmap(0x7f9f6f8f0000, 14880, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f9f6f8f0000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f6fb09000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f6fb08000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f6fb07000
arch_prctl(ARCH_SET_FS, 0x7f9f6fb08700) = 0
mprotect(0x7f9f6f8ea000, 16384, PROT_READ) = 0
mprotect(0x7f9f6fb14000, 4096, PROT_READ) = 0
munmap(0x7f9f6fb0a000, 29766) = 0
open("fconc.out", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 3
open("A", O_RDONLY) = 4
read(4, "i tzina \n", 1024) = 9
write(3, "i tzina \n", 9) = 9
read(4, "", 1024) = 0
close(4) = 0
open("B", O_RDONLY) = 4
read(4, "kai i zoi<3\n", 1024) = 12
write(3, "kai i zoi<3\n", 12) = 12
read(4, "", 1024) = 0
close(4) = 0
close(3) = 0
exit_group(0) = ?
+++ exited with 0 +++

```

## Προαιρετικές ερωτήσεις:

- 1) Χρησιμοποιήστε την εντολή *strace* για να εντοπίσετε με ποια κλήση συστήματος υλοποιείται η εντολή *strace*. Υπόδειξη: με *strace -o file* η έξοδος της εντολής τοποθετείται στο αρχείο *file*.

Προκειμένου να εντοπίσουμε με ποια sys calls υπολοποιείται η εντολή *strace*, τρέχουμε την παρακάτω εντολή:

```
strace -c -o totrace strace ls
```

Η επιλογή του *-c* εμφανίζει μια συνοπτική αναφορά των κλήσεων που πραγματοποιήθηκαν. Το αποτέλεσμα αυτής, εγγράφεται από την επιλογή *-o* στο αρχείο *totrace*.

Στο περιεχόμενο του αρχείου που φαίνεται παρακάτω, παρατηρούμε ότι το system call που εμφανίζεται περισσότερο είναι το 'ptrace'. Επομένως το πιο πιθανό είναι ότι η εντολή strace υλοποιείται με την κλήση συστήματος 'ptrace'.

Μπορούμε να επιβεβαιώσουμε τον ισχυρισμό μας, σκεπτόμενοι ότι η κλήση συστήματος ptrace() παρέχει έναν ισχυρό μηχανισμό για τον εντοπισμό κλήσεων συστήματος και διεργασίες εντοπισμού σφαλμάτων. Χρησιμοποιώντας το ptrace(), το strace είναι σε θέση να καταγράψει τις κλήσεις του συστήματος που γίνονται από τη διαδικασία εντοπισμού, χωρίς να τροποποιήσει τη συμπεριφορά ή την απόδοσή του. Αυτό επιτρέπει στο strace να παρέχει μια λεπτομερή καταγραφή των κλήσεων συστήματος που πραγματοποιούνται από τη διαδικασία εντοπισμού, μαζί με πληροφορίες σχετικά με τα ορίσματα και τις τιμές επιστροφής για κάθε κλήση.

```
/store/homes/oslab/oslab33/ex1/extras/1$ cat totrace
```

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.000008	0	480		rt_sigprocmask
0.00	0.000000	0	1		read
0.00	0.000000	0	234		write
0.00	0.000000	0	2		open
0.00	0.000000	0	2		close
0.00	0.000000	0	4	2	stat
0.00	0.000000	0	2		fstat
0.00	0.000000	0	8		mmap
0.00	0.000000	0	4		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	8		rt_sigaction
0.00	0.000000	0	3	3	access
0.00	0.000000	0	1		getpid
0.00	0.000000	0	3		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	244	1	wait4
0.00	0.000000	0	2		kill
0.00	0.000000	0	1		uname
0.00	0.000000	0	482		ptrace
0.00	0.000000	0	1		getuid
0.00	0.000000	0	1		getgid
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	70		process_vm_readv
100.00	0.000008		1559	6	total

- 2) Παρατηρήστε ότι εκτός των απολύτων διευθύνσεων, η μοναδική αλλαγή, στην *assembly* είναι το όρισμα της εντολής *call* (κλήση συνάρτησης). Πού οφείλεται η αλλαγή; Ποιος την έκανε;

Στην άσκηση 1.1 έχουμε την *main* να καλεί μια ρουτίνα *zing*() που βρίσκεται σε διαφορετικό .c/.o file. Χρησιμοποιούμε τον *gdb* και κάνουμε *disassemble* το object file *main.o* και παίρνουμε το αναμενόμενο αποτέλεσμα που αναφέρεται και στην εκφώνηση.

