Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

Άσκηση 1:
Εισαγωγή στο περιβάλλον προγραμματισμού

ΟΜΑΔΑ: oslab100

Ασπρογέρακας Ιωάννης
Κανατάς Άγγελος-Νικόλαος

Άσκηση 1.1 (Σύνδεση με αρχείο αντικειμένων)

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

- ***main.c***:

```
#include "zing.h"

int main(int argc, char **argv)
{
    zing();
    return 0;
}
```

- ***zing2.c***:

```c
#include <stdio.h>
#include <unistd.h>

void zing(void)
{
    printf("eimaste oi, %s\n", getlogin());
}
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **zing** καθώς και η έξοδος εκτέλεσής του είναι η εξής:

```
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  Makefile  zing2.c  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ gcc -Wall -c main.c
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  main.o  Makefile  zing2.c  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ gcc main.o zing.o -o zing
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  main.o  Makefile  zing  zing2.c  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ ./zing
Hello, oslaba100
```

Παρακάτω φαίνεται επίσης η διαδικασία μεταγλώττισης και σύνδεσης για την δημιουργία του εκτελέσιμου **zing2** καθώς και η έξοδος εκτέλεσής του:

```
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  main.o  Makefile  zing  zing2.c  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ gcc -Wall -c zing2.c
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  main.o  Makefile  zing  zing2.c  zing2.o  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ gcc main.o zing2.o -o zing2
oslaba100@os-node1:~/ex1/1.1$ ls
main.c  main.o  Makefile  zing  zing2  zing2.c  zing2.o  zing.h  zing.o
oslaba100@os-node1:~/ex1/1.1$ ./zing2
eimaste oi, oslaba100
```

Το τελικό **Makefile** φαίνεται παρακάτω:

