Λειτουργικά Συστήματα, 6ο Εξάμηνο, 2022-2023

Δημουλάς Ιωάννης, 03120083,
Ματζόρι Ενρίκα Ηλιάνα, 03120143
oslab20

Άσκηση 4η

Στο τέλος παρατίθεται το Makefile για την 2η άσκηση.

Άσκηση 1.1

Παρατίθεται ο κώδικας της 1ης άσκησης :

```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.
   */
  printf("Child's memory map:\n");
  show_maps();
  printf("Physical address for shared_buf after initialization: %ld
\n",get_physical_address((uint64_t)heap_shared_buf));
  /*
   * Step 8 - Child
   */
  if (0 != raise(SIGSTOP))
     die("raise(SIGSTOP)");
  /*
   * TODO: Write your code here to complete child's part of Step 8.
   */
  printf("Physical address for child: %ld
\n",get_physical_address((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.
   */
  printf("Old Physical address for child(before changes on heap_private_buf):
%ld \n",get_physical_address((uint64_t)heap_private_buf));
  unsigned int i=0;
  for(i=0; i< buffer_size; i++){
     heap_private_buf[i]=2;
  }
  printf("New Physical address for child(after changes on heap_private_buf):
%ld \n",get_physical_address((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.
   */
```

```
  for(i=0; i< buffer_size; i++){
     heap_shared_buf[i]=2;
  }
  printf("New Physical address of heap_shared_buf for child(after changes on
heap_shared_buf): %ld \n",get_physical_address((uint64_t)heap_shared_buf));
  /*
   * Step 11 - Child
   */
  if (0 != raise(SIGSTOP))
     die("raise(SIGSTOP)");
  /*
   * TODO: Write your code here to complete child's part of Step 11.
   */
  //prot-> λες ποια δικαιώματα έχει τώρα η διεργασία
  mprotect(heap_shared_buf, buffer_size,PROT_READ);
  printf("Memory Map for child after changing permissions\n");
  show_maps();

  /*
   * Step 12 - Child
   */
  /*
   * TODO: Write your code here to complete child's part of Step 12.
   */
  munmap(heap_private_buf, buffer_size);
  munmap(heap_shared_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.
   */
  printf("Parent's memory map:\n");
```

```
  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.
   */
  printf("Physical address for 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 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.
   */
  printf("Physical address for 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();

  /*
   * TODO: Write your code here to complete parent's part of Step 10.
   */
  printf("Physical address of heap_shared_buf for 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();

  /*
   * TODO: Write your code here to complete parent's part of Step 11.
   */
  printf("Memory Map for parent after changing permissions for child\n");
  show_maps();

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

int main(void)
```

```
{
  pid_t mypid, p;
  int fd = -1;
  uint64_t pa;

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

  //printf("buffer size: %d\n" , buffer_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.
   */

  //long page_size = get_page_size();
  //printf("size = %ld\n", page_size);

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

  if (heap_private_buf == MAP_FAILED) {
     perror("mmap");
     return -1;
  }
  show_maps();
  printf("Virtual Address Area (VMA) that heap_private_buf belongs to\n");
  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();
  /*
   * TODO: Write your code here to complete Step 3.
```

```
   */
  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();
  /*
   * TODO: Write your code here to complete Step 4.
   */
  unsigned int i=0;
  for(i=0; i< buffer_size; i++){
     heap_private_buf[i]=0;
  }
  printf("Physical address after initialization: %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();
  /*
   * TODO: Write your code here to complete Step 5.
   */
  //βρίσκουμε το fd του file.txt μέσω της open
  fd = open("file.txt", O_RDONLY);
  if(fd==-1){
     perror("open");
     return -1;
  }
  //off_t file_size = lseek(fd, 0, SEEK_END);  // Determine file size
  //χρησιμοποιούμε σαν length το μέγεθος της σελίδας και έτσι, ακόμα και αν
το αρχείο ήταν μεγαλύτερο του page,
  // αυτό δεν μας πειράζει γιατί η mmap() θα κάνει το mapping με μέγεθος
πολλαπλάσιο της σελίδας
  file_shared_buf = mmap(NULL, buffer_size, PROT_READ,MAP_SHARED, fd, 0);
  if (file_shared_buf == MAP_FAILED) {
     perror("mmap");
     return -1;
  }

  char c;
  for(i=0; i< buffer_size; i++){
     c = file_shared_buf[i];
     if(c != EOF){
        printf("%c",c);
     }
```

```
    else break;
  }
  /*
   * 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, MAP_SHARED
| MAP_ANONYMOUS, -1, 0);
  if (file_shared_buf == MAP_FAILED) {
     perror("mmap");
     return -1;
  }
  //τον αρχικοποιούμε με άσσους ώστε να εμφανιστεί στο page table και άρα να
υπάρχει η φυσική του διεύθυνση μνήμης
  for(i=0; i< buffer_size; i++){
     heap_shared_buf[i]=0;
  }
  show_maps();
  printf("Virtual Address Area (VMA) that heap_shared_buf belongs to\n");
  show_va_info((uint64_t)heap_shared_buf);

  printf("Physical address for shared_buf after initialization: %ld
\n",get_physical_address((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;
}
```

