Operating Systems – spring 2024
Tutorial-Assignment 2

Instructor: Hans P. Reiser

Submission Deadline: Monday, January 29, 2024 – 23:59

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Submission instructions can be found in the introductory section below.

In this assignment you will get familiar with the process abstraction, the basic address space layout of a process, and the Linux process API.

This week, there is a big lab exercise part. Make sure to do that part with attention, it will help a lot with the assignments.

1 Lab Exercises

Question 2.1: Processes in Unix

a. What keeps a process from accessing the memory contents of another process?

b. What are typical regions in a process address space? What is their purpose?

c. What does the `fork()` system call do?

d. Write a small C program that creates a child process. Each process shall print out who it is (i.e., parent or child). The parent shall also print out the child's PID and then wait for the termination of its child.

e. Assume you have to write a shell that can be used to launch arbitrary other programs. Is the `fork()` system call sufficient for that purpose?

Question 2.2: Stacks and Procedures in C

a. *Preliminary notes: You should remeber from the introduction to C programming that local variables of a function are placed on the stack. Unlike static global variables, which during the execution of an application are always at the same location in memory, the address in main memory (on the stack) of such local variables of a function might be different for each invocation of that function (depening on what data currently exists on the stack when the function is invoked).*

   Nevertheless, for accessing such a variable, the CPU needs to know its address. A very common approach for implementing local variables is the use of a stack-frame pointer. With this approach, when entering a function:

   - *The current value of the frame pointer (FP) of the previous function is saved (for example on the stack).*
   - *The current value of the stack pointer (SP) is copied to the frame pointer register (FP)*
   - *The stack pointer is decremented by the size of all local variables.*

   Any time within the execution of the function, the local variables can be found relative to the FP. For example, if there are two local variables of type uint32_t, they can be found at address (FP)-4 and (FP)-8. Likewise, if the caller passes arguments on the stack, these can be found relative to the FP as well. For example, assuming a 32-bit architecture, you could find the saved previous frame pointer at address (FP), the return address of the caller at (FP)+4 and a function argument at (FP)+8.

   Discuss the following code fragment. Try to visualize the stack contents before `foo` calls `bar`, as well as during and after the execution of `foo`. All values are passed via the stack between calling and called function (caller and callee). An `int` is 4 bytes and a `double` is 8 bytes long. Assume a 4-byte aligned, downwards growing pre-decrement stack and the existence of a stack-frame pointer. All local entities within a function are addressed relative to this frame pointer.

```
double foo ( int *p )
{
    int x;
    double y;
    x = *p;
    // do something useful
    return y;
}
```

```
double bar ()
{
   double d;
   int i = 42;
   d = foo( &i );
   return d;
}
```

Question 2.3: Dynamic memory management in C

a. malloc() and free()

`malloc()` is a C library function to dynamically allocate memory on the heap. The function works fully in user space, using memory that was previously allocated by the operating system for the heap of an application.

`malloc()` allocates memory of a specified size (in bytes) and returns a pointer to the beginning of the allocated block. `malloc()` does not know the type of data we are going to store in that memory, and thus the type of the pointer it returns is `void *`, a pointer with no type information. In order to access the pointer, we need to *cast* it to the type we want to use.

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

int main() {
        int *intPtr;

        intPtr = (int *)malloc(sizeof(int));
        if(intPtr == NULL) { printf("malloc failed\n"); exit(1); }

        *intPtr = 42;

        printf("The value is %d\n", *intPtr);
        free(intPtr);
}
```

Extend the following program (calculating prime numbers) such that the required memory for the `primes` array is dynamically allocated, corresponding to the maximum number provided as command line argument.

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

// First argument argv[0] is the program name, argv[1],... the real arguments
// argc is the total number of arguments, including the program name (i.e., it is always at least 1)
int main(int argc, char *argv[])
{
    int i, j, max;
    int *primes = NULL; // size not known at compile time, needs to be allocated dynamically

    if(argc<2) { printf("Usage: %s <number>\n", argv[0]); exit(1); }

    max = atoi(argv[1]);
    printf("Finding prime numbers from 1 to %d\n", max);

    ////
    ////////////////////////////
    //// ADD YOUR CODE HERE
    ////////////////////////////
    ////

    //populating array with naturals numbers
    for(i = 2; i<=max; i++) primes[i] = i;

    //standard prime number sieve
    for(i=2; i*i <= max; i++) {
        if (primes[i] != 0) {
            for(j=2; j<max; j++) {
                if (primes[i]*j > max)
                    break;
                else
                    primes[primes[i]*j]=0;
            }
        }
    }

    for(i = 2; i<=max; i++) {
        if (primes[i]!=0)
            printf("%d\n",primes[i]);
    }

    // All memory of a process will be freed in any case if the process terminates.
    // But in all other cases, make sure to free memory previously allocated with malloc,
    // as soon as you don't need that memory anymore.
    free(primes);
    return 0;
}
```

Question 2.4: Call by reference in C

a. Consider the following short C program. Does it print 12, 42, or another value? Explain why!

```
#include <stdio.h>

void update_value(int val) {
    val = 42;
}

int main() {
    int value = 12;
    update_value(value);
    printf("value is %d\n", value);
}
```

b. What needs to be changed such that the `update_value` function updates the variable `value` in the main function?

c. Now consider this example where the basic type of the variable we want to update is not an integer, but a pointer. Explain what needs to be changed such that the program actually prints the value selected by `update_value()`?

```
#include <stdio.h>

void update_value(char *val) {
    val = "YES";
}

int main() {
    char *answer = "NO";
    update_value(answer);
    printf("My answer is %s\n", answer);
}
```

