oslaba30/ser4/mmap/mmap.c

```
/*
 * mmap.c
 *
 * Examining the virtual memory of processes.
 *
 * Operating Systems course, CSLab, ECE, NTUA
 *
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdint.h>
#include <signal.h>
#include <sys/wait.h>

#include "help.h"

#define RED     "\033[31m"
#define RESET   "\033[0m"


char *heap_private_buf;
char *heap_shared_buf;

char *file_shared_buf;

uint64_t buffer_size;

/*
 * Child process' entry point.
 */
void child(void)
{
        uint64_t pa;

        /*
         * Step 7 - Child
         */
        if (0 != raise(SIGSTOP))
                die("raise(SIGSTOP)");
        /*
         * TODO: Write your code here to complete child's part of
Step 7.
         */

        show_maps();

        /*
```

```c
if (0 != raise(SIGSTOP))
		die("raise(SIGSTOP)");
	/*
	 * TODO: Write your code here to complete child's part of
Step 8.
	 */

	get_physical_address((uint64_t)heap_private_buf);
	show_va_info((uint64_t)heap_private_buf);

	/*
	 * Step 9 - Child
	 */
	if (0 != raise(SIGSTOP))
		die("raise(SIGSTOP)");
	/*
	 * TODO: Write your code here to complete child's part of
Step 9.
	 */

	heap_private_buf = "Writing something in this private
buffer...Shhhh it's a secret! :)";
	get_physical_address((uint64_t)heap_private_buf);
	show_va_info((uint64_t)heap_private_buf);
	/*
	 * Step 10 - Child
	 */
	if (0 != raise(SIGSTOP))
		die("raise(SIGSTOP)");
	/*
	 * TODO: Write your code here to complete child's part of
Step 10.
	 */

	heap_shared_buf = "Let's all share something together! :)";

	/*
	 * Step 11 - Child
	 */
	if (0 != raise(SIGSTOP))
		die("raise(SIGSTOP)");
	/*
	 * TODO: Write your code here to complete child's part of
Step 11.
	 */

	printf("Show maps (parent)\n");
	show_maps();
	printf("Show va info (child)\n");
	show_va_info((uint64_t)heap_shared_buf);

	/*
	 * Step 12 - Child
	 */
	/*
	 * TODO: Write your code here to complete child's part of
Step 12.
```

```
         */

        munmap(heap_shared_buf, buffer_size);
        munmap(heap_private_buf, buffer_size);
        munmap(file_shared_buf, buffer_size);
}

/*
 * Parent process' entry point.
 */
void parent(pid_t child_pid)
{
        uint64_t pa;
        int status;

        /* Wait for the child to raise its first SIGSTOP. */
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");

        /*
         * Step 7: Print parent's and child's maps. What do you see?
         * Step 7 - Parent
         */

        printf(RED "\nStep 7: Print parent's and child's map.\n"
RESET);
        press_enter();

        /*
         * TODO: Write your code here to complete parent's part of
Step 7.
         */

        show_maps();

        if (-1 == kill(child_pid, SIGCONT))
                die("kill");
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");


        /*
         * Step 8: Get the physical memory address for
heap_private_buf.
         * Step 8 - Parent
         */
        printf(RED "\nStep 8: Find the physical address of the
private heap "
                "buffer (main) for both the parent and the child.\n"
RESET);
        press_enter();

        /*
         * TODO: Write your code here to complete parent's part of
Step 8.
         */
```

```
get_physical_address((uint64_t)heap_private_buf);
        show_va_info((uint64_t)heap_private_buf);

        if (-1 == kill(child_pid, SIGCONT))
                die("kill");
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");


        /*
         * Step 9: Write to heap_private_buf. What happened?
         * Step 9 - Parent
         */
        printf(RED "\nStep 9: Write to the private buffer from the
child and "
                "repeat step 8. What happened?\n" RESET);
        press_enter();

        /*
         * TODO: Write your code here to complete parent's part of
Step 9.
         */

        show_va_info((uint64_t)heap_private_buf);

        if (-1 == kill(child_pid, SIGCONT))
                die("kill");
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");


        /*
         * Step 10: Get the physical memory address for
heap_shared_buf.
         * Step 10 - Parent
         */
        printf(RED "\nStep 10: Write to the shared heap buffer (main)
from "
                "child and get the physical address for both the
parent and "
                "the child. What happened?\n" RESET);
        press_enter();

         /*
         * TODO: Write your code here to complete parent's part of
Step 10.