```
all: zing zing2

zing: main.o zing.o
	gcc -o zing main.o zing.o

zing2: zing2.o main.o
	gcc -o zing2 main.o zing2.o

zing2.o: zing2.c
	gcc -Wall -c zing2.c

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

clean:
	rm -f main.o zing zing2 zing2.o
```

- *"Ποιο σκοπό εξυπηρετεί η επικεφαλίδα;"*

  Το preprocessor directive #include "header.h" επεξεργάζεται από τον preprocessor και ουσιαστικά αντικαθίσταται με το περιεχόμενο του header file. Η χρήση αρχείων επικεφαλίδων είναι απαραίτητη προφανώς αν η υλοποίηση μια συνάρτησης είναι σε διαφορετικό .c αρχείο από αυτό που την καλούμε, οπότε και είναι απαραίτητη η δήλωσή της πριν την κλήση της, η οποία βρίσκεται στο header file. Η υλοποίηση κομματιών κώδικα σε ξεχωριστά .c αρχεία καθώς και η χρήση header files που αυτό απαιτεί είναι θεμελιώδης ιδέα του δομημένου προγραμματισμού (επαναχρησιμοποίηση κώδικα κ.λπ.). Χρήσιμα είναι, ιδιαίτερα σε μεγάλα projects, η χρησιμοποίηση #include guards ή και του #pragma once preprocessor directive, έτσι ώστε να μην έχουμε διπλότυπες δηλώσεις των συναρτήσεων που γίνονται included από διαφορετικά header files (double inclusion).

- *"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;"*

  Το compilation time είναι μεγάλο διότι όλες οι συναρτήσεις είναι στο ίδιο .c file, οπότε έτσι μία αλλαγή μόνο σε μία συνάρτηση αναγκάζει την μεταγλώττιση όλου του κώδικα. Μια καλή λύση λοιπόν είναι ο καταμερισμός των συναρτήσεων σε επιμέρους .c files (ομαδοποίηση ή ακόμα και κάθε μία σε ξεχωριστό .c file), έτσι ώστε μία αλλαγή σε μία μόνο συνάρτηση να απαιτεί μόνο μεταγλώττιση του .c file που την περιέχει και μία σύνδεση για να παραχθεί το τελικό εκτελέσιμο του project. Προφανώς, το προηγούμενο απαιτεί την χρήση header files για λόγους που αναφέραμε προηγουμένως. Επίσης, καλή πρακτική είναι να χρησιμοποιούμε Makefiles έτσι ώστε ο κύκλος εργασίας να είναι λιγότερο χρονοβόρος και άρα περισσότερο αποδοτικός.

- *"Ο συνεργάτης σας και εσείς δουλεύατε στο πρόγραμμα foo.c όλη την προηγούμενη εβδομάδα. Καθώς κάνατε ένα διάλειμμα και ο συνεργάτης σας δούλευε 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;"*

  Η εντολή: **gcc -Wall -o foo.c foo.c** ουσιαστικά δημιουργεί ένα executable file με το όνομα foo.c. Έτσι, επειδή έχει το ίδιο όνομα με το αρχείο foo.c που περιέχει τον πηγαίο κώδικα γίνεται overwrite πάνω σε αυτό οπότε και χάνουμε τον πηγαίο κώδικα. Επειδή ο compiler του εργαστηρίου είναι παλιός δεν λαμβάνει υπόψιν του κάποιο τέτοιο σφάλμα, ενώ στην δική μας περίπτωση στο περιβάλλον Ubuntu (9.4.0) το λαμβάνει υπόψιν και δεν αφήνει να γίνει το compilation (gcc: fatal error: input file 'foo.c' is the same as output file compilation terminated).

Άσκηση 1.2 (Συνένωση δύο αρχείων σε τρίτο)

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

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

#define OFLAGS_AND_MODE_W O_CREAT | O_WRONLY | O_TRUNC,\
	S_IRUSR | S_IWUSR

// write to output-called by write_file()
void doWrite(int fd, int fd_in, const char *buff, int len) //we add fd_in as argument
{                                                          //in case write fails so
	size_t idx=0; //index in buffer                    //we can close the infile
	ssize_t wcnt;
	do {
		wcnt=write(fd, buff+idx, len-idx); //bytes i have wrote
		if(wcnt==-1) {
			perror("write");
			close(fd_in); //close infile
			close(fd); //close outfile
			exit(1);
		}
		idx+=wcnt; //slide the index
	} while(idx<len);
}

// read from input file and write to the ouput file using doWrite()
void write_file(int fd, const char *infile)
{
	char buff[1024]; //buffer
	ssize_t rcnt;
	int fd_in;
	fd_in=open(infile, O_RDONLY);
	if(fd_in==-1) { //error opening the file
		perror("open");
		close(fd); //close outfile
		exit(1);
	}
	for(;;) {
		rcnt=read(fd_in, buff, sizeof(buff)); //bytes i have read
		if (rcnt==0) break; //end of file
		if(rcnt==-1) {
			perror("read");
			close(fd_in); //close infile
			close(fd); //close outfile
			exit(1);
		}
		doWrite(fd, fd_in, buff, rcnt);
	}
	close(fd_in);
}
```

```c
int main(int argc, char **argv)
{
	if(argc!=3 && argc!=4) { //check if arguments are OK
		printf("Usage: ./fconc infile1 infile2 [outfile (default:fconc.out)]\n");
		exit(1);
	} else {
		int fd_out;
		if(argc==3) { //default output
			fd_out=open("fconc.out", OFLAGS_AND_MODE_W);
		} else { //output defined by 3rd argument
			fd_out=open(argv[3], OFLAGS_AND_MODE_W);
		}
		if(fd_out==-1) { //error opening the output file
			perror("open");
			exit(1);
		} else { //check if input files match the output file so we can ignore them (if we don't do this
			//in some cases we will fall in an endless loop), another alternative is to output an error
			//message like: 'Input files must be different from the output file'
			if(((argc==4) && (strcmp(argv[1], argv[3])!=0)) || argc==3) write_file(fd_out, argv[1]);
			if(((argc==4) && (strcmp(argv[2], argv[3])!=0)) || argc==3) write_file(fd_out, argv[2]);
			close(fd_out);
		}
	}
	return 0;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
all: fconc

fconc: fconc.c
	gcc fconc.c -o fconc

ex1:
	./fconc A

ex2:
	./fconc A B

ex3:
	./fconc infile1 infile2
	cat fconc.out

ex4:
	./fconc infile1 infile2 my_output.out
	cat my_output.out

ex5:
	./fconc infile1 infile2 infile2  #copies infile1 to infile2
	cat infile2

clean:
	rm -f fconc fconc.out my_output.out
```