2 T-Questions (graded quiz on canvas)

T-Question 2.1: The User-Kernel Boundary

a. Are the following statements true or false? (correctly marked: 0.5P) **3 T-pt**

true	false	
☐	☐	Turning off interrupts should be privileged.
☐	☐	Interrupts are synchronous to code.
☐	☐	System call parameters may be passed via the kernel stack.
☐	☐	System call parameters may be passed in registers.
☐	☐	A system call is a voluntary kernel entry.
☐	☐	Interrupts may only happen in the context of the kernel.

b. Alice and Bob engage in a debate about whether the trap instruction is a privileged instruction. Alice argues that it is not privileged, defining privileged instructions as those exclusively used in kernel mode. Explain why Alice takes a valid position. On the contrary, Bob contends that the trap instruction is privileged. Clarify how Bob's claim could be defended. **2 T-pt**

c. Assume that the kernel receives an address (a pointer) as a parameter of a system call from an application. For example, the read() system call receives a pointer to a memory location to which the system call will copy the data read from a file. What is an important functionality of a system call handler in the operating system kernel? **1 T-pt**

true	false	
☐	☐	The kernel must verify that the address lies within the user address range of the application
☐	☐	The kernel must ensure that the address lies within the protected kernel memory, inaccessible for the user application
☐	☐	The kernel must ensure that the address is a valid physical memory address (not a virtual memory address)
☐	☐	The kernel must make sure that the address is within the heap region of the user application (i.e., not the stack/code/static variables part)

T-Question 2.2: Processes

a. What is the difference between a program and a process? **1 T-pt**

b. In the lifecycle of a process on a computer system, explain when a process will become a zombie. What does this mean and why is this useful? And when will a zombie cease to exist? **2 T-pt**

c. A shell process ("A") creates process "B" which in turn creates process "C".
On a Linux system: What is C's parent after process B is killed? **1 T-pt**

P-Question 2.1: Anatomy of a Program (no code, just answer questions about code)

Consider the following C program that does some random computations. Refer to the introductory C slides if you need help with some of the keywords (e.g., const or static).

You can find the source code of the program in folder **p1** of the assignment template on Canvas and build it using `gcc` with the following command line:

```
gcc -g main.c func.c -o out
```

You should now have an executable file called `out`.

main.c:

```
#include <stdlib.h>
#include "func.h"

int main()
{
    int *parg, result;

    parg = (int*)malloc(sizeof(int));
    if (parg == NULL) exit(1);
    *parg = 10;

    result = func(parg);
    free(parg);

    return result;
}
```

func.c:

```
int a = 42;
const int b = 1;

int func(int *parg) {
  static int s = 0;
  int r;
  if (s == 0) {
    r = *parg + a;
    a++;
  } else {
    r = *parg + b;
    s = 1;
  }
  return r;
}
```

func.h:

```
int func(int *parg);
```

a. In which segments of the executable are `a`, `b`, `s`, and `r` stored?
You can use the command `readelf -hSs out` **on skel.ru.is** to verify your solution. Locate each object in the symbol table (`.symtab`) and match the section index given in the Ndx column with the section headers. Hint: The compiler may have renamed `s` to `s.n` with n being some decimal number to prevent name clashes. **2 P-pt**

b. In which address space segments does the variable `parg` and the variable `*parg` (the value (int) that `parg` points to) reside when executing the program? **1 P-pt**

c. Where is `func` and where is the return value of `func()` placed? Verify you solution by disassembling the executable with `objdump -Sd out` and finding the epilogue of `func()`. **1 P-pt**

Note: Submit the answers of your group by modifying and uploading the content of the file `solution.txt` in the template folder `p1`.

Note 2: There is one test available on skel for this part. The test will check if the file format for solution.txt is correct. It will NOT check if your answer is correct.

P-Question 2.2: A Simple Program Starter

Use the files in folder **p2** of the assignment template. You may only modify and upload the file `run_program.c`.
An important feature of every shell is to start external programs.

a. Write a function with the following features: **4 P-pt**

- Starts a program that may be specified by its full path and name (i.e., /usr/bin/who) or only by its name if it is located in one of the directories contained in the `PATH` environment variable. Do not use `system()`.
- Passes the supplied arguments on to the new process.
  Note 1: `args` contains a list of pointers to arguments. The end of the list is indicated with a NULL pointer.
  Note 2: `args[0]` is the first real argument (this is different to `argv` vector in a Linux system call, in which `argv[0]` has a special meaning and `argv[1]` is the first real argument).
- Waits for the newly created process to exit.
- Returns the special error value 255 to report an error condition and 0 to indicate success.

```
int run_program(char *file_path, char *args[]);
```

- If `file_path` is `NULL`, you should return the special error value. If `args` is `NULL`, you should run the program without arguments. (Your program starter should not be terminated by an exception in either of the two cases.)

b. Modify the starter to return the exit status of the previously started/exited process. Keep a return value of 255 to indicate error conditions in your own program. **2 P-pt**

Note: As explained on Piazza, additional test cases are available on skel. You can run these tests with the command
`/home/sty24/bin/runStudTests.sh A2/p2` *`folderwithyourp2solution`*
These tests cover more cases than the main.c program in the template. They are similar to the tests we will use for evaluating your solution (but there is no guarantee that passing all tests will give you full score for the assignment – we may add additional tests and/or manually inspect your code if it indeed correctly solves the problem).

Total:
10 T-pt
10 P-pt