         */

        get_physical_address((uint64_t)heap_shared_buf);
        show_va_info((uint64_t)heap_shared_buf);

        if (-1 == kill(child_pid, SIGCONT))
                die("kill");
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");
```

```
        /*
         * Step 11: Disable writing on the shared buffer for the
child
         * (hint: mprotect(2)).
         * Step 11 - Parent
         */
        printf(RED "\nStep 11: Disable writing on the shared buffer
for the "
                "child. Verify through the maps for the parent and
the "
                "child.\n" RESET);
        press_enter();

        /*
         * TODO: Write your code here to complete parent's part of
Step 11.
         */
        mprotect(NULL, buffer_size, PROT_NONE);
        printf("Show maps (child)\n");
        show_maps();
        printf("Show va info (child)\n");
        show_va_info((uint64_t)heap_shared_buf);

        if (-1 == kill(child_pid, SIGCONT))
                die("kill");
        if (-1 == waitpid(child_pid, &status, 0))
                die("waitpid");
         /*
         * Step 12: Free all buffers for parent and child.
         * Step 12 - Parent
         */

         /*
         * TODO: Write your code here to complete parent's part of
Step 12.
         */

        munmap(heap_shared_buf, buffer_size);
        munmap(heap_private_buf, buffer_size);
        munmap(file_shared_buf, buffer_size);

}

int main(void)
{
        pid_t mypid, p;
        int fd = -1;
        uint64_t pa;
        char c;

        mypid = getpid();
        buffer_size = 1 * get_page_size();

        /*
```

```
         * Step 1: Print the virtual address space layout of this
process.
         */
        printf(RED "\nStep 1: Print the virtual address space map of
this "
                "process [%d].\n" RESET, mypid);
        press_enter();
        /*
         * TODO: Write your code here to complete Step 1.
         */
        show_maps();
         /*
         * Step 2: Use mmap to allocate a buffer of 1 page and print
the map
         * again. Store buffer in heap_private_buf.
         */
        printf(RED "\nStep 2: Use mmap(2) to allocate a private
buffer of "
                "size equal to 1 page and print the VM map again.\n"
RESET);
        press_enter();
        /*
         * TODO: Write your code here to complete Step 2.
         */
        heap_private_buf = mmap(NULL, buffer_size, PROT_READ |
PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
        if(heap_private_buf == MAP_FAILED){
                perror("mmap failed :(");
        }
        show_maps();

        /*
         * Step 3: Find the physical address of the first page of
your buffer
         * in main memory. What do you see?
         */
        printf(RED "\nStep 3: Find and print the physical address of
the "
                "buffer in main memory. What do you see?\n" RESET);
        press_enter();
        /*
         * TODO: Write your code here to complete Step 3.
         */
        pa = get_physical_address((uint64_t)heap_private_buf);

        /*
         * Step 4: Write zeros to the buffer and repeat Step 3.
         */
        printf(RED "\nStep 4: Initialize your buffer with zeros and
repeat "
                "Step 3. What happened?\n" RESET);
        press_enter();
         /*
         * TODO: Write your code here to complete Step 4.
```

```
         */
        //memset(heap_private_buf, 0, buffer_size); ---> maybe not
this
        int i;
        for(i = 0; i < (int)buffer_size; i++){
                *(heap_private_buf + i) = 0;
        }

        show_va_info((uint64_t)heap_private_buf);
        get_physical_address((uint64_t)heap_private_buf);
        /*
         * Step 5: Use mmap(2) to map file.txt (memory-mapped files)
and print
         * its content. Use file_shared_buf.
         */
        printf(RED "\nStep 5: Use mmap(2) to read and print file.txt.
Print "
                "the new mapping information that has been
created.\n" RESET);
        press_enter();
        /*
         * TODO: Write your code here to complete Step 5.
         */
        fd = open("file.txt", O_RDONLY);
        if(fd == -1) {
                perror("open failed :(");
        }

        file_shared_buf = mmap(NULL, buffer_size, PROT_READ,
MAP_SHARED, fd, 0);

        if(file_shared_buf == MAP_FAILED){
                perror("mmap failed :(");
                exit(-1);
        }
        for(i = 0; i < (int) buffer_size; i++){
                c = *(file_shared_buf + i);
                if(c != EOF)
                        putchar(c);
                else
                        break;
        }
        /*
        In case we want the new mapping
        show_maps();
        show_va_info((uint64_t)file_shared_buf);
        */
        /*
         * Step 6: Use mmap(2) to allocate a shared buffer of 1 page.
Use
         * heap_shared_buf.
