



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

Άσκηση 4: Μηχανισμοί Εικονικής Μνήμης

ΟΜΑΔΑ: oslab100

Ασπρογέρακας Ιωάννης (03118942)

Κανατάς Άγγελος-Νικόλαος (03119169)

Σε αυτήν την εργαστηριακή άσκηση, θα ασχοληθούμε με την μελέτη του μηχανισμού της εικονική μνήμης (**Virtual Memory – VM**) και με την χρήση της για διαδιεργασιακή επικοινωνία. Συγκεκριμένα, στο πρώτο μέρος της άσκησης θα πειραματιστούμε με κλήσεις συστήματος για να μελετήσουμε βασικούς μηχανισμούς του ΛΣ (**1.1**). Στο δεύτερο μέρος θα χρησιμοποιήσουμε την εικονική μνήμη για τον διαμοιρασμό πόρων μεταξύ διεργασιών (*διαδιεργασιακή επικοινωνία πάνω σε κοινή μνήμη*), με σκοπό τον παράλληλο υπολογισμό του συνόλου Mandelbrot (**1.2**) (βλ. Άσκηση 3 όπου το υλοποιήσαμε με συγχρονισμό νημάτων).

Άσκηση 1.1 (Κλήσεις συστήματος και βασικοί μηχανισμοί του ΛΣ για την διαχείριση της εικονικής μνήμης)

Παρακάτω φαίνεται ο **πηγαίος κώδικας (source code)** για την άσκηση:

```
/*
 * Examining the virtual memory of processes.
 */
#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 "../helpers/help.h"

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

char *heap_private_buf; //heap buffers
char *heap_shared_buf;
char *file_shared_buf; //memory-mapped file

uint64_t buffer_size; //buffer size in bytes

/*
 * Child process' entry point.
 */
void child(void)
{
    /*
     * Step 7 - Child
     */
    if(0!=raise(SIGSTOP)) //for sync
        die("raise(SIGSTOP)");

    printf("VM map of child process:\n");
    show_maps();

    /*
     * Step 8 - Child
     */
    if(0!=raise(SIGSTOP))
        die("raise(SIGSTOP)");

    printf("Physical Address of private buffer requested by child: %ld\n",
        get_physical_address((uint64_t)heap_private_buf));

    printf("VA info of private buffer in child: ");
    show_va_info((uint64_t) heap_private_buf);

    /*
     * Step 9 - Child
     */
}
```

```

    if(0!=raise(SIGSTOP))
        die("raise(SIGSTOP)");

    int j;
    for(j=0; j<(int)buffer_size; j++) {
        heap_private_buf[j]=0;
    }
    printf("VA info of private buffer in child: ");
    show_va_info((uint64_t) heap_private_buf);

    printf("Physical Address of private buffer requested by child: %ld\n",
        get_physical_address((uint64_t)heap_private_buf));

/*
 * Step 10 - Child
 */
    if(0!=raise(SIGSTOP))
        die("raise(SIGSTOP)");

    for(j=0; j<(int)buffer_size; j++) {
        heap_shared_buf[j]=0;
    }

    printf("VA info of shared buffer in child: ");
    show_va_info((uint64_t) heap_shared_buf);

    printf("Physical Address of shared buffer requested by child: %ld\n",
        get_physical_address((uint64_t)heap_shared_buf));

/*
 * Step 11 - Child
 */
    if(0!=raise(SIGSTOP))
        die("raise(SIGSTOP)");

    mprotect(heap_shared_buf, buffer_size, PROT_READ);
    printf("VM map of child:\n");
    show_maps();
    show_va_info((uint64_t)heap_shared_buf);

/*
 * Step 12 - Child
 */
    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)
{
    int status;

    /*
     * Father executes first, then the child
     */

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

printf("VM map of parent process:\n");
show_maps();

/*
 * For parent and child execution synchronization
 */
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();

printf("Physical Address of private buffer requested by parent: %ld\n",
        get_physical_address((uint64_t)heap_private_buf));

printf("VA info of private buffer in parent: ");
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();

printf("VA info of private buffer in parent: ");
show_va_info((uint64_t) heap_private_buf);

printf("Physical Address of private buffer requested by parent: %ld\n",
        get_physical_address((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();

