ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Λειτουργικά Συστήματα

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2020-2021

Ονοματεπώνυμο	Αριθμός Μητρώου
Γουλόπουλος Δημήτριος	03107627
Σιαφάκας Ξενοφών	03115753

oslaba116

Άσκηση 4: Μηχανισμοί Εικονικής Μνήμης

1.1 Κλήσεις συστήματος και βασικοί μηχανισμοί του ΛΣ για τη διαχείριση της εικονικής μνήμης (Virtual Memory – VM)

Πηγαίος Κώδικας: mmap.c

```
#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)
{
        int i;
        uint64_t pa, va;  // pa: physical address, va: virtual address
        /*  ***************** Step 7 - Child *****************  */
        if (0 != raise(SIGSTOP)) die("raise(SIGSTOP)");
        /* TODO  */
        show_maps();
        /* TODO  */
        /*  ***************** Step 8 - Child *****************  */
        if (0 != raise(SIGSTOP)) die("raise(SIGSTOP)");
        /* TODO  */
        va = (uint64_t) heap_private_buf;
        printf("virtual address (on child proc) is 0x%lx\n", va);
```

```
        pa = get_physical_address(va);
        printf("Physical address (on child proc) is 0x%lx\n", pa);
        /* TODO  */
        /*  ***************** Step 9 - Child *****************  */
        if (0 != raise(SIGSTOP)) die("raise(SIGSTOP)");
        /* TODO  */
        for (i = 0; i < (int) buffer_size; i++)
              heap_private_buf[i] = 1 ;
        va = (uint64_t) heap_private_buf;
        printf("virtual address (on child proc) is 0x%lx\n", va);
        pa = get_physical_address(va);
        printf("Physical address (on child proc) is 0x%lx\n", pa);
        /* TODO  */
        /*  ***************** Step 10 - Child *****************  */
        if (0 != raise(SIGSTOP)) die("raise(SIGSTOP)");
        /* TODO  */
        for (i = 0; i < (int) buffer_size; i++)
             heap_shared_buf[i] = 1 ;
        va = (uint64_t) heap_shared_buf;
        printf("virtual address (on child proc) is 0x%lx\n", va);
        pa = get_physical_address(va);
        printf("Physical address (on child proc) is 0x%lx\n", pa);
        /* TODO  */
        /*  ***************** Step 11 - Child *****************  */
        if (0 != raise(SIGSTOP)) die("raise(SIGSTOP)");
        /* TODO  */
         mprotect(heap_shared_buf, buffer_size, PROT_READ);
        show_maps();
        printf(RED "\nDifference between parent and child: parent can read and write, child only reads:" RESET);
        printf(RED "\nparent: 7f0c2f499000-7f0c2f49a000 rw-s 00000000 00:04 2059970                   /dev/zero (deleted)\n" RESET);
        printf(RED "child:  7f0c2f499000-7f0c2f49a000 r--s 00000000 00:04 2059970                  /dev/zero (deleted)\n" RESET);
        /* TODO  */
        /*  ***************** Step 12 - Child *****************  */
        /* TODO  */
        /* TODO  */
}
```

```
/*
 * Parent process' entry point.
 */
void parent(pid_t child_pid)
{
        uint64_t pa, va;
        int status;
        /* Wait for the child to raise its first SIGSTOP. */
        if (-1 == waitpid(child_pid, &status, WUNTRACED))
                die("waitpid");
        /*  ***************** Step 7 - Parent *****************  */
        /* Step 7: Print parent's and child's maps. What do you see? */
        printf(RED "\nStep 7: Print parent's and child's map.\n" RESET);
        press_enter();
        /* TODO  */
        show_maps();
        /* TODO  */
        if (-1 == kill(child_pid, SIGCONT)) die("kill"); if (-1 == waitpid(child_pid, &status, WUNTRACED)) die("waitpid");
        /*  ***************** Step 8 - Parent *****************  */
        /*  Step 8: Get the physical memory address for heap_private_buf.  */
        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  */
        va = (uint64_t) heap_private_buf;
        printf("virtual address (on parent proc) is 0x%lx\n", va);
        pa = get_physical_address(va);
        printf("Physical address (on parent proc) is 0x%lx\n", pa);
        /* TODO  */
        if (-1 == kill(child_pid, SIGCONT)) die("kill"); if (-1 == waitpid(child_pid, &status, WUNTRACED)) die("waitpid");
        /*  ***************** Step 9 - Parent *****************  */
        /*  Step 9: Write to heap_private_buf. What happened?  */
        printf(RED "\nStep 9: Write to the private buffer from the child and repeat step 8. What happened?\n" RESET);
        press_enter();
        /* TODO  */
        va = (uint64_t) heap_private_buf;
        printf("virtual address (on parent proc) is 0x%lx\n", va);
        pa = get_physical_address(va);
        printf("Physical address (on parent proc) is 0x%lx\n", pa);
        /* TODO  */
        if (-1 == kill(child_pid, SIGCONT)) die("kill"); if (-1 == waitpid(child_pid, &status, WUNTRACED)) die("waitpid");
```

```
        /*  ***************** Step 10 - Parent *****************  */