         */
        printf(RED "\nStep 6: Use mmap(2) to allocate a shared buffer
of size "
```

```
                "equal to 1 page. Initialize the buffer and print the
new "
                "mapping information that has been created.\n"
RESET);
        press_enter();
        /*
         * TODO: Write your code here to complete Step 6.
         */
        heap_shared_buf = mmap(NULL, buffer_size, PROT_READ |
PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_ANONYMOUS, fd, 0);
        if(heap_shared_buf == MAP_FAILED){
                perror("mmap failed :(");
                exit(-1);
        }
        for(i = 0; i < (int)buffer_size; i++){
                *(heap_shared_buf + i) = 0;
        }
        show_maps();
        show_va_info((uint64_t)heap_shared_buf);
        p = fork();
        if (p < 0)
                die("fork");
        if (p == 0) {
                child();
                return 0;
        }

        parent(p);

        if (-1 == close(fd))
                perror("close");
        return 0;
}
```

oslaba30/ser4/sync-mmap/mandel-fork.c

```c
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdint.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

int **buff;

/***************************
 * Compile-time parameters *
 ***************************/

/*
 * Output at the terminal is is x_chars wide by y_chars long
*/
int y_chars = 50;
int x_chars = 90;

/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax,
ymin)
*/
double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */
double xstep;
double ystep;

int safe_atoi(char *s, int *val)
```

```
        char *endp;

        l = strtol(s, &endp, 10);
        if (s != endp && *endp == '\0') {
                *val = l;
                return 0;
        } else
                return -1;
}

void *safe_malloc(size_t size)
{
        void *p;

        if ((p = malloc(size)) == NULL) {
                fprintf(stderr, "Out of memory, failed to allocate
%zd bytes\n",
                        size);
                exit(1);
        }

        return p;
}

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
        /*
         * x and y traverse the complex plane.
         */
        double x, y;

        int n;
        int val;

        /* Find out the y value corresponding to this line */
        y = ymax - ystep * line;

        /* and iterate for all points on this line */
        for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {
              /* Compute the point's color value */
                val = mandel_iterations_at_point(x, y,
MANDEL_MAX_ITERATION);
                if (val > 255)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val);
                color_val[n] = val;
        }
}

/*
 * This function outputs an array of x_char color values
```

```
* to a 256-color xterm.
 */

void output_mandel_line(int fd, int color_val[])
{
        int i;
        char point ='@';
        char newline='\n';

        for (i = 0; i < x_chars; i++) {
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]);
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write
point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write
newline");
                exit(1);
        }
}

void compute_and_output_mandel_line(int fd, int line)
{
        /*
         * A temporary array, used to hold color values for the line
being drawn
         */
        int color_val[x_chars];

        compute_mandel_line(line, color_val);
        output_mandel_line(fd, color_val);
}

void usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count\n\n"
                "Exactly one argument required:\n"
                "       proc_count: The number of processes to
create.\n",
                argv0);
        exit(1);
}

/*
 * Create a shared memory area, usable by all descendants of the
calling
 * process.
 */
void *create_shared_memory_area(unsigned int numbytes)
{
        int pages;
```

```c
        void *addr;

        if (numbytes == 0) {
                fprintf(stderr, "%s: internal error: called for
numbytes == 0\n", __func__);
                exit(1);
        }

        /*
         * Determine the number of pages needed, round up the
requested number of
         * pages
         */
        pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

        /* Create a shared, anonymous mapping for this number of
pages */
        /* TODO:
                addr = mmap(...)
