Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2014-02-12
Time: 15 min.

Quiz Sheet #1

Problem 1.1: *process creation using* `fork()` (6 points)

Consider the program shown below and explain how many processes it creates during its execution. What is the output produced? Assume that all system calls succeed at runtime and that no other processes are created on the system during the execution of the program and process identifiers are allocated sequentially.

```
#include <stdio.h>                                  /*  1 */
#include <unistd.h>                                 /*  2 */
                                                    /*  3 */
static int x = 0;                                   /*  4 */
                                                    /*  5 */
int main(int argc, char *argv[])                    /*  6 */
{                                                   /*  7 */
    pid_t p = getpid();                             /*  8 */
                                                    /*  9 */
    fork();                                         /* 10 */
    x++;                                            /* 11 */
    if (! fork()) {                                 /* 12 */
        x++;                                        /* 13 */
        if (fork()) {                               /* 14 */
            x++;                                    /* 15 */
        }                                           /* 16 */
    }                                               /* 17 */
                                                    /* 18 */
    printf("p%d: x = %d\n", getpid() - p, x);       /* 19 */
    sleep(60);                                      /* 20 */
    return 0;                                       /* 21 */
}                                                   /* 22 */
```

Solution:

The original process (p0) starts executing the `main()` function. The first `fork()` (line 8) creates a child process (p1), an identical copy of of p0. Both, p0 and p1 then set their copy of x to 1. Both processes create another child process (p2 and p3) in line 10. The two new child processes p2 and p3 increment their copy of x, i.e., their copy of x becomes 2. The processes p2 and p3 then each fork again another child process, leading to the creation of p4 and p5. The processes p2 and p3 then increment their copy of x again. This results in the following process tree:

```
p0-+-p1---p3---p4
   `-p2---p5
```

The output produced is the following (note that lines and results may appear in a different order since the execution order is non-deterministic):

```
p0: x = 1
p1: x = 1
p2: x = 3
p3: x = 3
p4: x = 2
p5: x = 2
```

Problem 1.2: *stream I/O vs. system call I/O* (2+1+1 = 4 points)

The standard C library provides a stream I/O function library (`fopen()`, `puts()`, `printf()`, `fputs()`, `fprintf()`, `fgets()`, `scanf()`, `fscanf()`, `fclose()`, `...`). On Unix/POSIX systems, this stream I/O library is implemented on top of the underlying low-level system call I/O interface (`open()`, `read()`, `write()`, `close`, `...`).

a) What are the advantages of using the stream I/O library? Are there any potential pitfalls?

b) In which cases might the usage of the underlying low-level system calls be appropriate?

c) What are the benefits of the scatter/gather I/O system calls `writev()` and `readv()`?

Solution:

a) The stream I/O library provides, besides convenient formatting functions, buffered I/O: Data is not immediately written to the underlying I/O channel but instead into an internal buffer that gets flushed at appropriate times. The buffered I/O library reduces the number of system calls and thus improves the I/O performance of many programs. However, since data is buffered, programs have to take care that the buffers are getting flushed successfully (runtime errors might not get noticed until the buffers are flushed).

b) In cases where a program needs precise control over the I/O operations, it might be worthwhile to use the low-level system call interface.

c) The `writev()`, `readv()` system calls are useful in situations where data is scattered in memory. Without scatter/gather I/O system calls, either multiple system calls are needed to read/write the data or a single system call plus an additional buffer and copy operations are needed.