        /*  Step 10: Get the physical memory address for heap_shared_buf.  */

        printf(RED "\nStep 10: Write to the shared heap buffer (main) from child\n" RESET);

         printf(RED "and get the physical address for both the parent and the child. What happened?\n" RESET);

        press_enter();

        /* TODO  */

        va = (uint64_t) heap_shared_buf;

        printf("virtual address (on parent proc) is 0x%lx\n", va);

        pa = get_physical_address(va);

        printf("Physical address (on parent proc) is 0x%lx\n", pa);

        /* TODO  */

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

        /*  ***************** Step 11 - Parent *****************  */

        /*  Step 11: Disable writing on the shared buffer for the child * (hint: mprotect(2)).  */

        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  */

        show_maps();

        /* TODO  */

        if (-1 == kill(child_pid, SIGCONT)) die("kill"); if (-1 == waitpid(child_pid, &status, 0)) die("waitpid");

        /*  ***************** Step 12 - Parent *****************  */

        /*  Step 12: Free all buffers for parent and child.  */


        /* TODO  */

         munmap(heap_private_buf, buffer_size);

         munmap(file_shared_buf, buffer_size);

         munmap(heap_shared_buf, buffer_size);

        printf(RED "\nStep 12: All buffers were freed. Program terminates\n" RESET);

        /* TODO  */

}
```

```
int main(void)
{
        int i;
        pid_t mypid, p;
        int fd = -1, newfd = -1;
        uint64_t pa, va;
        mypid = getpid();
        buffer_size = 1 * get_page_size();

        /*  ***************** Step 1 - main() ******************  */
        /*  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  */
        show_maps();
        /* TODO  */




        /*  ***************** Step 2 - main() ******************  */
        /*  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  */
        heap_private_buf = mmap(NULL, buffer_size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, fd, 0);
        if (heap_private_buf == MAP_FAILED) printf("MMAP FAILED\n");
        show_maps();
        va = (uint64_t) heap_private_buf;
         printf("private buffer info: \n");
        show_va_info(va);
        /* TODO  */
        /*  ***************** Step 3 - main() ******************  */
        /*  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  */
        printf("virtual address (on main()) is 0x%lx\n", va);
        pa = get_physical_address(va);    /* get_physical_address() prints a message: VA[0x7f48fcdd5000] is not mapped */
        /* TODO  */
```

```
/*  ***************** Step 4 - main() ******************  */
/*  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  */
for (i = 0; i < (int) buffer_size; ++i) heap_private_buf[i]=0;
printf("virtual address (on main()) is 0x%lx\n", va);
pa = get_physical_address(va);
printf("Physical address for VA 0x%lx is 0x%lx\n", va, pa);   /* now VA[0x7f48fcdd5000] is in memory ! */
/* TODO  */




/*  ***************** Step 5 - main() ******************  */
/*  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  */
fd = open("file.txt", O_RDONLY);
if(fd < 0) perror("Cannot open file.txt\n");
file_shared_buf = mmap(NULL, buffer_size, PROT_READ, MAP_SHARED, fd, 0);
if (file_shared_buf == MAP_FAILED) printf("MMAP FAILED\n");
fprintf(stdout, file_shared_buf);
va = (uint64_t) file_shared_buf;
 printf("private buffer for file info: \n");
show_va_info(va);
pa = get_physical_address(va);
printf("Physical address is 0x%lx\n", pa);
/* TODO  */
```

```
        /*  ***************** Step 6 - main() ******************  */
        /*  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." RESET);
       printf(RED "\nInitialize the buffer and print the new mapping information that has been created.\n" RESET);
        press_enter();

        /* TODO  */
        heap_shared_buf = mmap(NULL, buffer_size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, newfd, 0);
        if (heap_shared_buf == MAP_FAILED) printf("MMAP FAILED\n");
        for (i = 0; i < (int) buffer_size; ++i) heap_shared_buf[i]=0;
        va = (uint64_t) heap_shared_buf;
         printf("shared buffer info: \n");
        show_va_info(va);
        pa = get_physical_address(va);
        printf("Physical address is 0x%lx\n", pa);
        /* TODO  */

        /*Fork is called*/
        p = fork();
        if (p < 0)  die("fork");
        if (p == 0)  /* i am the child */
         {
                child();
                return 0;
        }

        parent(p);   /* parent knows p, child's PID */

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

Σας δίνεται ο σκελετός προγράμματος mmap.c τον οποίο και πρέπει να συμπληρώσετε ακολουθώντας τα εξής βήματα:

1. Τυπώστε το χάρτη της εικονικής μνήμης της τρέχουσας διεργασίας.