Έτσι, κάνοντας compile για να δημιουργήσουμε το εκτελέσιμο **fconc** και τρέχοντας τα διάφορα ενδεικτικά παραδείγματα που φαίνονται στο παραπάνω Makefile για να διαπιστώσουμε την λειτουργία του προγράμματος έχουμε:

```
oslaba100@os-node1:~/ex1/1.2$ ls
Makefile  fconc.c  infile1  infile2  trace.log
oslaba100@os-node1:~/ex1/1.2$ make
gcc fconc.c -o fconc
oslaba100@os-node1:~/ex1/1.2$ cat infile1
kala tha paei h askhsh
oslaba100@os-node1:~/ex1/1.2$ cat infile2
nomizoume
oslaba100@os-node1:~/ex1/1.2$ make ex1
./fconc A
Usage: ./fconc infile1 infile2 [outfile (default:fconc.out)]
Makefile:7: recipe for target 'ex1' failed
make: *** [ex1] Error 1
oslaba100@os-node1:~/ex1/1.2$ make ex2
./fconc A B
open: No such file or directory
Makefile:10: recipe for target 'ex2' failed
make: *** [ex2] Error 1
oslaba100@os-node1:~/ex1/1.2$ make ex3
./fconc infile1 infile2
cat fconc.out
kala tha paei h askhsh
nomizoume
oslaba100@os-node1:~/ex1/1.2$ make ex4
./fconc infile1 infile2 my_output.out
cat my_output.out
kala tha paei h askhsh
nomizoume
oslaba100@os-node1:~/ex1/1.2$ make ex5
./fconc infile1 infile2 infile2  #copies infile1 to infile2
cat infile2
kala tha paei h askhsh
```

Τώρα, εκτελώντας ένα παράδειγμα της fconc χρησιμοποιώντας την εντολή **strace** έχουμε:

```
oslaba100@os-node1:~/ex1/1.2$ ls
Makefile  fconc  fconc.c  fconc.out  infile1  infile2
oslaba100@os-node1:~/ex1/1.2$ strace ./fconc infile1 infile2
execve("./fconc", ["./fconc", "infile1", "infile2"], [/* 27 vars */]) = 0
brk(0)                                  = 0x1ac4000
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) = 0x7ff686ba3000
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=32730, ...}) = 0
mmap(NULL, 32730, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff686b9b000
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\0\3\0>\0\1\0\0\0P\34\2\0\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) = 0x7ff6865da000
mprotect(0x7ff68677b000, 2097152, PROT_NONE) = 0
mmap(0x7ff68697b000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a1000) = 0x7ff68697b000
mmap(0x7ff686981000, 14880, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff686981000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff686b9a000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff686b99000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff686b98000
arch_prctl(ARCH_SET_FS, 0x7ff686b99700) = 0
mprotect(0x7ff68697b000, 16384, PROT_READ) = 0
mprotect(0x7ff686ba5000, 4096, PROT_READ) = 0
munmap(0x7ff686b9b000, 32730)           = 0
open("fconc.out", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 3
open("infile1", O_RDONLY)               = 4
read(4, "kala tha paei h askhsh\n", 1024) = 23
write(3, "kala tha paei h askhsh\n", 23) = 23
read(4, "", 1024)                       = 0
close(4)                                = 0
open("infile2", O_RDONLY)               = 4
read(4, "nomizoume\n", 1024)            = 10
write(3, "nomizoume\n", 10)             = 10
read(4, "", 1024)                       = 0
close(4)                                = 0
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Με μωβ περίγραμμα φαίνονται τα **system calls** που χρησιμοποιήσαμε στον κώδικά μας.

Προαιρετικές ερωτήσεις

1.

Τρέχοντας την παρακάτω εντολή δημιουργούμε ένα αρχείο με όνομα **“trace.log”**, όπου περιέχει τα system calls, τον αριθμό τους κ.α. για την εντολή strace ls. Χρησιμοποιούμε ως παράδειγμα την εντολή strace ls για να εντοπίσουμε με ποια κλήση συστήματος υλοποιείται η εντολή strace.

