Operating Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-562
Date: 2025-10-17
Due: 2025-10-24

Problem Sheet #7

Problem 7.1: *linking* (2+1+1 = 4 points)

The following C source files are compiled separately into object files and afterwards linked with other object files into an executable.

```c
/* a.c */

#include <stdio.h>

extern int x;
int y;

static void f(void)
{
    static char z = 'Z';
    puts("a.c: f()");
}

void g(void)
{
    puts("a.c: g()");
    f();
}

void h(void)
{
    puts("a.c: h()");
    g();
}
```

```c
/* b.c */

#include <stdio.h>

extern void h(void);

int x = 1;
static double y = 1;
static char z = 'A';

static void g(void)
{
    puts("b.c: g()");
    h();
}

void f(void)
{
    puts("b.c: f()");
    g();
}
```

a) Which symbols defined in the files `a.c` and `b.c` are

- internally defined symbols not accessible outside of the object file,
- references to externally defined symbols that must be resolved by the linker,
- weak linkable symbols defined in the object file, or
- strong linkable symbols defined in the object file?

Mark the corresponding cell in the following table (we ignore the `puts` symbol).

file	symbol	internal unlinkable symbol	reference of external symbol	weak linkable symbol	strong linkable symbol
a.c	x				
a.c	y				
a.c	f				
a.c	g				
a.c	h				
b.c	x				
b.c	y				
b.c	z				
b.c	f				
b.c	g				

b) What will be printed to the standard output by the following `main()` function? Explain.

```c
/* main.c */

extern void f(void);

int main(void)
{
    f();
    return 0;
}
```

c) What is name mangling and why do programming languages like C++ use name mangling? Why do I sometimes need to use `extern "C" {}` in C++ header files?

Problem 7.2: *memory allocation profiling using library interposition* (2+4 = 6 points)

Write a dynamically loadable library `memprof` implementing wrapper functions for the standard library functions `malloc()`, `calloc()`, `realloc()`, and `free()`. The goal is to write the amount of data allocated and the value returned to the standard error. The standard error can be redirected into a file so that the memory allocation trace can be further analyzed.

a) Write an implementation of the dynamically loadable library `memprof`. An example invocation could look as follows:

```
$ LD_PRELOAD=$(pwd)/memprof.so date 2> date-memprof.csv
```

This shell command runs the program `date` with the `memprof` library preloaded, redirecting the standard error output to the file `date-memprof.csv`.

The output format should be a comma separated values (csv) file with the first field holding the function name, the second the allocation size (where applicable, for `calloc()` the total allocated size), the third the argument pointer (in case of `realloc()` and `free()`, and the fourth the result pointer (if any). Here is the start of an example csv file (including a header line):

```
function,asize,aptr,rptr
malloc,5,,0x558429ff62a0
free,,0x558429ff62a0,
malloc,120,,0x558429ff62c0
malloc,12,,0x558429ff62a0
malloc,792,,0x558429ff6340
malloc,104,,0x558429ff6660
```

b) Use your memory profiler to measure the memory allocations of a Python interpreter printing `hello world`:

```
$ echo "print('hello world')" | \
  LD_PRELOAD=$(pwd)/memprof.so python3 2> python-memprof.csv
```

Analyze the collected information.

(a) What is the total amount of bytes allocated by `malloc`, `calloc`, `realloc`, and `free` calls recorded?

(b) What is the minimum, mean, and maximum allocation size for each function?

(c) What are the 10 most frequently allocated sizes?

(d) Are there cases where `malloc` or `calloc` returned the same pointer, i.e., memory was reused?

The command line tool `miller` may be convenient for data analysis. Include the miller commands used to find the answers to the questions and your raw data file.