```
oslab33@os-node2:~$ cd ex1/extras/2/B-zing2
oslab33@os-node2:~/ex1/extras/2/B-zing2$ ls
main.c main.o Makefile zing zing2 zing2.c zing2.o zing.h zing.o
oslab33@os-node2:~/ex1/extras/2/B-zing2$ gdb -q main.o
Reading symbols from main.o...
(No debugging symbols found in main.o)
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000000000 <+0>:    push    %rbp
   0x0000000000000001 <+1>:    mov     %rsp,%rbp
   0x0000000000000004 <+4>:    sub     $0x10,%rsp
   0x0000000000000008 <+8>:    mov     %edi,-0x4(%rbp)
   0x000000000000000b <+11>:   mov     %rsi,-0x10(%rbp)
   0x000000000000000f <+15>:   call    0x14 <main+20>
   0x0000000000000014 <+20>:   mov     $0x0,%eax
   0x0000000000000019 <+25>:   leave
   0x000000000000001a <+26>:   ret
End of assembler dump.
(gdb)
```

Τώρα, με τον ίδιο τρόπο κάνουμε *disassemble* την *main* από το εκτελέσιμο *zing*. Και έχουμε πάλι το εξής αποτέλεσμα:

```
oslab33@os-node2:~/ex1/extras/2/B-zing2$ ls
main.c main.o Makefile zing zing2 zing2.c zing2.o zing.h zing.o
oslab33@os-node2:~/ex1/extras/2/B-zing2$ gdb -q zing
Reading symbols from zing...
(No debugging symbols found in zing)
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005d2 <+0>:    push    %rbp
   0x00000000004005d3 <+1>:    mov     %rsp,%rbp
   0x00000000004005d6 <+4>:    sub     $0x10,%rsp
   0x00000000004005da <+8>:    mov     %edi,-0x4(%rbp)
   0x00000000004005dd <+11>:   mov     %rsi,-0x10(%rbp)
   0x00000000004005e1 <+15>:   call    0x400596 <zing>
   0x00000000004005e6 <+20>:   mov     $0x0,%eax
   0x00000000004005eb <+25>:   leave
   0x00000000004005ec <+26>:   ret
End of assembler dump.
(gdb)
```

Παρατηρούμε ότι εκτός των απόλυτων διευθύνσεων, η μοναδική αλλαγή στην assembly είναι το όρισμα της εντολής callq. Αυτή η αλλαγή οφείλεται στο στάδιο του linking και άρα αυτός που ευθύνεται είναι ο linker.

Η εντολή “objdump” μας εμφανίζει πληροφορίες σχετικά με τα object files. Η επιλογή -r λέει στο objdump να εμφανίζει relocation entries για το object file και η επιλογή -d λέει στο objdump να κάνει disassemble τα εκτελέσιμα τμήματα του object file.

Οπότε για να δούμε καλύτερα τι συμβαίνει κάνουμε τα εξής:

```
(gdb) quit
oslab33@os-node2:~/ex1/extras/2/B-zing2$ ls
main.c main.o Makefile zing zing2 zing2.c zing2.o zing.h zing.o
oslab33@os-node2:~/ex1/extras/2/B-zing2$ objdump -r -d main.o

main.o:          file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 10       sub     $0x10,%rsp
 8: 89 7d fc          mov     %edi,-0x4(%rbp)
 b: 48 89 75 f0       mov     %rsi,-0x10(%rbp)
 f: e8 00 00 00 00    callq   14 <main+0x14>
                10: R_X86_64_PC32      zing-0x4
14: b8 00 00 00 00    mov     $0x0,%eax
19: c9               leaveq
1a: c3               retq
oslab33@os-node2:~/ex1/extras/2/B-zing2$
```

Η διαδικασία με το objdump είναι παρόμοια με αυτή που κάναμε προηγουμένως με το gdb, αλλά τώρα μπορούμε να δούμε και τα relocation entries μαζί με την disassembly.. Αυτά τα entries υποδεικνύουν στον linker ότι χρειάζεται να τροποποιήσει το 32-bit PC-relative reference σε μιας συγκεκριμένης θέσης μνήμης (0x14) για να δείχνει τη συνάρτηση zing() κατά τη διάρκεια του χρόνου εκτέλεσης. Ως αποτέλεσμα, η εντολή callq στην disassembly της main στο εκτελέσιμο αρχείο zing έχει την relocated μορφή: callq 0x4005b1 <zing>.

**3) Γράψτε το πρόγραμμα της άσκησης 1.2, ώστε να υποστηρίζει αόριστο αριθμό αρχείων εισόδου (π.χ. 1, 2, 3, 4, ...). Θεωρήστε ότι το τελευταίο όρισμα είναι πάντα το αρχείο εξόδου.**

Οι κύριες αλλαγές που έγιναν στον αρχικό κώδικα περιλαμβάνουν:

- Κατάργηση των δηλώσεων μεταβλητών και των ελέγχων για τα αρχεία εισόδου (infile1, infile2, κ.λπ.) καθώς ο αριθμός των αρχείων εισαγωγής μπορεί να είναι αόριστος.
- Προσθήκη ενός βρόχου για επανάληψη σε όλα τα αρχεία εισόδου που καθορίζονται στα ορίσματα της γραμμής εντολών, ξεκινώντας από το δεύτερο όρισμα μέχρι το προτελευταίο όρισμα (καθώς το τελευταίο όρισμα είναι πάντα το αρχείο εξόδου).
- Έλεγχος εάν κάποιο από τα αρχεία εισόδου έχει το ίδιο όνομα με το αρχείο εξόδου, χρησιμοποιώντας έναν βρόχο επανάληψης ελέγχοντας καθένα από αυτά με το όνομα του αρχείου εξόδου.
- Ενημέρωση του μηνύματος χρήσης ώστε να αντικατοπτρίζει τη νέα σύνταξη στο command line.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

// Set buffer size to 1024
#define BUFFSIZE 1024

// Helper function to write to file
void doWrite(int fd, const char *buff, int len) {
    int bytes_written = 0;
    while (bytes_written < len) {
        // Write data to file descriptor
        int result = write(fd, buff + bytes_written, len -
bytes_written);
        // Check for error in write
        if (result < 0) {
            perror("write error");
            exit(EXIT_FAILURE);
        }
        bytes_written += result;
    }
}

// Helper function to write data from file to file descriptor
void write_file(int fd, const char *infile) {
```

```

// Open input file
int infile_fd = open(infile, O_RDONLY);
if (infile_fd < 0) {
    // Check for error in opening input file
    perror("open error");
    exit(EXIT_FAILURE);
}
// Read data from input file and write it to file descriptor
char buffer[BUFSIZE];
int bytes_read;
while ((bytes_read = read(infile_fd, buffer, BUFSIZE)) > 0) {
    doWrite(fd, buffer, bytes_read);
}
// Close input file
close(infile_fd);
}

// Main function
int main(int argc, char *argv[]) {
    const char *outfile;
    int outfile_fd, i;

    // Check for correct number of arguments
    if (argc < 3) {
        printf("Usage: %s infile1 infile2 ... outfile\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Set output file name
    outfile = argv[argc - 1];

    // Check if input and output files are the same
    for (i = 1; i < argc - 1; i++) {
        if (strcmp(argv[i], outfile) == 0) {
            printf("Error: Input and output files must be
different.\n");
            exit(EXIT_FAILURE);
        }
    }
}