```

```

printf("VA info of shared buffer in parent: ");
show_va_info((uint64_t) heap_shared_buf);

printf("Physical Address of shared buffer requested by parent: %ld\n",
        get_physical_address((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();

printf("VM map of parent");
show_maps();
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
 */
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;

    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();

    show_maps(); //virtual memory map

    /*
     * 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();

heap_private_buf=mmap(NULL, buffer_size, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS, fd,0); //fd=-1

if (heap_private_buf==MAP_FAILED)
    die("mmap");

show_maps();
show_va_info((uint64_t)heap_private_buf);

/*
 * 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();

printf("Physical address: %ld\n",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();

int i;
for (i=0; i<(int)buffer_size; i++) { //zero-ing
    heap_private_buf[i]=0;
}

printf("Physical address: %ld\n",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();

fd=open("file.txt",O_RDONLY);
if(fd==-1)
    die("open");

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

if(file_shared_buf==MAP_FAILED)
    die("mmap");

char c;
for (i=0; i<(int)buffer_size; i++) { //print file contents
    c=file_shared_buf[i];
    if (c!= EOF)
        putchar(c);
    else break;
}

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();

heap_shared_buf=mmap(NULL, buffer_size, PROT_READ| PROT_WRITE,
                     MAP_SHARED| MAP_ANONYMOUS, -1, 0);

if(heap_private_buf==MAP_FAILED)
    die("mmap");

for(i=0; i<(int)buffer_size; i++) {
    heap_shared_buf[i]=i;
}

show_maps();
show_va_info((uint64_t)heap_shared_buf);

//forking

p=fork();
if(p<0)
    die("fork");
if (p==0) {
    child(); //i'm the child
    return 0;
}

//i'm the parent
parent(p);

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

```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Το μεταγλωττίζουμε, κάνουμε το κατάλληλο linking και το εκτελούμε:

1. Αρχικά, τυπώνουμε το χάρτη της εικονικής μνήμης της τρέχουσας διεργασίας:

```

oslab100@os-node1:~/ex4/1.1$ ./mmap
Step 1: Print the virtual address space map of this process [22930].
mmap_min_addr
Virtual Memory Map of process [22930]:
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
018d8000-018f9000 rw-p 00000000 00:00 0 [heap]
7fc181cb4000-7fc181e55000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc181e55000-7fc182055000 --p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182055000-7fc182059000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182059000-7fc18205b000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc18205b000-7fc18205f000 rw-p 00000000 00:00 0
7fc18205f000-7fc182080000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182272000-7fc182275000 rw-p 00000000 00:00 0
7fc18227a000-7fc18227f000 rw-p 00000000 00:00 0
7fc18227f000-7fc182280000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182280000-7fc182281000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182281000-7fc182282000 rw-p 00000000 00:00 0
7ffcc7c4d000-7ffcc7c6e000 rw-p 00000000 00:00 0 [stack]
7ffcc7dec000-7ffcc7def000 r--p 00000000 00:00 0 [vvar]
7ffcc7def000-7ffcc7df1000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
-----

```

2. Δεσμεύουμε buffer μεγέθους μίας σελίδας (page) με την χρήση της `mmap()` (απεικόνιση *σωρού*) και τυπώνουμε ξανά το χάρτη:

```
Step 2: Use mmap(2) to allocate a private buffer of size equal to 1 page and print the VM map again.

Virtual Memory Map of process [22930]:
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
018d8000-018f9000 rw-p 00000000 00:00 0 [heap]
7fc181cb4000-7fc181e55000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc181e55000-7fc182055000 ---p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182055000-7fc182059000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182059000-7fc18205b000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc18205b000-7fc18205f000 rw-p 00000000 00:00 0
7fc18205f000-7fc182080000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182272000-7fc182275000 rw-p 00000000 00:00 0
7fc182279000-7fc18227f000 rw-p 00000000 00:00 0
7fc18227f000-7fc182280000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182280000-7fc182281000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182281000-7fc182282000 rw-p 00000000 00:00 0
7ffc7c4d000-7ffc7c6e000 rw-p 00000000 00:00 0 [stack]
7ffc7dec000-7ffc7def000 r--p 00000000 00:00 0 [vvar]
7ffc7def000-7ffc7df1000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
-----
7fc182279000-7fc18227f000 rw-p 00000000 00:00 0
```

3. Προσπαθούμε να βρούμε και να τυπώσουμε τη *φυσική διεύθυνση μνήμης* στην οποία απεικονίζεται η εικονική διεύθυνση του buffer:

```
Step 3: Find and print the physical address of the buffer in main memory. What do you see?

VA[0x7fc18227a000] is not mapped; no physical memory allocated.
Physical address: 0
```