                */
        addr = mmap(NULL, pages, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1,0);

        return addr;
}

void destroy_shared_memory_area(void *addr, unsigned int numbytes) {
        int pages;

        if (numbytes == 0) {
                fprintf(stderr, "%s: internal error: called for
numbytes == 0\n", __func__);
                exit(1);
        }

        /*
         * Determine the number of pages needed, round up the
requested number of
         * pages
         */
        pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

        if (munmap(addr, pages * sysconf(_SC_PAGE_SIZE)) == -1) {
                perror("destroy_shared_memory_area: munmap failed");
                exit(1);
        }
}
/*For each thread--->START */
void *proc_execute(int line, int procnt)
{
        int l;
        for (l=line ; l<y_chars; l+=procnt) {
                compute_mandel_line(l, buff[l]);
        }

        return NULL;
}
```

```c
int main(int argc, char *argv[])
{
        int i, procnum, status;

        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;

        if (argc != 2)
                usage(argv[0]);
        if (safe_atoi(argv[1], &procnum) < 0 || procnum <= 0) {
                fprintf(stderr, "`%s' is not valid for
`processes'\n", argv[1]);
                exit(1);
        }

        /*Creating shared space for computing*/
        buff = create_shared_memory_area(y_chars * sizeof(int));
        for (i=0; i<y_chars; i++) {
          buff[i] = create_shared_memory_area(x_chars * sizeof(int));
        }
         /*Creating processes*/
        pid_t child_pid[procnum];
        for (i=0 ; i<procnum ; i++) {
                child_pid[i] = fork();
                if (child_pid[i] < 0) {
                        perror("error with creation of child");
                        exit(1);
                }
                if (child_pid[i] == 0) {
                        proc_execute(i, procnum);
                        exit(1);
                }
        }

        for (i=0; i<procnum ; i++) {
                child_pid[i] = wait(&status);
        }

        for (i=0; i<y_chars ; i++) {
                output_mandel_line(1, buff[i]);
        }

        destroy_shared_memory_area(buff,sizeof(int));

        reset_xterm_color(1);
        return 0;
}
```

oslaba30/ser4/sync-mmap/mandel-sem.c

```
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdint.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

sem_t *sem_table;

/***************************
 * Compile-time parameters *
 ***************************/

/*
 * Output at the terminal is is x_chars wide by y_chars long
*/
int y_chars = 50;
int x_chars = 90;

/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax,
ymin)
*/
double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */
double xstep;
double ystep;

int safe_atoi(char *s, int *val)
```

```
{
        long l;
        char *endp;

        l = strtol(s, &endp, 10);
        if (s != endp && *endp == '\0') {
                *val = l;
                return 0;
        } else
                return -1;
}

void *safe_malloc(size_t size)
{
        void *p;

        if ((p = malloc(size)) == NULL) {
                fprintf(stderr, "Out of memory, failed to allocate
%zd bytes\n",
                        size);
                exit(1);
        }

        return p;
}

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */

void compute_mandel_line(int line, int color_val[])
{
        /*
         * x and y traverse the complex plane.
         */
        double x, y;

        int n;
        int val;

        /* Find out the y value corresponding to this line */
        y = ymax - ystep * line;

        /* and iterate for all points on this line */
        for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

                /* Compute the point's color value */
                val = mandel_iterations_at_point(x, y,
MANDEL_MAX_ITERATION);
                if (val > 255)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val);
                color_val[n] = val;
        }
```

```
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
{
        int i;
        char point ='@';
        char newline='\n';

        for (i = 0; i < x_chars; i++) {
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]);
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write
point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write
newline");
                exit(1);
        }
}
void compute_and_output_mandel_line(int fd, int line)
{
        /*
         * A temporary array, used to hold color values for the line
being drawn
         */
        int color_val[x_chars];

        compute_mandel_line(line, color_val);
        output_mandel_line(fd, color_val);
}

void usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count\n\n"
                "Exactly one argument required:\n"
                "       proc_count: The number of processes to
create.\n",
                argv0);
        exit(1);
}

/*
 * Create a shared memory area, usable by all descendants of the
calling
 * process.
 */
void *create_shared_memory_area(unsigned int numbytes)
```

```
{
	int pages;
	void *addr;

	if (numbytes == 0) {
		fprintf(stderr, "%s: internal error: called for
numbytes == 0\n", __func__);
		exit(1);
	}

	/*
	 * Determine the number of pages needed, round up the
requested number of
	 * pages
	 */
	pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

	/* Create a shared, anonymous mapping for this number of
pages */
	/* TODO:
		addr = mmap(...)