```
oslaba100@os-node1:~/ex1/extra$ strace -c -o trace.log strace ls
```

Βλέποντας τώρα το περιεχόμενο του αρχείου trace.log έχουμε:

```
oslaba100@os-node1:~/ex1/extra$ cat trace.log
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000012           0       270         1 wait4
  0.00    0.000000           0         1           read
  0.00    0.000000           0       260           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       532           rt_sigprocmask
  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         2           kill
  0.00    0.000000           0         1           uname
  0.00    0.000000           0       534           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        93           process_vm_readv
------ ----------- ----------- --------- --------- ----------------
100.00    0.000012                  1738         6 total
```

Παρατηρούμε, λοιπόν, ότι το system call **ptrace** εμφανίζεται αρκετές φορές και συγκρίνοντάς τα με τα άλλα system calls που ξέρουμε συμπεραίνουμε ότι μάλλον με αυτό υλοποιείται η strace.

Πράγματι, ψάχνοντας στα man pages των strace και ptrace καθώς και στο http://strace.io/ επιβεβαιώνεται το παραπάνω:

> “The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.”

“The operation of strace is made possible by the kernel feature known as ptrace.”.

3.

Τώρα, τροποποιούμε το πρόγραμμα της άσκησης 1.2, ώστε να υποστηρίζει αόριστο αριθμό αρχείων εισόδου (π.χ. 1, 2,3, 4,…). Θεωρούμε ότι το τελευταίο όρισμα είναι πάντα το αρχείο εξόδου. Ο **πηγαίος κώδικας** φαίνεται παρακάτω:

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

#define OFLAGS_AND_MODE_W O_CREAT | O_WRONLY | O_TRUNC,\
	S_IRUSR | S_IWUSR

// write to output-called by write_file()
void doWrite(int fd, int fd_in, const char *buff, int len) //we add fd_in as argument
{                                                          //in case write fails so
	size_t idx=0; //index in buffer                    //we can close the infile
	ssize_t wcnt;
	do {
		wcnt=write(fd, buff+idx, len-idx); //bytes i have wrote
		if(wcnt==-1) {
			perror("write");
			close(fd_in); //close infile
			close(fd); //close outfile
			exit(1);
		}
		idx+=wcnt; //slide the index
	} while(idx<len);
}

//read from input file and write to the output file using doWrite()
void write_file(int fd, const char *infile)
{
	char buff[1024]; //buffer
	ssize_t rcnt;
	int fd_in;
	fd_in=open(infile, O_RDONLY);
	if(fd_in==-1) { //error opening the infile
		perror("open");
		close(fd); //close outfile
		exit(1);
	}
	for(;;) {
		rcnt=read(fd_in, buff, sizeof(buff)); //bytes i have read
		if (rcnt==0) break; //end of file
		if(rcnt==-1) {
			perror("read");
			close(fd_in); //close infile
			close(fd); //close outfile
			exit(1);
		}
		doWrite(fd, fd_in, buff, rcnt);
	}
	close(fd_in);
}


int main(int argc, char **argv)
{
	if(argc<3) { //check if arguments are OK
		printf("Usage: ./fconc_extra infile1 infile2 ... outfile\n");
```

```
        exit(1);
    } else {
        int fd_out;
        fd_out=open(argv[argc-1], OFLAGS_AND_MODE_W);
        if(fd_out==-1) { //error opening the output file
            perror("open");
            exit(1);
        } else {
            int i;
            for(i=1; i<argc-1; i++) { //check if input files match the output file so we can ignore them
                                      ///(if we don't do this in some cases we will fall in an endless loop),
                                      //another alternative is to output an error message like:'Input files
                                      //must be different from the output file'
                if(strcmp(argv[i], argv[argc-1])!=0) write_file(fd_out, argv[i]);
            }
            close(fd_out);
        }
    }
    return 0;
}
```

Η λειτουργία του κώδικα φαίνεται από τα επεξηγηματικά σχόλια. Προφανώς η μόνη διαφοροποίηση από την άσκηση 1.2 είναι κάποιες αλλαγές στην main().

Παρόμοια με την άσκηση 1.2 δημιουργούμε το εκτελέσιμο **fconc_extra** και τρέχουμε διάφορα ενδεικτικά παραδείγματα για να διαπιστώσουμε την λειτουργία του:

```
oslaba100@os-node1:~/ex1/extra/1.2_extra$ ls
Makefile  fconc_extra.c  infile1  infile2  infile3  infile4
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make
gcc fconc_extra.c -o fconc_extra
oslaba100@os-node1:~/ex1/extra/1.2_extra$ cat infile1
isws
oslaba100@os-node1:~/ex1/extra/1.2_extra$ cat infile2
na
oslaba100@os-node1:~/ex1/extra/1.2_extra$ cat infile3
douleuei
oslaba100@os-node1:~/ex1/extra/1.2_extra$ cat infile4
kai ayto
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make ex1
./fconc_extra A
Usage: ./fconc_extra infile1 infile2 ... outfile
Makefile:7: recipe for target 'ex1' failed
make: *** [ex1] Error 1
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make ex2
./fconc_extra A B
open: No such file or directory
Makefile:10: recipe for target 'ex2' failed
make: *** [ex2] Error 1
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make ex3
./fconc_extra infile3 infile4 fconc_extra.out
cat fconc_extra.out
douleuei
kai ayto
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make ex4
./fconc_extra infile1 infile2 infile3 infile4 fconc_extra.out
cat fconc_extra.out
isws
na
douleuei
kai ayto
oslaba100@os-node1:~/ex1/extra/1.2_extra$ make ex5
./fconc_extra infile1 infile2 infile2 #copies infile1 to infile2
cat infile2
isws
```