Παρατηρούμε, λοιπόν, ότι ενώ έχει δεσμευθεί η μνήμη όπως φαίνεται και στον χάρτη μνήμης δεν έχει γίνει ακόμα η απεικόνιση της εικονικής μνήμης στην φυσική. Αυτό οφείλεται στο ότι η φυσική μνήμη δεσμεύεται **on demand** από το ΛΣ όταν πάει να προσπελαστεί. Έτσι, όταν πάμε να προσπελάσουμε εικονική μνήμη όπου δεν έχει απεικονιστεί ακόμα (οπότε και δεν υπάρχει απεικόνιση στον πίνακα σελίδων) τότε θα γίνει **page fault** και αφού διαπιστωθεί ότι η εικονική μνήμη βρίσκεται στον χάρτη μνήμης τότε το ΛΣ θα επιλέξει ένα frame για να γίνει η απεικόνιση.

4. Γεμίζουμε με μηδενικά τον buffer (επομένως τον προσπελάζουμε) και επαναλαμβάνουμε το προηγούμενο βήμα:

```
Step 4: Initialize your buffer with zeros and repeat Step 3. What happened?

Physical address: 1433354240
```

Παρατηρούμε λοιπόν ότι τώρα έχει γίνει η απεικόνιση στην φυσική μνήμη όπως αναλύσαμε και παραπάνω.

5. Απεικονίζουμε το αρχείο **file.txt** στο χώρο διευθύνσεων της διεργασίας με την χρήση της `mmap()` (*memory-mapped file*) και τυπώνουμε το περιεχόμενό του:

Step 5: Use `mmap(2)` to read and print `file.txt`. Print the new mapping information that has been created.

Hello everyone!

Virtual Memory Map of process [22930]:

```
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslabai100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslabai100/ex4/1.1/mmap
018d8000-018f9000 rw-p 00000000 00:00 0 [heap]
7fc181cb4000-7fc181e55000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc181e55000-7fc182055000 ---p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182055000-7fc182059000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182059000-7fc18205b000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc18205b000-7fc18205f000 rw-p 00000000 00:00 0
7fc18205f000-7fc182080000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182272000-7fc182275000 rw-p 00000000 00:00 0
7fc182278000-7fc182279000 rw-p 00000000 00:00 0
7fc182279000-7fc18227a000 r--s 00000000 00:21 8665738 /home/oslab/oslabai100/ex4/1.1/file.txt
7fc18227a000-7fc18227f000 rw-p 00000000 00:00 0
7fc18227f000-7fc182280000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182280000-7fc182281000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182281000-7fc182282000 rw-p 00000000 00:00 0
7ffcc7c4d000-7ffcc7c6e000 rw-p 00000000 00:00 0 [stack]
7ffcc7dec000-7ffcc7def000 r--p 00000000 00:00 0 [vvar]
7ffcc7def000-7ffcc7df1000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
-----
7fc182279000-7fc18227a000 r--s 00000000 00:21 8665738 /home/oslab/oslabai100/ex4/1.1/file.txt
```

6. Δεσμεύουμε νέο buffer, διαμοιραζόμενο (shared) αυτή τη φορά μεταξύ διεργασιών, μεγέθους μίας σελίδας (page) με την χρήση της `mmap()` (*απεικόνιση σωρού*) και τυπώνουμε ξανά το χάρτη:

Step 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.

Virtual Memory Map of process [22930]:

```
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslabai100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslabai100/ex4/1.1/mmap
018d8000-018f9000 rw-p 00000000 00:00 0 [heap]
7fc181cb4000-7fc181e55000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc181e55000-7fc182055000 ---p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182055000-7fc182059000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc182059000-7fc18205b000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7fc18205b000-7fc18205f000 rw-p 00000000 00:00 0
7fc18205f000-7fc182080000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182272000-7fc182275000 rw-p 00000000 00:00 0
7fc182277000-7fc182278000 rw-p 00000000 00:00 0
7fc182278000-7fc182279000 rw-s 00000000 00:04 4003521 /dev/zero (deleted)
7fc182279000-7fc18227a000 r--s 00000000 00:21 8665738 /home/oslab/oslabai100/ex4/1.1/file.txt
7fc18227a000-7fc18227f000 rw-p 00000000 00:00 0
7fc18227f000-7fc182280000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182280000-7fc182281000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7fc182281000-7fc182282000 rw-p 00000000 00:00 0
7ffcc7c4d000-7ffcc7c6e000 rw-p 00000000 00:00 0 [stack]
7ffcc7dec000-7ffcc7def000 r--p 00000000 00:00 0 [vvar]
7ffcc7def000-7ffcc7df1000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
-----
7fc182278000-7fc182279000 rw-s 00000000 00:04 4003521 /dev/zero (deleted)
```

Στο σημείο αυτό καλείται η `fork()` και δημιουργείται μία νέα διεργασία.