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

```
Step 1: Print the virtual address space map of this process [31626].


Virtual Memory Map of process [31626]:
00400000-00402000 r-xp 00000000 fe:10 7619813                            /store/homes/oslab/oslab20/ask4/mmap
00601000-00602000 rw-p 00001000 fe:10 7619813                            /store/homes/oslab/oslab20/ask4/mmap
0095a000-0097b000 rw-p 00000000 00:00 0                                  [heap]
7fa41401b000-7fa4141bc000 r-xp 00000000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7fa4141bc000-7fa4143bc000 ---p 001a1000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7fa4143bc000-7fa4143c0000 r--p 001a1000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7fa4143c0000-7fa4143c2000 rw-p 001a5000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7fa4143c2000-7fa4143c6000 rw-p 00000000 00:00 0
7fa4143c6000-7fa4143e6000 r-xp 00000000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7fa4145d9000-7fa4145dc000 rw-p 00000000 00:00 0
7fa4145e1000-7fa4145e6000 rw-p 00000000 00:00 0
7fa4145e6000-7fa4145e7000 r--p 00020000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7fa4145e7000-7fa4145e8000 rw-p 00021000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7fa4145e8000-7fa4145e9000 rw-p 00000000 00:00 0
7ffcada64000-7ffcada85000 rw-p 00000000 00:00 0                          [stack]
7ffcadad4000-7ffcadad6000 r-xp 00000000 00:00 0                          [vdso]
7ffcadad6000-7ffcadad8000 r--p 00000000 00:00 0                          [vvar]
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.

```
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 [31879]:
00400000-00402000 r-xp 00000000 fe:10 7619813                            /store/homes/oslab/oslab20/ask4/mmap
00601000-00602000 rw-p 00001000 fe:10 7619813                            /store/homes/oslab/oslab20/ask4/mmap
01d16000-01d37000 rw-p 00000000 00:00 0                                  [heap]
7ff594ab5000-7ff594c56000 r-xp 00000000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7ff594c56000-7ff594e56000 ---p 001a1000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7ff594e56000-7ff594e5a000 r--p 001a1000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7ff594e5a000-7ff594e5c000 rw-p 001a5000 fe:01 1045760                    /lib/x86_64-linux-gnu/libc-2.19.so
7ff594e5c000-7ff594e60000 rw-p 00000000 00:00 0
7ff594e60000-7ff594e80000 r-xp 00000000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7ff595073000-7ff595076000 rw-p 00000000 00:00 0
7ff59507a000-7ff595080000 rw-p 00000000 00:00 0
7ff595080000-7ff595081000 r--p 00020000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7ff595081000-7ff595082000 rw-p 00021000 fe:01 1044705                    /lib/x86_64-linux-gnu/ld-2.19.so
7ff595082000-7ff595083000 rw-p 00000000 00:00 0
7fff0bcbf000-7fff0bce0000 rw-p 00000000 00:00 0                          [stack]
7fff0bde9000-7fff0bdeb000 r-xp 00000000 00:00 0                          [vdso]
7fff0bdeb000-7fff0bded000 r--p 00000000 00:00 0                          [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
-----------------------------------------------------------