```

```

    // Open output file
    outfile_fd = open(outfile, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR
| S_IWUSR);
    if (outfile_fd < 0) {
        // Check for error in opening output file
        perror("open error");
        exit(EXIT_FAILURE);
    }

    // Write data from input files to output file
    for (i = 1; i < argc - 1; i++) {
        write_file(outfile_fd, argv[i]);
    }

    // Close output file
    close(outfile_fd);

    // Exit program with success status
    exit(EXIT_SUCCESS);
}

```

**4) Εάν τρέξετε το εκτελέσιμο /home/oslab/code/whoops/whoops θα δώσει:**

```
$ /home/oslab/code/whoops/whoops Problem!
```

**Θεωρήστε ότι πραγματικά υπάρχει πρόβλημα. Εντοπίστε το.**

Τρέχοντας το εκτελέσιμο whoops έχουμε:

```

oslab33@os-node2:~/ex1/extras/4$ ls
whoops
oslab33@os-node2:~/ex1/extras/4$ cd whoops
oslab33@os-node2:~/ex1/extras/4/whoops$ ls
whoops
oslab33@os-node2:~/ex1/extras/4/whoops$ ./whoops
Problem!
oslab33@os-node2:~/ex1/extras/4/whoops$

```

Για να εντοπίσουμε το πρόβλημα χρησιμοποιούμε το ‘strace’ όπως και στις παραπάνω ασκήσεις:

```
oslab33@os-node2:~/ex1/extras/4/whoops$ strace ./whoops
execve("./whoops", ["/whoops"], 0x7ffe804a6630 /* 22 vars */) = 0
[ Process PID=69646 runs in 32 bit mode. ]
brk(NULL) = 0x8a40000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7f6a000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=20405, ...}) = 0
mmap2(NULL, 20405, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7f65000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib32/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360\257\1\0004\0\0\0"... , 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1991764, ...}) = 0
mmap2(NULL, 2000744, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7d7c000
mprotect(0xf7d95000, 1875968, PROT_NONE) = 0
mmap2(0xf7d95000, 1396736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19000) = 0xf7d95000
mmap2(0xf7eea000, 475136, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16e000) = 0xf7eea000
mmap2(0xf7f5f000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e2000) = 0xf7f5f000
mmap2(0xf7f62000, 10088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7f62000
close(3) = 0
set_thread_area({entry_number=-1, base_addr=0xf7f6b0c0, limit=0x0fffff, seg_32bit=1, contents=0, read_exec_only=0, limit_in_pages=1,
seg_not_present=0, useable=1}) = 0 (entry_number=12)
mprotect(0xf7f5f000, 8192, PROT_READ) = 0
mprotect(0xf7f9c000, 4096, PROT_READ) = 0
munmap(0xf7f65000, 20405) = 0
openat(AT_FDCWD, "/etc/shadow", O_RDONLY) = -1 EACCES (Permission denied)
write(2, "Problem!\n", 9Problem!
) = 9
exit_group(1) = ?
+++ exited with 1 +++
```

Παρατηρούμε λοιπόν από τις τελευταίες γραμμές, ότι το μήνυμα λάθους “Problem!” εμφανίζεται επειδή δεν έχουμε άδεια για πρόσβαση στο αρχείο /etc/shadow.

Το αρχείο /etc/shadow περιέχει πληροφορίες για τα account του συστήματος και ανήκει στον user root και στο group shadow. Επομένως, ένας απλός user όπως εμείς που ανήκουμε στο group oslab δεν έχουμε πρόσβαση σε αυτό το αρχείο.

```
oslab33@os-node2:~/ex1/extras/4/whoops$ groups
oslab
oslab33@os-node2:~/ex1/extras/4/whoops$ su -
Password:
```

Και παρακάτω φαίνεται πάλι ότι δεν έχουμε τα permissions.

```
oslab33@os-node1:~$ ls -la /etc/shadow
-rw-r----- 1 root shadow 828 Mar 14 12:24 /etc/shadow
oslab33@os-node1:~$
```