7. Τυπώνουμε τον χάρτη εικονικής μνήμης της διεργασίας πατέρα και της διεργασίας παιδιού:

Step 7: Print parent's and child's map.

VM map of parent process:

```
Virtual Memory Map of process [26750]:
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
0080e000-0082f000 rw-p 00000000 00:00 0 [heap]
7f48418f6000-7f4841a97000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841a97000-7f4841c97000 ---p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c97000-7f4841c9b000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c9b000-7f4841c9d000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c9d000-7f4841ca1000 rw-p 00000000 00:00 0
7f4841ca1000-7f4841cc2000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841eb4000-7f4841eb7000 rw-p 00000000 00:00 0
7f4841eb9000-7f4841eba000 rw-p 00000000 00:00 0
7f4841eba000-7f4841ebb000 rw-s 00000000 00:04 4006623 /dev/zero (deleted)
7f4841ebb000-7f4841ebc000 r--s 00000000 00:21 8665738 /home/oslab/oslab100/ex4/1.1/file.txt
7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
7f4841ec1000-7f4841ec2000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841ec2000-7f4841ec3000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841ec3000-7f4841ec4000 rw-p 00000000 00:00 0
7ffda7e22000-7ffda7e43000 rw-p 00000000 00:00 0 [stack]
7ffda7f90000-7ffda7f93000 r--p 00000000 00:00 0 [vvar]
7ffda7f93000-7ffda7f95000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

VM map of child process:

```
Virtual Memory Map of process [26751]:
00400000-00403000 r-xp 00000000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
00602000-00603000 rw-p 00002000 00:21 8665740 /home/oslab/oslab100/ex4/1.1/mmap
0080e000-0082f000 rw-p 00000000 00:00 0 [heap]
7f48418f6000-7f4841a97000 r-xp 00000000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841a97000-7f4841c97000 ---p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c97000-7f4841c9b000 r--p 001a1000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c9b000-7f4841c9d000 rw-p 001a5000 08:01 6032227 /lib/x86_64-linux-gnu/libc-2.19.so
7f4841c9d000-7f4841ca1000 rw-p 00000000 00:00 0
7f4841ca1000-7f4841cc2000 r-xp 00000000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841eb4000-7f4841eb7000 rw-p 00000000 00:00 0
7f4841eb9000-7f4841eba000 rw-p 00000000 00:00 0
7f4841eba000-7f4841ebb000 rw-s 00000000 00:04 4006623 /dev/zero (deleted)
7f4841ebb000-7f4841ebc000 r--s 00000000 00:21 8665738 /home/oslab/oslab100/ex4/1.1/file.txt
7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
7f4841ec1000-7f4841ec2000 r--p 00020000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841ec2000-7f4841ec3000 rw-p 00021000 08:01 6032224 /lib/x86_64-linux-gnu/ld-2.19.so
7f4841ec3000-7f4841ec4000 rw-p 00000000 00:00 0
7ffda7e22000-7ffda7e43000 rw-p 00000000 00:00 0 [stack]
7ffda7f90000-7ffda7f93000 r--p 00000000 00:00 0 [vvar]
7ffda7f93000-7ffda7f95000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Παρατηρούμε, ότι καθώς η νέα διεργασία που δημιουργείται μέσω της `fork()` είναι αντίγραφο της παλιάς, κληρονομεί και ένα αντίγραφο της **εικονικής μνήμης** της αρχικής διεργασίας. Επιπλέον, κληρονομεί ένα αντίγραφο του πίνακα σελίδων (*page table*) της αρχικής διεργασίας, όπου αφαιρούνται και από τους δύο τα write δικαιώματα στις σελίδες που είναι private-COW.

8. Βρίσκουμε και τυπώνουμε τη φυσική διεύθυνση στην κύρια μνήμη του private buffer για τις διεργασίες πατέρα και παιδί:

Step 8: Find the physical address of the private heap buffer (main) for both the parent and the child.

```
Physical Address of private buffer requested by parent: 1312690176
VA info of private buffer in parent: 7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by child: 1312690176
VA info of private buffer in child: 7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
```

Παρατηρούμε ότι απεικονίζονται στην ίδια φυσική μνήμη όπως αναφέραμε και παραπάνω.