4.

Τρέχοντας το εκτελέσιμο `/home/oslab/code/whoops/whoops` έχουμε:

```
oslaba100@os-node1:~/ex1/extra$ /home/oslab/code/whoops/whoops
Problem!
```

Τώρα, για να εντοπίσουμε τι πρόβλημα υπάρχει κάνουμε τα παρακάτω:

```
oslaba100@os-node1:~/ex1/extra$ ls
1.2_extra  trace.log  whoops
oslaba100@os-node1:~/ex1/extra$ strace ./whoops
execve("./whoops", ["./whoops"], [/* 27 vars */]) = 0
[ Process PID=4086 runs in 32 bit mode. ]
brk(0)                                  = 0x9c02000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7737000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=32730, ...}) = 0
mmap2(NULL, 32730, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffffff772f000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\233\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1750708, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff772e000
mmap2(NULL, 1755772, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xffffffff7581000
mmap2(0xf7728000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a7000) = 0xffffffff7728000
mmap2(0xf772b000, 10876, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffffff772b000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7580000
set_thread_area({entry_number:-1, base_addr:0xf7580700, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_
mprotect(0xf7728000, 8192, PROT_READ)   = 0
mprotect(0xf775c000, 4096, PROT_READ)   = 0
munmap(0xf772f000, 32730)               = 0
open("/etc/shadow", O_RDONLY)           = -1 EACCES (Permission denied)
write(2, "Problem!\n", 9Problem!
)               = 9
exit_group(1)                           = ?
+++ exited with 1 +++
```

Παρατηρούμε λοιπόν με χρήση της strace ότι το μήνυμα λάθους “Problem!” εμφανίζεται διότι δεν έχουμε το permission να κάνουμε access το αρχείο **`/etc/shadow`.**

Το αρχείο **`/etc/shadow`** περιέχει πληροφορίες για τα account του συστήματος (κρυπτογραφημένα passwords κ.α.) και ανήκει στον user root και στο group shadow. Επομένως, ένας απλός user χωρίς root privileges όπως εμείς που ανήκουμε στο group oslab δεν έχουμε access σε αυτό το αρχείο.

```
oslaba100@os-node1:~/ex1/extra$ groups
oslab
oslaba100@os-node1:~/ex1/extra$ su -
Password:
```

Πηγαίνοντας τώρα στο δικό μας περιβάλλον (Ubuntu) έχουμε:

```
angelos@ubuntu-linux:~/Documents/source_code/c/oslab/ex1/extra$ ls
1.2_extra  trace.log  whoops
angelos@ubuntu-linux:~/Documents/source_code/c/oslab/ex1/extra$ sudo ./whoops
[sudo] password for angelos:
You are not supposed to see this!
angelos@ubuntu-linux:~/Documents/source_code/c/oslab/ex1/extra$ sudo strace ./whoops
execve("./whoops", ["./whoops"], 0x7ffc63038680 /* 25 vars */) = 0
strace: [ Process PID=1106796 runs in 32 bit mode. ]
brk(NULL)                               = 0x964a000
arch_prctl(0x3001 /* ARCH_??? */, 0xfffe5b58) = -1 EINVAL (Invalid argument)
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) = 0xf7ef5000
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=98773, ...}) = 0
mmap2(NULL, 98773, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7edc000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\240\260\1\0004\0\0\0"..., 512) = 512
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\1/?\36aL\271\310)\270\321Y\r\"\214\306"..., 96, 468) = 96
fstat64(3, {st_mode=S_IFREG|0755, st_size=2020556, ...}) = 0
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\1/?\36aL\271\310)\270\321Y\r\"\214\306"..., 96, 468) = 96
mmap2(NULL, 2025152, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7ced000
mprotect(0xf7d06000, 1900544, PROT_NONE) = 0
mmap2(0xf7d06000, 1421312, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19000) = 0xf7d06000
mmap2(0xf7e61000, 475136, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x174000) = 0xf7e61000
mmap2(0xf7ed6000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e8000) = 0xf7ed6000
mmap2(0xf7ed9000, 9920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7ed9000
close(3)                                = 0
set_thread_area({entry_number=-1, base_addr=0xf7ef60c0, limit=0x0fffff, seg_32bit=1, contents=0, read_exec_only=0, limi
mprotect(0xf7ed6000, 8192, PROT_READ)   = 0
mprotect(0xf7f26000, 4096, PROT_READ)   = 0
munmap(0xf7edc000, 98773)               = 0
openat(AT_FDCWD, "/etc/shadow", O_RDONLY) = 3
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL)                               = 0x964a000
brk(0x966b000)                          = 0x966b000
brk(0x966c000)                          = 0x966c000
write(1, "You are not supposed to see this"..., 34You are not supposed to see this!
) = 34
exit_group(0)                           = ?
+++ exited with 0 +++
```