Virtual Address Area (VMA) that heap_private_buf belongs to
7ff59507a000-7ff595080000 rw-p 00000000 00:00 0
```

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;

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

VA[0x7fdb8bc3a000] is not mapped; no physical memory allocated.
Physical address: 0
```

Αφού δεσμεύσαμε τον χώρο στην μνήμη μέσω της mmap(), χρησιμοποιώντας τον heap_private_buf, και είδαμε ότι ο buffer υπάρχει σαν δεσμευμένος εικονικός χώρος διευθύνσεων στο memory map της τρέχουσας διεργασίας, δηλαδή η μνήμη έχει δεσμευτεί, δεν έχει γίνει ακόμα η απεικόνιση της εικονικής μνήμης στην φυσική. Για τον λόγο αυτό όταν ζητάμε να μας δοθεί η διεύθυνση του buffer στην φυσική μνήμη, λαμβάνουμε το μήνυμα ότι αυτή δεν υπάρχει. Γνωρίζουμε ότι όταν μία διεργασία απεικονίζει ένα αρχείο στη μνήμη, ο πίνακας σελίδων (page table) της δεν

αλλάζει και συνεπώς στην συγκεκριμένη περίπτωση, όταν πηγαίνουμε σε αυτόν για να βρούμε την εν λόγω διεύθυνση, προκύπτει page fault.
Επειδή η φυσική μνήμη δεσμεύεται on demand απ' το ΛΣ όταν πάει να προσπελαστεί, μόνο μετά από μία τέτοια πράξη θα απεικονιστεί στο page table και άρα θα αποκτήσει φυσική διεύθυνση μνήμης.

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

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

Physical address after initialization: 5318868992
```

Όπως και αναμέναμε, παρατηρούμε ότι, τώρα που επεξεργαστήκαμε τον buffer, η φυσική μνήμη δεσμεύτηκε απ' το ΛΣ και άρα πλέον ο buffer έχει φυσική διεύθυνση μνήμης την οποία και τυπώνουμε.

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

Χρησιμοποιώντας την mmap() και εκτυπώνοντας τον file_shared_buf έχουμε:

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

Hello everyone!
```

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.

```
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 [604763]:
00400000-00402000 r-xp 00000000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00602000-00603000 rw-p 00002000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00756000-00777000 rw-p 00000000 00:00 0                                  [heap]
7fbb812d3000-7fbb812f5000 r--p 00000000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fbb812f5000-7fbb8144e000 r-xp 00022000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fbb8144e000-7fbb8149d000 r--p 0017b000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fbb8149d000-7fbb814a1000 r--p 001c9000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fbb814a1000-7fbb814a3000 rw-p 001cd000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fbb814a3000-7fbb814a9000 rw-p 00000000 00:00 0
7fbb814ad000-7fbb814ae000 rw-s 00000000 00:01 14641                      /dev/zero (deleted)
7fbb814ae000-7fbb814af000 r--s 00000000 00:26 7619832                    /home/oslab/oslab20/ask4/file.txt
7fbb814af000-7fbb814b0000 r--p 00000000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fbb814b0000-7fbb814d0000 r-xp 00001000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fbb814d0000-7fbb814d8000 r--p 00021000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fbb814d8000-7fbb814d9000 rw-p 00000000 00:00 0
7fbb814d9000-7fbb814da000 r--p 00029000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fbb814da000-7fbb814db000 rw-p 0002a000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fbb814db000-7fbb814dc000 rw-p 00000000 00:00 0
7ffe9356d000-7ffe9358e000 rw-p 00000000 00:00 0                          [stack]
7ffe935b5000-7ffe935b9000 r--p 00000000 00:00 0                          [vvar]
7ffe935b9000-7ffe935bb000 r-xp 00000000 00:00 0                          [vdso]
-----------------------------------------------------

Virtual Address Area (VMA) that heap_shared_buf belongs to
7fbb814ad000-7fbb814ae000 rw-s 00000000 00:01 14641                      /dev/zero (deleted)
```