9. Γράφουμε στον private buffer από τη διεργασία παιδί και επαναλαμβάνουμε το προηγούμενο βήμα:

Step 9: Write to the private buffer from the child and repeat step 8. What happened?

```
VA info of private buffer in parent: 7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by parent: 1312690176
VA info of private buffer in child: 7f4841ebc000-7f4841ec1000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by child: 2871701504
```

Σε συστήματα εικονικής μνήμης, όταν δημιουργείται μία διεργασία μέσω `fork()`, δεν αντιγράφεται όλη η μνήμη της γονικής διεργασίας. Αυτό που συμβαίνει είναι αυτό που αναφέραμε παραπάνω: Αντιγράφεται η εικονική μνήμη και ο πίνακας σελίδων (με την αφαίρεση των write δικαιωμάτων αν πρόκειται για private page) και μέχρι κάποια από τις διεργασίες προσπαθήσει να γράψει σε κάποια private σελίδα η απεικόνιση της εικονικής διεύθυνσης στην φυσική διεύθυνση είναι η ίδια. Όταν κάποια διεργασία προσπαθήσει να γράψει σε κάποια private σελίδα τότε το ΛΣ βρίσκει ένα νέο πλαίσιο για την απεικόνιση, αντιγράφεται το περιεχόμενο του page στην νέα φυσική διεύθυνση και ενημερώνεται ο πίνακας σελίδων. Αυτή η τεχνική ονομάζεται **Copy-on-Write (COW)**. Επομένως, παρατηρούμε εδώ ότι όταν γράφουμε στον private buffer από τη διεργασία παιδί τότε οι φυσικές διευθύνσεις είναι διαφορετικές.

10. Γράφουμε στον shared buffer από τη διεργασία παιδί και τυπώνουμε τη φυσική διεύθυνση για τις διεργασίες πατέρα και παιδί:

Step 10: Write to the shared heap buffer (main) from child and get the physical address for both the parent and the child. What happened?

```
VA info of shared buffer in parent: 7f4841eba000-7f4841ebb000 rw-s 00000000 00:04 4006623 /dev/zero (deleted)
Physical Address of shared buffer requested by parent: 937357312
VA info of shared buffer in child: 7f4841eba000-7f4841ebb000 rw-s 00000000 00:04 4006623 /dev/zero (deleted)
Physical Address of shared buffer requested by child: 937357312
```

Παρατηρούμε εδώ, σε σύγκριση με τον private buffer, ότι ο shared buffer απεικονίζεται στην ίδια φυσική διεύθυνση και για τις δύο διεργασίες. Αυτό συμβαίνει διότι η σελίδα τώρα είναι διαμοιραζόμενη (shared) μεταξύ των διεργασιών (**MAP_SHARED** flag στην `mmap()`) και επομένως αντιστοιχίζεται στην ίδια φυσική διεύθυνση. Αυτό δίνει την δυνατότητα για διαδιεργασιακή επικοινωνία αφού “μοιράζονται” την ίδια μνήμη (βλ. 1.2).

11. Απαγορεύουμε τις εγγραφές στον shared buffer για την διεργασία παιδί (αφαιρούμε το **write** permission) με χρήση της `mprotect()`:

VMA of parent:

```
7f4841eba000-7f4841ebb000 rw-s 00000000 00:04 4006623 /dev/zero (deleted)
```

VMA of child:

```
7f4841eba000-7f4841ebb000 r--s 00000000 00:04 4006623 /dev/zero (deleted)
```

Παρατηρούμε λοιπόν ότι έχει αφαιρεθεί το write δικαίωμα από την διεργασία παιδί (αν πάει να γράψει θα έχουμε **segfault**).

12. Τέλος, αποδεσμεύουμε όλους τους buffers στις δύο διεργασίες με χρήση της `munmap()`.

Άσκηση 1.2 (Παράλληλος υπολογισμός συνόλου Mandelbrot με διεργασίες αντί για νήματα)

Σε αυτήν την άσκηση, τροποποιούμε το πρόγραμμα υπολογισμού του **Mandelbrot set** της **Άσκησης 3**, ώστε αντί να χρησιμοποιούμε threads (pthreads) για την παραλληλοποίηση του υπολογισμού του, να χρησιμοποιούμε διεργασίες (processes).

