Operating Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-562
Date: 2025-09-19
Due: 2025-09-26

Problem Sheet #3

Problem 3.1: *process creation using `fork()`* (1+1 = 2 points)

Consider the following C program. Assume that all system calls succeed at runtime, that no other processes are created during the execution of the program, and that process identifiers are allocated sequentially.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  static int x = 0;
5
6  int main(int argc, char *argv[])
7  {
8      pid_t p = getpid();
9
10     x++;
11     fork();
12     if (! fork()) {
13         if (fork()) {
14             x++;
15         }
16         x++;
17     }
18
19     printf("p%d: x = %d\n", getpid() - p, x);
20     return 0;
21 }
```

Try to solve this question on paper and not by typing the code into your computer. During an exam, you will have to answer questions like this on paper as well.

a) How many processes does the program create during its execution. Draw the process tree and indicate the value of x on the edges whenever it changes in a process.

b) What is the output produced by the program?

Problem 3.2: *reap all child processes* (1 point)

Implement a function `int reapall(void)` which reaps all child processes of the calling process and returns the number of processes that terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`.

Problem 3.3: *perfect numbers (multi-threading)* (2+3+2 = 7 points)

A *perfect number* is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For example, 6 has the positive divisors $\{1, 2, 3\}$ and $1 + 2 + 3 = 6$.

Write a C program called `perfect` that finds perfect numbers in a range for numbers. The default number range is $[1, 10000]$. The program accepts the `-s` option to set the lower bound and the `-e` option to set the higher bound. Hence, the invocation `perfect -s 100 -e 1000` will search for perfect numbers in the range $[100, 1000]$.

The following function can be used to test whether a given number is a perfect number:

```c
#include <stdint.h>

static int
is_perfect(uint64_t num)
{
    uint64_t i, sum;

    if (num < 2) {
        return 0;
    }

    for (i = 2, sum = 1; i*i <= num; i++) {
        if (num % i == 0) {
            sum += (i*i == num) ? i : i + num / i;
        }
    }

    return (sum == num);
}
```

a) Write a program that searches for perfect numbers in a range of numbers. Your program must support the `-s` and `-e` options to define non-default search intervals.

```
./perfect -s 100 -e 10000
496
8128
```

b) Implement an option `-t` that can be used to define how many concurrent threads should be used to execute the search. If the `-t` option is not present, then a single thread is used to carry out the search. For debugging purposes, implement an option `-v` that writes trace information to the standard error. Below is an invocation with two threads and a verbose trace.

```
./perfect -t 2 -v
perfect: t0 searching [1,5000]
perfect: t1 searching [5001,10000]
6
28
496
8128
perfect: t0 finishing
perfect: t1 finishing
```

c) Determine how the `-t` option impacts the execution time. Pick a search interval that is a reasonable load for your computer hardware and then increase the threading level and determine how the execution time changes. Produce a plot presenting the measurements you have obtained and discuss the results.