Τώρα που είμαστε στο δικό μας περιβάλλον με την χρήση του sudo αποκτούμε root privileges, οπότε και μπορούμε να κάνουμε access το αρχείο `/etc/shadow`. Έτσι, εμφανίζεται το μήνυμα: **“You are not supposed to see this!”**.

2.

Στην άσκηση 1.1 είχαμε την main να καλεί μία ρουτίνα zing() που βρισκόταν σε διαφορετικό .c/.o file. Χρησιμοποιώντας τον **gdb** κάνουμε disassemble το object file **main.o** και έχουμε:

```
oslaba100@os-node1:~/ex1/1.1$ ls
Makefile  main.c  main.o  zing  zing.h  zing.o  zing2  zing2.c  zing2.o
oslaba100@os-node1:~/ex1/1.1$ gdb -q main.o
Reading symbols from main.o...(no debugging symbols found)...done.
(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>:    callq  0x14 <main+20>
   0x0000000000000014 <+20>:    mov    $0x0,%eax
   0x0000000000000019 <+25>:    leaveq
   0x000000000000001a <+26>:    retq
End of assembler dump.
(gdb)
```

Τώρα, κάνοντας disassemble την main από το εκτελέσιμο **zing** έχουμε επίσης:

```
oslaba100@os-node1:~/ex1/1.1$ ls
Makefile  main.c  main.o  zing  zing.h  zing.o  zing2  zing2.c  zing2.o
oslaba100@os-node1:~/ex1/1.1$ gdb -q zing
Reading symbols from zing...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400596 <+0>:     push   %rbp
   0x0000000000400597 <+1>:     mov    %rsp,%rbp
   0x000000000040059a <+4>:     sub    $0x10,%rsp
   0x000000000040059e <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000004005a1 <+11>:    mov    %rsi,-0x10(%rbp)
   0x00000000004005a5 <+15>:    callq  0x4005b1 <zing>
   0x00000000004005aa <+20>:    mov    $0x0,%eax
   0x00000000004005af <+25>:    leaveq
   0x00000000004005b0 <+26>:    retq
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**.

Για να δούμε καλύτερα τι συμβαίνει κάνουμε τα παρακάτω:

```
oslaba100@os-node1:~/ex1/1.1$ ls
Makefile  main.c  main.o  zing  zing.h  zing.o  zing2  zing2.c  zing2.o
oslaba100@os-node1:~/ex1/1.1$ 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
```

Το παραπάνω δεν διαφέρει από αυτό που κάναμε προηγούμενως με τον gdb, απλά εμφανίζονται και τα **relocation entries** μαζί με την disassembly. Το relocation entry ουσιαστικά "λέει" στον linker να κάνει modify το 32-bit PC-relative reference που ξεκινάει σε απόλυτη απόσταση 0x14 έτσι ώστε να δείχνει την ρουτίνα zing() στον χρόνο εκτέλεσης (δεν θα έπρεπε να είναι `callq 10 <main+0x10>` ;). Έτσι, η εντολή callq στην disassembly της main στο εκτελέσιμο zing έχει την **relocated μορφή** που φαίνεται παραπάνω (`callq 0x4005b1 <zing>`).