1.2.1 Semaphores πάνω από διαμοιραζόμενη μνήμη

Το πρόβλημα που δημιουργείται είναι το εξής: Εφόσον χρησιμοποιούμε διεργασίες, πως θα μπορέσουμε να έχουμε μια κοινή μνήμη όπου όλες οι διεργασίες μπορούν να βλέπουν τους semaphores, ώστε να γνωρίζουν ότι είναι κλειδωμένο το κρίσιμο τμήμα τους. Εδώ είναι που καλούμαστε να επεκτείνουμε την συνάρτηση **create_shared_memory_area()**, η οποία θα επιστρέφει έναν pointer στην πρώτη θέση μνήμης ενός τμήματος που δεσμεύτηκε και είναι **κοινό** (shared/anonymous) και **προσβάσιμο (read/write)** από όλες τις διεργασίες παιδιά της αρχικής διεργασίας. Αντίστοιχα, χρησιμοποιούμε και την συνάρτηση **destroy_shared_memory_area()**, έτσι ώστε να μπορούμε αποδεσμεύσουμε την μνήμη.

Ο **πηγαίος κώδικας (source code)** φαίνεται παρακάτω:

```
/*
 * 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 <signal.h>
#include <sys/mman.h>
#include <sys/wait.h>

#include "../helpers/mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

#define perror_pthread(ret, msg) \
    do { errno = ret; perror(msg); } while (0)

sem_t *sem;

/*
 * 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;

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

/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */
void sigint_handler(int signum)
{
    reset_xterm_color(1);
    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 */
    addr = mmap(NULL, pages * sysconf(_SC_PAGE_SIZE), PROT_READ |
PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if(addr==MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    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 fork_execute(int line, int procnt)
{
    //same as ex3
    int line_num;
    int color_val[x_chars];
    for (line_num=line; line_num<y_chars; line_num+=procnt) {
        compute_mandel_line(line_num, color_val);
        if(sem_wait(&sem[line])<0) {
            perror("sem_wait");
            exit(1);
        }
        output_mandel_line(1, color_val);
        if(sem_post(&sem[(line_num+1) % procnt])<0) {
            perror("sem_post");
            exit(1);
        }
    }
}

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

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

    /*
     * signal handling
     */
    struct sigaction sa;
    sa.sa_handler=sigint_handler;