```
Step 1: Print the virtual address space map of this process [13650].

Virtual Memory Map of process [13650]:
00400000-00403000 r-xp 00000000 00:21 20454573                           /home/oslab/oslaba116/askisi--4/mmap
00602000-00603000 rw-p 00002000 00:21 20454573                           /home/oslab/oslaba116/askisi--4/mmap
00b24000-00b45000 rw-p 00000000 00:00 0                                  [heap]
7fbd9ebcc000-7fbd9ed6d000 r-xp 00000000 08:01 6032227                    /lib/x86_64-linux-gnu/libc-2.19.so
7fbd9ed6d000-7fbd9ef6d000 ---p 001a1000 08:01 6032227                    /lib/x86_64-linux-gnu/libc-2.19.so
7fbd9ef6d000-7fbd9ef71000 r--p 001a1000 08:01 6032227                    /lib/x86_64-linux-gnu/libc-2.19.so
7fbd9ef71000-7fbd9ef73000 rw-p 001a5000 08:01 6032227                    /lib/x86_64-linux-gnu/libc-2.19.so
7fbd9ef73000-7fbd9ef77000 rw-p 00000000 00:00 0
7fbd9ef77000-7fbd9ef98000 r-xp 00000000 08:01 6032224                    /lib/x86_64-linux-gnu/ld-2.19.so
7fbd9f18a000-7fbd9f18d000 rw-p 00000000 00:00 0
7fbd9f192000-7fbd9f197000 rw-p 00000000 00:00 0
7fbd9f197000-7fbd9f198000 r--p 00020000 08:01 6032224                    /lib/x86_64-linux-gnu/ld-2.19.so
7fbd9f198000-7fbd9f199000 rw-p 00021000 08:01 6032224                    /lib/x86_64-linux-gnu/ld-2.19.so
7fbd9f199000-7fbd9f19a000 rw-p 00000000 00:00 0
7ffec00c0000-7ffec00e1000 rw-p 00000000 00:00 0                          [stack]
7ffec0102000-7ffec0105000 r--p 00000000 00:00 0                          [vvar]
7ffec0105000-7ffec0107000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

2. Με την κλήση συστήματος mmap() δεσμεύστε buffer (προσωρινή μνήμη) μεγέθους μίας σελίδας (page) και τυπώστε ξανά το χάρτη. Εντοπίστε στον χάρτη μνήμης τον χώρο εικονικών διευθύνσεων που δεσμεύσατε.

πριν:

7fbd9f**192000**-7fbd9f197000 rw-p 00000000 00:00 0

μετά:

7fbd9f**191000**-7fbd9f197000 rw-p 00000000 00:00 0

έχει δεσμευτεί μια σελίδα (...191000-...192000)

3. Προσπαθήστε να βρείτε και να τυπώσετε τη φυσική διεύθυνση μνήμης στην οποία απεικονίζεται η εικονική διεύθυνση του buffer (τη διεύθυνση όπου βρίσκεται αποθηκευμένος στη φυσική κύρια μνήμη). Τι παρατηρείτε και γιατί;

virtual address (on main()) is 0x7fbd9f192000

VA[0x7fbd9f192000] is not mapped; no physical memory allocated.