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

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

Parent's memory map:

Virtual Memory Map of process [604882]:
00400000-00402000 r-xp 00000000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00602000-00603000 rw-p 00002000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
01c22000-01c43000 rw-p 00000000 00:00 0                                  [heap]
7f685f7e6000-7f685f808000 r--p 00000000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f808000-7f685f961000 r-xp 00022000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f961000-7f685f9b0000 r--p 0017b000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b0000-7f685f9b4000 r--p 001c9000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b4000-7f685f9b6000 rw-p 001cd000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b6000-7f685f9bc000 rw-p 00000000 00:00 0
7f685f9c0000-7f685f9c1000 rw-s 00000000 00:01 14642                      /dev/zero (deleted)
7f685f9c1000-7f685f9c2000 r--s 00000000 00:26 7619832                    /home/oslab/oslab20/ask4/file.txt
7f685f9c2000-7f685f9c3000 r--p 00000000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9c3000-7f685f9e3000 r-xp 00001000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9e3000-7f685f9eb000 r--p 00021000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9eb000-7f685f9ec000 rw-p 00000000 00:00 0
7f685f9ec000-7f685f9ed000 r--p 00029000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9ed000-7f685f9ee000 rw-p 0002a000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9ee000-7f685f9ef000 rw-p 00000000 00:00 0
7ffeb2f92000-7ffeb2fb3000 rw-p 00000000 00:00 0                          [stack]
7ffeb2feb000-7ffeb2fef000 r--p 00000000 00:00 0                          [vvar]
7ffeb2fef000-7ffeb2ff1000 r-xp 00000000 00:00 0                          [vdso]
-----------------------------------------------------
```

```
Child's memory map:

Virtual Memory Map of process [604883]:
00400000-00402000 r-xp 00000000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00602000-00603000 rw-p 00002000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
01c22000-01c43000 rw-p 00000000 00:00 0                                  [heap]
7f685f7e6000-7f685f808000 r--p 00000000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f808000-7f685f961000 r-xp 00022000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f961000-7f685f9b0000 r--p 0017b000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b0000-7f685f9b4000 r--p 001c9000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b4000-7f685f9b6000 rw-p 001cd000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f685f9b6000-7f685f9bc000 rw-p 00000000 00:00 0
7f685f9c0000-7f685f9c1000 rw-s 00000000 00:01 14642                      /dev/zero (deleted)
7f685f9c1000-7f685f9c2000 r--s 00000000 00:26 7619832                    /home/oslab/oslab20/ask4/file.txt
7f685f9c2000-7f685f9c3000 r--p 00000000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9c3000-7f685f9e3000 r-xp 00001000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9e3000-7f685f9eb000 r--p 00021000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9eb000-7f685f9ec000 rw-p 00000000 00:00 0
7f685f9ec000-7f685f9ed000 r--p 00029000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9ed000-7f685f9ee000 rw-p 0002a000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f685f9ee000-7f685f9ef000 rw-p 00000000 00:00 0
7ffeb2f92000-7ffeb2fb3000 rw-p 00000000 00:00 0                          [stack]
7ffeb2feb000-7ffeb2fef000 r--p 00000000 00:00 0                          [vvar]
7ffeb2fef000-7ffeb2ff1000 r-xp 00000000 00:00 0                          [vdso]
-------------------------------------------------------
```

Παρατηρούμε ότι μετά την εκτέλεση της fork() και άρα της δημιουργίας της διεργασίας-παιδιού, τα 2 memory maps για τις αντίστοιχες διεργασίες ταυτίζονται. Αυτό συμβαίνει γιατί η διεργασία-παιδί αποτελεί αντίγραφο της διεργασίας-πατέρα και, με την εκτέλεση της fork() , κληρονομεί ένα αντίγραφο του χάρτη μνήμης και ένα του πίνακα σελίδων απ' τον πατέρα. Αυτό που συμβαίνει όμως, είναι ότι, και απ' τις δύο διεργασίες αφαιρούνται τα δικαιώματα εγγραφής στο page table τους. Έτσι, αν κάποια διεργασία πάει να κάνει κάποια αλλαγή, το ΛΣ ακολουθεί τον μηχανισμό της Αντιγραφής κατά την Εγγραφή(Copy-On-Write /COW). Τότε θα προκύψει page fault αλλά δεν θα διαγραφεί η διεργασία, αλλά(αφού το ΛΣ ελέγξει τα "πραγματικά" δικαιώματα της εν λόγω διεργασίας και συμβαδίζουν με την αλλαγή που αυτή πάει να κάνει τότε) δημιουργείται μία καινούργια, ανανεωμένη σελίδα στην φυσική μνήμη που αφορά την διεργασία που έκανε την τροποποίηση. Τότε, ανανεώνεται και το page table της συγκεκριμένης διεργασίας ώστε να "δείχνει" στην νέα σελίδα στην φυσική μνήμη. Έτσι, τυχόν αλλαγές που γίνονται στην φυσική μνήμη που μπορεί να κάνει μία διεργασία δεν θα επηρεάζει τις υπόλοιπες.

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?

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

Physical address for parent: 6072700928
Physical address for child: 6072700928
```