```

```

sa.sa_flags=0;
sigemptyset(&sa.sa_mask);
if(sigaction(SIGINT, &sa, NULL)<0) {
    perror("sigaction");
    exit(1);
}

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

sem=create_shared_memory_area(procnt * sizeof(sem_t)); //allocate shared
//mem to store
//semaphore array

for (i=0; i<procnt; i++) {
    if(sem_init(&sem[i],1,0)<0) {
        perror("sem_init");
        exit(1);
    }
}

if(sem_post(&sem[0])<0) {
    perror("sem_post");
    exit(1);
}

/*create the processes and call the execution function*/
pid_t child_pid;
for(i=0; i<procnt; i++) {
    child_pid=fork();
    if(child_pid< 0) {
        perror("error with creation of child");
        exit(1);
    }
    if(child_pid== 0) {
        fork_execute(i, procnt);
        exit(1);
    }
}
for(i=0; i<procnt ; i++) {
    child_pid= wait(&status);
}

for(i=0; i<procnt ; i++) {
    sem_destroy(&sem[i]);
}

destroy_shared_memory_area(sem, procnt * sizeof(sem_t));

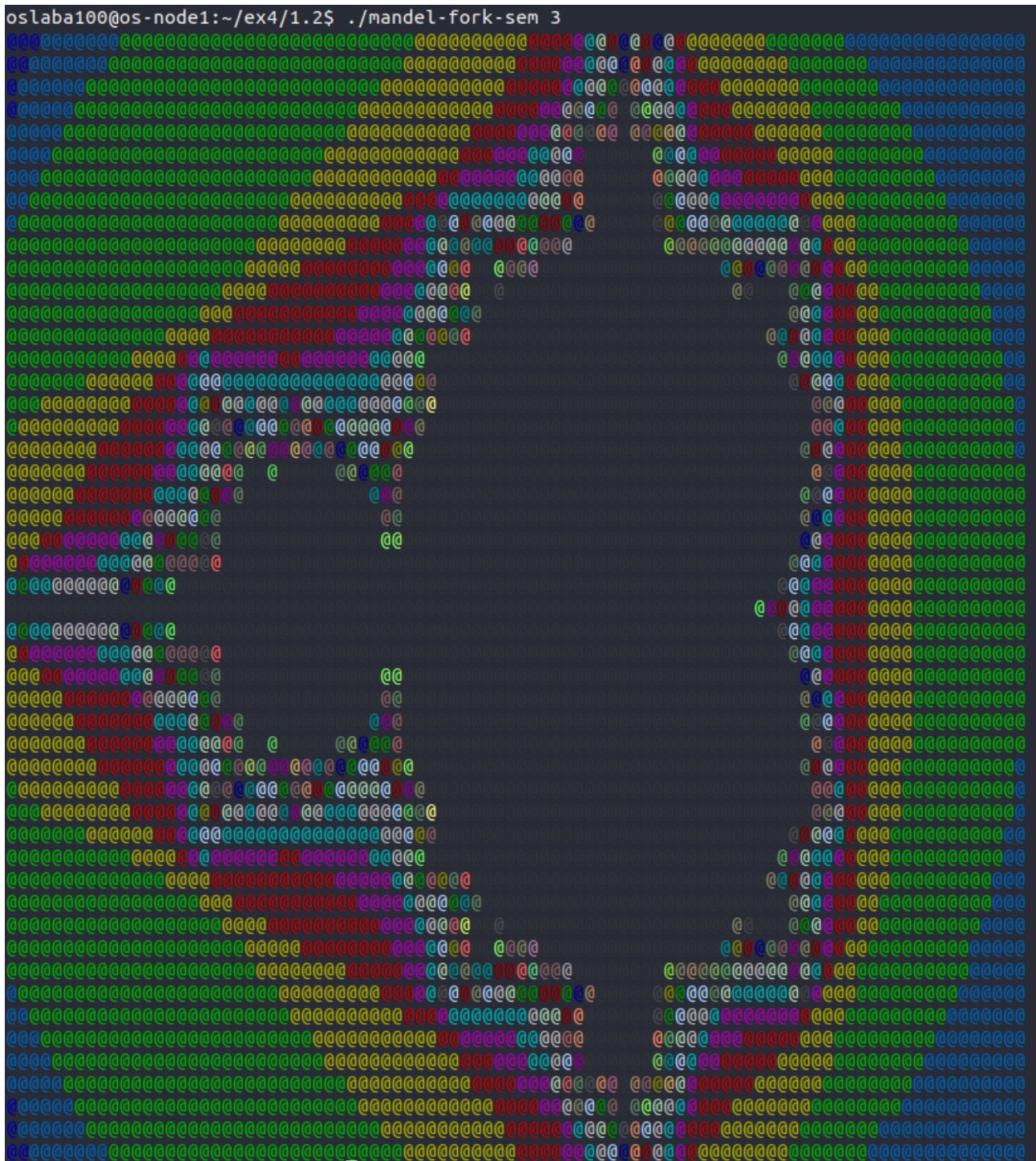
reset_xterm_color(1);
return 0;
}

```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Το μεταγλωττίζουμε, κάνουμε το κατάλληλο linking και το εκτελούμε:


```
oslaba100@os-node1:~/ex4/1.2$ ./mandel-fork-sem 3
```



1.

Περιμένουμε καλύτερη απόδοση να έχει η υλοποίηση με τα threads, διότι η δημιουργία διεργασιών είναι πολύ πιο χρονοβόρα, λόγω των απαραίτητων ενεργειών που πρέπει να γίνουν, όπως δημιουργία των PCB's, αφαίρεση δικαιώματος write από τα pages που έχει δεσμεύσει η γονική διεργασία για να μπορεί να υποστηριχτεί το Copy-on-Write κλπ.. Παράλληλα, και η εναλλαγή διεργασιών είναι πιο αργή από την εναλλαγή των threads, αφού για να γίνει το context switch πρέπει να αντικατασταθεί και το PCB εκτός από την αποθήκευση του PC και των registers. Επιπλέον, τα threads έχουν εν γένει κοινή μνήμη, ενώ στην περίπτωση των processes πρέπει να χρησιμοποιήσουμε την συνάρτηση `create_shared_memory_area()`, η οποία κάνει `mmap()`.

