Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO20-320202
Date: 2016-11-02
Deadline: 2016-11-08

Problem Sheet #4

Problem 4.1: *memstress* (1+2+2 = 5 points)

Study and compile the following C program called memstress.

```
/*
 * memstress/memstress.c --
 *
 * $ /usr/bin/time -v memstress
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>

static size_t size = 1000;
static long *m;                 /* 2-dimensional array m[size][size] */

static void
init_rbr(long *m, size_t size)
{
    int i, j;
    long l = 0;

    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            m[i * size + j] = l++;
        }
    }
}

static void
init_cbc(long *m, size_t size)
{
    int i, j;
    long l = 0;

    for (j = 0; j < size; j++) {
        for (i = 0; i < size; i++) {
            m[i * size + j] = l++;
        }
    }
}

int
main(int argc, char *argv[])
{
    int opt;
    int num;
    void (*init)(long *m, size_t size) = init_rbr;

    while ((opt = getopt(argc, argv, "rs:")) != -1) {
        switch (opt) {
        case 'r':
            init = init_cbc;
```

```
            break;
        case 's':
            num = atoi(optarg);
            if (num <= 0) {
                fprintf(stderr, "%s: size must be a positive number\n",
                        argv[0]);
                exit(EXIT_FAILURE);
            }
            size = num;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-r]\n", argv[0]);
            exit(EXIT_FAILURE);
        }
    }

    m = malloc(size * size * sizeof(long));
    if (! m) {
        fprintf(stderr, "%s: memory allocation failure\n", argv[0]);
        return EXIT_FAILURE;
    }

    init(m, size);
    free(m);

    return EXIT_SUCCESS;
}
```

a) Explain what the program is doing. What is the difference between `init_rbr` and `init_cbc`?

b) Run the program with and without the -r option and with different values for the -s option under the control of the GNU time program. Use the -v option for GNU time to obtained some detailed statistics. Produce a table that looks like this:

options	malloc	user time	system time	wall clock	major	minor
-s 10						
-s 100						
-s 1000						
-s 10000						
-r -s 10						
-r -s 100						
-r -s 1000						
-r -s 10000						

The column **malloc** should indicate the amount of memory allocated, the other columns are all from the GNU time -v output. Pick values for the -s option that make sense on your machine.

c) Looking at the table that you have produced, what do you observe? Try to interpret the numbers you see.

Solution:

a) The program allocates a large 2-dimensional array of long integers and initializes it with numbers counting up 0. The function `init_rbr` initializes the array in row-by-row order while the function `init_cbc` initializes the array in column-by-column order. The `-r` option selects whether row-by-row order (default) or column-by-column order is used (when `-r` is present). The `-s` option defines the size s of the array. The amount of memory allocated is given by $m = s \cdot s \cdot 8$ on systems where the size of a long is 8 bytes.

b) Measurements taken on cox (Darwin Kernel Version 16.1.0, page size 4096 bytes):

options	malloc	user time	system time	wall clock	major	minor
-s 10	800	0.00	0.00	0:00.00	0	289
-s 100	80000	0.00	0.00	0:00.00	0	309
-s 1000	8000000	0.00	0.00	0:00.01	0	2246
-s 10000	800000000	0.39	0.28	0:00.69	0	195606
-s 20000	3200000000	1.83	1.54	0:04.17	1	781539
-s 30000	7200000000	4.59	4.15	0:09.28	1	1758102
-s 40000	12800000000	8.38	7.55	0:18.92	0	3125328
-r -s 10	800	0.00	0.00	0:00.00	0	289
-r -s 100	80000	0.00	0.00	0:00.00	0	309
-r -s 1000	8000000	0.01	0.00	0:00.01	0	2244
-r -s 10000	800000000	2.36	0.39	0:02.77	1	195604
-r -s 20000	3200000000	13.15	1.84	0:15.16	1	781539
-r -s 30000	7200000000	39.11	5.29	0:46.03	1	1787232
-r -s 40000	12800000000	73.47	9.26	1:27.67	1	3195192

c) Initializing the array in column-by-column order is significantly slower than initializing the array in row-by-row order. Note that the C language in general organizes n-dimensional arrays in row-major-order. By normalizing the page faults with the total amount of memory allocated, we can see that the page fault number is largely correlated to the size of the allocation (plus some additional page faults for loading the program itself). Furthermore, the number of page faults for row-by-row and column-by-column order are roughly the same. Hence, paging does not seem to explain the observed differences.

What this program actually demonstrates is the impact of spatial locality on modern CPU performance. The row-by-row traversal of the array exhibits spatial locality while the column-by-column traversal does not.

Problem 4.2: *pipes and splice* (1+1+3 = 5 points)

Study and compile the following source code on a Linux system:

```
/*
 * pipes/pipes.c --
 */

#define _GNU_SOURCE

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

static void
write_range(int fd, char start, char end)
{
    pid_t pid;

    pid = fork();
    if (pid == -1) {
        perror("fork()");
        return;
    }

    if (pid) {
        return;
    }

    while (1) {
        for (char c = start; c <= end; c++) {
            if (write(fd, &c, 1) == -1) {
                perror("write()");
                exit(EXIT_FAILURE);
            }
        }
    }

    exit(EXIT_SUCCESS);
}

int
main(int argc, char **argv)
{
    int fd[2], rc;

    if (pipe(fd) == -1) {
        perror("pipe()");
        return EXIT_FAILURE;
    }

    if (dup2(fd[0], STDIN_FILENO) == -1) {
        perror("dup2()");
        return EXIT_FAILURE;
    }
    (void) close(fd[0]);

    write_range(fd[1], 'a', 'z');
    write_range(fd[1], 'A', 'Z');
    write_range(fd[1], '0', '9');
```

```
    while (1) {
        rc = splice(STDIN_FILENO, NULL, STDOUT_FILENO, NULL, 4096, SPLICE_F_MORE);
        if (rc == -1) {
            perror("splice()");
            break;
        }
    }

    return EXIT_SUCCESS;
}
```

a) What is the GNU library function `splice()` doing?

b) On Linux, `splice()` is a system call. Can `splice()` be emulated on systems that do not have a `splice()` system call? Explain.

c) What is the main logic of the program? Describe what the program is doing. Is the data written to the standard output by the program deterministic (i.e., does the program always produce the same output)?

Solution:

a) The `splice()` function copies data between two file descriptors without copying between kernel address space and user address space.

b) The functionality of `splice()` can be easily implemented in user-space using `seek()`, `read()` and `write()` system call. Of course, such an implementation will be significantly slower. Some optimizations may be possible in case the file descriptors refer to files that can be memory mapped.

c) The program creates a single pipe and duplicates the read end of the pipe to the standard input file descriptor. It then creates three child processes, each one writing in an endless loop into the write end of the pipe. The first child process writes the character range a-z, the second process writes the character range A-Z, the third process writes the character range 0-9. The parent process then starts a loop to copy data from the standard input file descriptor (that is the read end of the pipe) to the standard output file descriptor.

   The output produced by the program is not deterministic since it is possible to stop or slow down child processes, which influences what is written into the pipe.