Όπως εξηγήθηκε στην προηγούμενη ερώτηση, όπως αναμέναμε ο private buffer βρίσκεται στην ίδια διεύθυνση φυσικής μνήμης και για τις δύο διεργασίες. Αυτό συμβαίνει γιατί μετά την fork η φυσική μνήμη είναι πλέον κοινή και τα 2 page tables που πλέον υπάρχουν(του παιδιού που είναι αντίγραφο του πατέρα) δείχνουν στις ίδιες φυσικές διευθύνσεις.

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

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

Physical address for parent: 7552724992
Old Physical address for child(before changes on heap_private_buf): 7552724992
New Physical address for child(after changes on heap_private_buf): 4779044864
```

Όπως εξηγήθηκε προηγουμένως και όπως αναμέναμε η διεύθυνση φυσικής μνήμης του ανανεωμένου private buffer για την διεργασία-παιδί είναι διαφορετική απ' ό,τι ήταν πριν την τροποποίησή της. Ωστόσο, για την διεργασία-πατέρα η εν λόγω διεύθυνση δεν άλλαξε πριν και μετά την τροποποίηση. Αυτό συνέβη λόγω του μηχανισμού COW που ακολουθεί το ΛΣ(εδώ ο private buffer είναι MAP_PRIVATE οπότε αν μία διεργασία κάνει κάποια αλλαγή δεν θα είναι ορατή στις υπόλοιπες διεργασίες). Η διεύθυνση που λέει το "New Physical address for child " είναι η διεύθυνση της νέας σελίδας που προστέθηκε στην φυσική μνήμη και περιέχει τον τροποποιημένο buffer. Στα δύο page tables, το page στο VMA για τον συγκεκριμένο buffer παραμένει και για τις 2 διεργασίες το ίδιο, ωστόσο ,για την διεργασία-παιδί συγκεκριμένη διεύθυνση εικονικής μνήμης αφορά πλέον/"δείχνει" σε διαφορετικό block της φυσικής μνήμης(δηλαδή τελικά έχουμε ίδια addresses στην εικονική μνήμη αλλά διαφορετικά για την φυσική μνήμη).

10. Γράψτε στον shared buffer (Βήμα6) από τη διεργασία παιδί και τυπώστε τη φυσική του διεύθυνση για τις διεργασίες πατέρα και παιδί. Τι παρατηρείτε σε σύγκριση με τον private buffer?