	*/
	addr = mmap(NULL, pages, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1,0);

	return addr;
}

void destroy_shared_memory_area(void *addr, unsigned int numbytes) {
	int pages;

	if (numbytes == 0) {
		fprintf(stderr, "%s: internal error: called for
numbytes == 0\n", __func__);
		exit(1);
	}

	/*
	 * Determine the number of pages needed, round up the
requested number of
	 * pages
	 */
	pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

	if (munmap(addr, pages * sysconf(_SC_PAGE_SIZE)) == -1) {
		perror("destroy_shared_memory_area: munmap failed");
		exit(1);
	}
}

/*For each process--->START */
void *proc_execute(int line, int procn)
{
	int l;
	int color_val[x_chars];
	for (l=line ; l<y_chars; l+=procn) {
		compute_mandel_line(l, color_val);
```

```
                sem_wait(&sem_table[line]);
                output_mandel_line(1, color_val);
                sem_post(&sem_table[(l+1) % procn]);
        }

        return NULL;
}

int main(int argc, char *argv[])
{
        int i, procnum, status;

        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;

        if (argc != 2)
                usage(argv[0]);
        if (safe_atoi(argv[1], &procnum) < 0 || procnum <= 0) {
                fprintf(stderr, "`%s' is not valid for
`processes'\n", argv[1]);
                exit(1);
        }

        /*Creating one semaphore/process*/
        sem_table = create_shared_memory_area(procnum *
sizeof(sem_t));
        for (i=0; i<procnum; i++) {
                sem_init(&sem_table[i],1,0); //sem_init
(pshared=1:shared between procs)
        }

        /*first semaphore initialization*/
        sem_post(&sem_table[0]);

        /*Creating processes*/
        pid_t child_pid[procnum];
        for (i=0 ; i<procnum ; i++) {
                child_pid[i] = fork();
                if (child_pid[i] < 0) {
                        perror("error with creation of child");
                        exit(1);
                }
                if (child_pid[i] == 0) {
                        proc_execute(i, procnum);
                        exit(1);
                }
        }
        for (i=0; i<procnum ; i++) {
                child_pid[i] = wait(&status);
        }

        for (i=0; i<procnum ; i++) {
              sem_destroy(&sem_table[i]);
      }

        destroy_shared_memory_area(sem_table, procnum *
sizeof(sem_t));
```

```
        reset_xterm_color(1);
        return 0;
}
```

Ερωτήσεις 1.2.1

1. Ποια από τις δύο παραλληλοποιημένες υλοποιήσεις (threads vs processes) περιμένετε να έχει καλύτερη επίδοση και γιατί; Πώς επηρεάζει την επίδοση της υλοποίησης με διεργασίες το γεγονός ότι τα semaphores βρίσκονται σε διαμοιραζόμενη μνήμη μεταξύ διεργασιών;

Η υλοποίηση με threads έχει υψηλότερη επίδοση, διότι τα νήματα έχουν εξαρχής διαμοιραζόμενη μνήμη. Αντίθετα, στην υλοποίηση με διεργασίες, πρέπει μέσω system calls(mmap), να δεσμεύσουμε χώρο για την κοινή μνήμη μεταξύ των διεργασιών και αυτό το κομμάτι κώδικα "κοστίζει" περισσότερο. Επομένως, είναι προτιμότερη η υλοποίηση με νήματα. Ωστόσο από τη στιγμή που δεσμεύεται η απαιτούμενη μνήμη για τις διεργασίες, η διαδικασία υλοποίησης με processes και παράλληλη επεξεργασία για τους σημαφόρους δε διαφέρει με την υλοποίηση με threads.

Ερωτήσεις 1.2.2

1. Με ποιο τρόπο και σε ποιο σημείο επιτυγχάνεται ο συγχρονισμός σε αυτή την υλοποίηση; Πώς θα επηρεαζόταν το σχήμα συγχρονισμού αν ο buffer είχε διαστάσεις NPROCS x x_chars;

Ο συγχρονισμός επιτυγχάνεται με έναν κοινό, για όλες τις διεργασίες, πίνακα σημαφόρων, ο οποίος αρχικοποιείται με τη **sem_init()**(της οποίας η δεύτερη παράμετρος είναι ίση με 1 ώστε να μοιράζονται οι σημαφόροι σε όλες τις διεργασίες). Όπως έχουμε υλοποιήσει τον πίνακα, γίνονται πρώτα οι υπολογισμοί οι οποίοι μπαίνουν στον πίνακα στις σωστές θέσεις και στη συνέχεια, μετά το τέλος των υπολογισμών, εκτυπώνεται ο πίνακας με τη σωστή σειρά. Αν ο buffer είχε διαστάσεις NPROCS x x_chars, θα αναθέταμε μία γραμμή ανά διεργασία, οπότε πάλι θα πρέπει να περιμένουμε να τυπωθεί η σειρά του πίνακα που αντιστοιχεί στην κάθε διεργασία, ώστε να μπορέσει να ξαναγράψει στην ίδια γραμμή του πίνακα η ίδια διεργασία με τα νέα υπολογισμένα δεδομένα. Συνεπώς, δεν μπορούν να γίνονται οι υπολογισμοί από τις διεργασίες πριν τυπωθεί ο πίνακας γιατί θα χαθούν τα αρχικά στοιχεία και θα υπάρχει και επιπλέον αναμονή.