ο buffer δε βρίσκεται στη μνήμη

4. Γεμίστε με μηδενικά τον buffer και επαναλάβετε το Βήμα 3. Ποια αλλαγή παρατηρείτε;

ο buffer βρίσκεται στη μνήμη

virtual address (on main()) is 0x7fbd9f192000

Physical address for VA 0x7fbd9f192000 is 0x1115dc000

επομένως το λειτουργικό εφαρμόζει on-demand paging

5. Χρησιμοποιείστε την mmap() για να απεικονίσετε (memory map) το αρχείο file.txt στον χώρο διευθύνσεων της διεργασίας σας και να τυπώσετε το περιεχόμενό του. Εντοπίστε τη νέα απεικόνιση (mapping) στον χάρτη μνήμης.

private buffer for file info:

7fbd9f191000-7fbd9f192000 r--s 00000000 00:21 20454553
/home/oslab/oslaba116/askisi--4/file.txt

Physical address is 0x474d8000

6. Χρησιμοποιείστε την mmap() για να δεσμεύσετε έναν νέο buffer, διαμοιραζόμενο (shared) αυτή τη φορά μεταξύ διεργασιών με μέγεθος μια σελίδας. Εντοπίστε τη νέα απεικόνιση (mapping) στο χάρτη μνήμης.

shared buffer info:

7fbd9f190000-7fbd9f191000 rw-s 00000000 00:04 2310891
/dev/zero (deleted)

Physical address is 0x128b62000

Στο σημείο αυτό καλείται η συνάρτηση fork() και δημιουργείται μια νέα διεργασία.

7. Τυπώστε τον χάρτη της εικονικής μνήμης της διεργασίας πατέρα και της διεργασίας παιδιού. Τι παρατηρείτε?

Το παιδί είναι κλώνος του πατέρα και έχουν τις ίδιες εικονικές διευθύνσεις.

8. Βρείτε και τυπώστε τη φυσική διεύθυνση στη κύρια μνήμη του private buffer (Βήμα 3) για τις διεργασίες πατέρα και παιδί. Τι συμβαίνει αμέσως μετά το fork?

Το παιδί είναι κλώνος του πατέρα και έχουν τις ίδιες διευθύνσεις, τόσο εικονικές όσο και φυσικές.

virtual address (on parent proc) is 0x7fbd9f192000

Physical address (on parent proc) is 0x1115dc000

virtual address (on child proc) is 0x7fbd9f192000

Physical address (on child proc) is 0x1115dc000

9. Γράψτε στον private buffer από τη διεργασία παιδί και επαναλάβετε το Βήμα 8. Τι αλλάζει και γιατί?

virtual address (on parent proc) is 0x7fbd9f192000

Physical address (on parent proc) is 0x1115dc000

virtual address (on child proc) is 0x7fbd9f192000

Physical address (on child proc) is 0x103fc6000

Οι φυσικές διευθύνσεις είναι τώρα διαφορετικές (φαινόμενο copy-on-write).

10. Γράψτε στον shared buffer (Βήμα 6) από τη διεργασία παιδί και τυπώστε τη φυσική του διεύθυνση για τις διεργασίες πατέρα και παιδί. Τι παρατηρείτε σε σύγκριση με τον private buffer?

virtual address (on parent proc) is 0x7fbd9f190000

Physical address (on parent proc) is 0x128b62000

virtual address (on child proc) is 0x7fbd9f190000

Physical address (on child proc) is 0x128b62000

Το παιδί και ο πατέρας έχουν τις ίδιες διευθύνσεις για τον shared buffer, τόσο εικονικές όσο και φυσικές. Λογικό, αφού είναι shared buffer !

11. Απαγορεύστε τις εγγραφές στον shared buffer για τη διεργασία παιδί. Εντοπίστε και τυπώστε την απεικόνιση του shared buffer στο χάρτη μνήμης των δύο διεργασιών για να επιβεβαιώσετε την απαγόρευση.

Difference between parent and child: parent can read and write, child only reads:

parent: 7f0c2f499000-7f0c2f49a000 rw-s 00000000 00:04 2059970 /dev/zero (deleted)

child: 7f0c2f499000-7f0c2f49a000 r--s 00000000 00:04 2059970 /dev/zero (deleted)

12. Αποδεσμεύστε όλους τους buffers στις δύο διεργασίες.

Μέσω της συνάρτησης munmap.

