Name: ID#: X.500:

CSCI 2021: Practice Final Exam SOLUTION

Spring 2021

University of Minnesota

Exam period: 20 minutes

Points available: 40

Background: Nearby are several C files along with two attempts to compile them on the left. Study these and answer the questions that follow.

```
> gcc vf_weak_var.c vf_strong_func.c vf_main.c   # COMPILE 1
/usr/bin/ld: warning: size of symbol ‘foo’ changed from 4 to 14
/usr/bin/ld: warning: type of symbol ‘foo’ changed from 1 to 2
> file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version
> ./a.out
-573193927
4
> rm a.out

> gcc vf_strong_var.c vf_strong_func.c vf_main.c # COMPILE 2
/usr/bin/ld: multiple definition of ‘foo’;
collect2: error: ld returned 1 exit status
> file a.out
a.out: cannot open ‘a.out’ (No such file or directory)
```

```c
// FILE: vf_main.c
#include <stdio.h>
int foo(int x);
int main(){
  printf("%d\n",foo);
  printf("%d\n",foo(2));
  return 0;
}

// FILE: vf_strong_func.c
int foo(int x){
  return 2*x;
}

// FILE: vf_strong_var.c
int foo = 0;

// FILE: vf_weak_var.c
int foo;
```

Problem 1 (10 pts): Why does `COMPILE 1` succeed while `COMPILE 2` fails? Mention pertinent properties of ELF files in your answer.

SOLUTION: `COMPILE 1` *succeeds because the integer* `int foo` *is uninitialized and therefore weak. It is overridden by the strong symbol* `int foo(int x)` *so the resulting ELF file has only the function version.* `COMPILE 2` *fails as the C file initializes* `int foo=0;` *making both definitions strong. Two strong symbols with the same name cannot exist in an ELF file causing linking to fail.*

Problem 2 (10 pts): Nearby is the output of `pmap` showing page table virtual memory mapping information for a running program called `memory_parts`. Answer the following questions about this output.

(A) The mapped memory references something called `libc-2.26.so`. Describe this entity and what kind of information you would expect to find at the mapped locations.

SOLUTION: This is the C standard library. It is a shared object with the `.so` *extension and is likely to contain binary assembly instructions standard C functions like* `printf()` *and* `malloc()`.

(B) Why does `pmap` only show a limited number of virtual addresses? What would happen if the program attempted to access an address not listed in the output? Example: address `0x00` is not in the listing.

SOLUTION: The page table only contains mapped pages for program. These mapped addresses are what is shown. The large number of other addresses are unmapped. Attempting to access these unmapped addresses will result in errors such as ***`segmentation faults`****; this usually causes the program to be immediately terminated.*

```
> pmap 7986
7986:   ./memory_parts
00005579a4abd000      4K r-x-- memory_parts
00005579a4cbd000      4K r---- memory_parts
00005579a4cbe000      4K rw--- memory_parts
00005579a4cbf000      4K rw---   [ anon ]
00005579a53aa000    132K rw---   [ heap ]
00007f441f2e1000   1720K r-x-- libc-2.26.so
00007f441f48f000   2044K ----- libc-2.26.so
00007f441f68e000     16K r---- libc-2.26.so
00007f441f692000      8K rw--- libc-2.26.so
00007f441f694000     16K rw---   [ anon ]
00007f441f698000    148K r-x-- ld-2.26.so
00007f441f88f000      8K rw---   [ anon ]
00007f441f8bb000      4K r---- gettysburg.txt
00007f441f8bc000      4K r---- ld-2.26.so
00007f441f8bd000      4K rw--- ld-2.26.so
00007f441f8be000      4K rw---   [ anon ]
00007fff96ae1000    132K rw---   [ stack ]
00007fff96b48000     12K r----   [ anon ]
00007fff96b4b000      8K r-x--   [ anon ]
 total             4276K
```

Problem 3 (10 pts): Adja Centblock is trying to understand why, once she `malloc()`'s an array, she cannot change its size to be larger. She has seen some functions that do this but all of them mention that they may copy an existing array to a new larger location. Use your knowledge of Project 5's El Malloc system to explain roughly to Adja why in most cases she cannot simply "make an array bigger" due to how heap memory allocators are constructed.

SOLUTION: Arrays in C are blocks of contiguous memory. To expand one, you would need to get the memory that is directly "above" it; e.g. the next higher addresses. The heap is doled out in tightly packed blocks of memory. As one `malloc()`*'s a block, it has a fixed size and another block may be allocated right next to it. This means that the next higher memory for an array is often not available as it has already been allocated for another block that is in use. This severely limits the ability to expand the size of arrays or other data in many cases.*

Problem 4 (10 pts): We have seen that a common use of `mmap()` is to map files into the virtual memory space of a program to make it easy for them to processed. However, this is only one of the uses for `mmap()` which is a fundamental tool for programs to interact with the Operating System and hardware. The Loader is the program which will take the disk image of a program like `a.out` and load it into memory to run. Discuss how `mmap()` can be used by the loader to place the required sections of ELF files in memory and establish areas such as the Stack and Heap for that program. In this, mention what important data structure about a program `mmap()` manipulates.

SOLUTION: `mmap()` *manipulates the Page Table for a program, the OS data structure that maps virtual addresses that the program uses to physical addresses. This allows programs to directly map more memory into their address space. When loading a program like* `a.out` *to start it running, the Loader would likely locate the* `.text` *section of an* `a.out` *and* `mmap()` *it into the address space of a program as Executable to allow its code to be executed. Similarly, the Loader would map the* `.data` *section into pages marked Read/Write allowing those variables to change over the course of the program. In order for the program to have a Stack and Heap, the Loader would likely use* `mmap()` *to map Read/Write pages for these. Unlike the* `.text / .data` *sections, the Stack and Heap don't correspond to any part of the* `a.out` *ELF file: the Stack and Heap are only used at run time so aren't saved on disk. To that end, the space that is* `mmap()`*'d is "anonymous" and not associated with any file area.*