OBL1-OS

August 12, 2020

This is a mandatory assignment. Use resources from the course to answer the following questions. **Take care to follow the numbering structure of the assignment in your submission**. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine, for example running natively or virtually on your own PC, or by connecting to `gremlin.stud.iie.ntnu.no` over SSH (Secure Shell). Working in groups is **permitted**, but submissions must be **individual**. **Submissions in PDF are preferred.**

1 The process abstraction

1. Briefly describe what happens when a process is started from a program on disk. A mode switch from kernel- to user-mode must happen. Explain why this is necessary.
2. Download the latest Linux kernel source code from `https://kernel.org` and unpack it. Use a web search engine to help identify the file in the source tree that contains the process descriptor structure (hint: its name is `task_struct`). List the field name from this structure that:

   (a) Stores the process ID

   (b) Keeps track of accumulated virtual memory

   Use the Linux command-line tool `top` to explore other fields relating to running processes. Can you match them to field names in the process descriptor `task_struct`? Name two such fields (besides those listed above).

2 Process memory and segments

The memory region allocated to a process contains the following segments.

- Text segment
- Data segment
- Stack
- Heap

1. Sketch the organisation of a process' address space. Start with high addresses at the top, and the lowest address (0x0) at the bottom.
2. Briefly describe the purpose of each segment. Why is address 0x0 unavailable to the process?
3. What are the differences between a *global*, *static*, and *local* variable?

Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.

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

int var1 = 0;
void main()
{
    int var2 = 1;
    int *var3 = (int *)malloc(sizeof(int)); // Note, since we are using malloc(), var3 will be a
                                            // pointer into the heap!
                                            // So the question is, where is the pointer stored?
    *var3 = 2;
    printf("Address: %x; Value: %d\n", &var1, var1);
    printf("Address: %x; Value: %d\n", &var2, var2);
    printf("Address: %x; Address: %x; Value: %d\n", &var3, var3, *var3);
}
```

3 Program code

1. Compile the example given above using `gcc mem.c -o mem`. Determine the sizes of the **text, data, and bss** segments using the command-line tool `size`.
2. Find the start address of the program using `objdump -f mem`.
3. Disassemble the compiled program using `objdump -d mem`. Capture the output and find the name of the function at the start address. Do a web search to find out what this function does, and why it is useful.
4. Run the program several times (hint: running a program from the current directory is done using the syntax `./mem`). The addresses change between consecutive runs. Why?

4 The stack

Consider the following C program:

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

void func()
{
    int localvar = 2;
    printf("func() with localvar @ 0x%08x\n", &localvar);
    printf("func() frame address @ 0x%08x\n", __builtin_frame_address(0));
    localvar++;
    func();
}

int main()
{
    printf("main() frame address @ 0x%08x\n", __builtin_frame_address(0));
    func();
    exit(0);
}
```

1. Compile the example given above using `gcc stackoverflow.c -o stackoverflow`.
2. Determine the default size of the stack for your Linux system. Hint: use the `ulimit` command (a web search or running the command `ulimit --help` will help find the appropriate command-line flags).
3. Run the program. Describe your observations and find the cause of the error.
4. Run the program and pipe the output to `grep` and `wc -l`:

   ```
   ./stackoverflow | grep func | wc -l
   ```

   What does this number tell you about the stack? How does this relate to the default stack size you found using the `ulimit` command?
5. How much stack memory (in bytes) does each recursive function call occupy?