1.2 Παράλληλος υπολογισμός Mandelbrot με διεργασίες αντί για νήματα

Πηγαίος Κώδικας: mandel-fork.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mman.h>

/*TODO header file for m(un)map*/

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000



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

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

double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

double xstep;
double ystep;
```

```
/*
 * 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, int i, int n, sem_t *nsem)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        int color_val[x_chars], value;

        compute_mandel_line(line, color_val);

        /* TODO */
        /* synchronize processes with a circular method*/
        sem_getvalue(&nsem[i], &value);
        while (value != 1) { sem_getvalue(&nsem[i], &value); }
        sem_wait(&nsem[i]);
        /* TODO */

        output_mandel_line(fd, color_val);

        /* TODO */
        /* synchronize processes with a circular method*/
        if (i + 1 < n) { sem_post(&nsem[i + 1]); }
        else           { sem_post(&nsem[0]); }
        /* TODO */
}
```

```
/*
 * 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;

         /* TODO */
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_SHARED | MAP_ANONYMOUS;
         /* TODO */

        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;
         pages = pages ;   /* shut up, compiler! */

        /* Create a shared, anonymous mapping for this number of pages */

         /* TODO */
        addr = mmap(NULL, numbytes, protection, visibility, -1, 0);
         /* TODO */

        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);
        }
}
```

```
void drawmandelbrot(int i, int n, sem_t *nsem)    /* draw the Mandelbrot Set, one line at a time */
{
        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;
        int line;
        for (line = i; line < y_chars; line += n) { compute_and_output_mandel_line(1, line, i, n, nsem); }
        reset_xterm_color(1);
}

void sig_handler(int signum)
{
        printf("\nCaught SIGINT\n");
        set_xterm_color(1, 7);
        exit(1);
}
```

```
int main(int argc, char *argv[])
{
        sem_t *nsem;
        int i, n=1, j;
        signal(SIGINT, sig_handler);  /* handle signal */

        /* gets the number of processes from stdin */
        if (argc < 2) { printf("Usage: ./mandel [NPROCS]\nSwitching to default. Using 1 process.\n"); }
        else { n = atoi(argv[1]); }

        /* initialize semaphores */
        nsem = create_shared_memory_area(sizeof(sem_t) * n);
        sem_init(&nsem[0], 0, 1);
        for (j = 1; j < n; j++) sem_init(&nsem[j], 0, 0);

        /* create processes and print mandelbrot set */
        pid_t pid[n];
        int status[n];
        for (i = 0; i < n; i++)
        {
                pid[i] = fork();
                if (pid[i] < 0) { perror("main: fork"); exit(1); }
                else if (pid[i] == 0) { drawmandelbrot(i, n, nsem); exit(1); }
        }

        /* wait till processes end */
        for (i = 0; i < n; i++) { pid[i] = wait(&status[i]); }
        destroy_shared_memory_area(nsem, sizeof(sem_t) * n);
        return 0;
}
```

Ερωτήσεις:

1. Ποια από τις δύο παραλληλοποιημένες υλοποιήσεις (threads vs processes) περιμένετε να έχει καλύτερη επίδοση και γιατί;

(vs processes)

Περιμένουμε ότι τα threads θα έχουν καλύτερη επίδοση, καθώς:

- Έχουν μικρότερο έργο για τη δημιουργία και τον τερματισμό έναντι των processes, επειδή απαιτείται πολύ λίγη αντιγραφή μνήμης (μόνο η στοίβα νήματος).
- Ταχύτερη εναλλαγή: σε πολλές περιπτώσεις, είναι πιο γρήγορο για ένα λειτουργικό σύστημα να εναλλάσσεται μεταξύ νημάτων από ότι εναλλάσσεται μεταξύ διαφορετικών διεργασιών. Οι caches της CPU και το πλαίσιο προγράμματος μπορούν να διατηρηθούν μεταξύ των νημάτων, αντί να φορτωθούν ξανά όπως στην περίπτωση της αλλαγής CPU σε διαφορετική διεργασία.
- Κοινή χρήση δεδομένων με άλλα νήματα: για εργασίες που απαιτούν κοινή χρήση μεγάλου όγκου δεδομένων, το γεγονός ότι όλα τα νήματα μοιράζονται το σύνολο μνήμης είναι πολύ ευεργετικό. Το να μην έχετε ξεχωριστά αντίγραφα σημαίνει ότι διαφορετικά νήματα μπορούν να διαβάσουν και να τροποποιήσουν εύκολα μια κοινή μνήμη.

Πώς επηρεάζει την επίδοση της υλοποίησης με διεργασίες το γεγονός ότι τα semaphores βρίσκονται σε διαμοιραζόμενη μνήμη μεταξύ διεργασιών;

Επηρεάζει θετικά, καθώς η ενεργοποίηση και η απενεργοποίηση των νημάτων γίνεται χωρίς καμμιά καθυστέρηση.