1.2.2 Υλοποίηση χωρίς semaphores

Εφόσον δεν έχουμε κάποιο σχήμα συγχρονισμού με semaphores μπορούμε απλά να δεσμεύσουμε έναν πίνακα μεγέθους όσο και αυτού που θέλουμε να τυπώσουμε. Με χρήση κοινής μνήμης θα δημιουργήσουμε ένα buffer που θα κρατήσει ότι υπολογίζει η κάθε διεργασία. Έπειτα, θα δημιουργήσουμε τις διεργασίες και η κάθε μια θα υπολογίζει και θα γεμίζει την αντίστοιχη γραμμή στο buffer που της αντιστοιχεί. Τέλος, η αρχική διεργασία θα αναλάβει να τυπώσει τον δισδιάστατο buffer.

Ο πηγαίος κώδικας (source code) φαίνεται παρακάτω:

```
/*  
 * 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 <errno.h>  
#include <signal.h>  
#include <sys/mman.h>  
#include <sys/wait.h>  
  
#include "../helpers/mandel-lib.h"  
  
#define MANDEL_MAX_ITERATION 100000  
  
#define perror_pthread(ret, msg) \  
    do { errno=ret; perror(msg); } while (0)  
  
int **buff;  
  
/*  
 * 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;  
  
//helping functions  
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 usage(char *argv0)
{
    fprintf(stderr, "Usage: %s thread_count\n\n"
        "Exactly one argument required:\n"
        "    thread_count: The number of threads 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 */
    addr=mmap(NULL, pages * sysconf(_SC_PAGE_SIZE), PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if(addr==MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    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);
}

}

/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */
void sigint_handler(int signum)
{
    reset_xterm_color(1);
    exit(1);
}

void fork_execute(int line, int procnt)
{
    int line_num;
    //every process writes to the buffer every line = #processes*iterator
    for (line_num=line ; line_num<y_chars; line_num+=procnt) {
        compute_mandel_line(line_num, buff[line_num]);
    }
    return;
}

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

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

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

    /*
     * signal handling
     */
    struct sigaction sa;
    sa.sa_handler=sigint_handler;
    sa.sa_flags=0;
    sigemptyset(&sa.sa_mask);
    if(sigaction(SIGINT, &sa, NULL)<0) {
        perror("sigaction");
        exit(1);
    }
}

```



```

buff=create_shared_memory_area(y_chars * sizeof(int)); //create the
//initial id
//array

for (i=0; i<y_chars; i++) {
    buff[i]=create_shared_memory_area(x_chars * sizeof(char));
    //go to every position and make x_chars of memory
}

//create processes and call execution function
pid_t child_pid;
for(i=0 ; i<procnt ; i++) {
    child_pid=fork();
    if(child_pid<0) {
        perror("error with creation of child");
        exit(1);
    }
    if(child_pid==0) {
        fork_execute(i, procnt);
        exit(1);
    }
}

for(i=0; i<procnt ; i++) {
    child_pid=wait(&status);
}

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

for(i=0; i<y_chars; i++){
    destroy_shared_memory_area(buff[i], sizeof(buff[i]));
}
destroy_shared_memory_area(buff, sizeof(buff));
reset_xterm_color(1);
return 0;
}

```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Το μεταγλωττίζουμε, κάνουμε το κατάλληλο linking και το εκτελούμε:

1.

Σε αυτό το σημείο, ο συγχρονισμός επιτυγχάνεται από το γεγονός πως η κάθε διεργασία γράφει στον buffer μόνο εκεί που της αναλογεί και έτσι δημιουργείται παράλληλα το output, χωρίς η μια διεργασία να επηρεάζει την άλλη. Πρακτικά, δεν υπάρχει κρίσιμο τμήμα, διότι την ευθύνη για το output την έχει η αρχική διεργασία.

Εάν είχαμε μικρότερο πίνακα (**NPROCS x x_chars** αντί **y_chars x x_chars**) τότε δεν θα μπορούσαμε να τυπώναμε με την μια όλο το output και θα έπρεπε να γίνει σε “δόσεις”. Μια λύση θα ήταν η χρήση σημάτων, δηλαδή όταν μια διεργασία υπολογίσει την γραμμή της θα κάνει *raise(SIGSTOP)*, έπειτα η αρχική διεργασία θα περιμένει μέχρι όλες οι διεργασίες παιδιά της να την ενημερώσουν πως έχουν σταματήσει και άρα έχουν υπολογίσει την γραμμή που τους αντιστοιχεί, θα τυπώσει το αντίστοιχο buffer και θα τις ενεργοποιήσει μία μία, επαναλαμβάνοντας την παραπάνω διαδικασία μέχρι να υπολογιστεί η έξοδος.