```
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?

Physical address of heap_shared_buf for parent: 5204779008
New Physical address of heap_shared_buf for child(after changes on heap_shared_buf): 5204779008
```

Ο shared buffer ορίστηκε ώστε να είναι MAP_SHARED, δηλαδή πολλές διαφορετικές διεργασίες μοιράζονται την ίδια mapped memory region και οι τροποποιήσεις που τυχόν να γίνουν απ' τη μία διεργασία θα είναι ορατές και από τις υπόλοιπες. Εδώ δηλαδή δεν υφίσταται ο μηχανισμός του Copy-On-Write αφού ό,τι αλλαγές κάνει η μία διεργασία αυτόματα θα γίνουν για όλες τις υπόλοιπες. Γι' αυτό και ο shared_buffer αφού τροποποιηθεί απ' την διεργασία-παιδί δεν αλλάζει διεύθυνση στην φυσική μνήμη για το παιδί και παραμένει η ίδια κοινή διεύθυνση και για τις δύο διεργασίες. Στην ίδια διεύθυνση με πριν βρίσκεται πλέον ο τροποποιημένος 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.

```
Step 11: Disable writing on the shared buffer for the child. Verify through the maps for the parent and the child.

Memory Map for parent after changing permissions for child

Virtual Memory Map of process [340907]:
00400000-00403000 r-xp 00000000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00602000-00603000 rw-p 00002000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
02550000-02571000 rw-p 00000000 00:00 0                                  [heap]
7fc1aba3b000-7fc1aba5d000 r--p 00000000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1aba5d000-7fc1abbb6000 r-xp 00022000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abbb6000-7fc1abc05000 r--p 0017b000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc05000-7fc1abc09000 r--p 001c9000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc09000-7fc1abc0b000 rw-p 001cd000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc0b000-7fc1abc11000 rw-p 00000000 00:00 0
7fc1abc14000-7fc1abc15000 rw-s 00000000 00:01 8050                       /dev/zero (deleted)
7fc1abc15000-7fc1abc16000 r--s 00000000 00:26 7619832                    /home/oslab/oslab20/ask4/file.txt
7fc1abc16000-7fc1abc17000 r--p 00000000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc17000-7fc1abc37000 r-xp 00001000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc37000-7fc1abc3f000 r--p 00021000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc3f000-7fc1abc40000 rw-p 00000000 00:00 0
7fc1abc40000-7fc1abc41000 r--p 00029000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc41000-7fc1abc42000 rw-p 0002a000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc42000-7fc1abc43000 rw-p 00000000 00:00 0
7fff95bb6000-7fff95bd7000 rw-p 00000000 00:00 0                          [stack]
7fff95bec000-7fff95bf0000 r--p 00000000 00:00 0                          [vvar]
7fff95bf0000-7fff95bf2000 r-xp 00000000 00:00 0                          [vdso]
-------------------------------------------------------
```

```
Memory Map for child after changing permissions

Virtual Memory Map of process [340908]:
00400000-00403000 r-xp 00000000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
00602000-00603000 rw-p 00002000 00:26 7619813                            /home/oslab/oslab20/ask4/mmap
02550000-02571000 rw-p 00000000 00:00 0                                  [heap]
7fc1aba3b000-7fc1aba5d000 r--p 00000000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1aba5d000-7fc1abbb6000 r-xp 00022000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abbb6000-7fc1abc05000 r--p 0017b000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc05000-7fc1abc09000 r--p 001c9000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc09000-7fc1abc0b000 rw-p 001cd000 fe:01 144567                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fc1abc0b000-7fc1abc11000 rw-p 00000000 00:00 0
7fc1abc14000-7fc1abc15000 r--s 00000000 00:01 8050                       /dev/zero (deleted)
7fc1abc15000-7fc1abc16000 r--s 00000000 00:26 7619832                    /home/oslab/oslab20/ask4/file.txt
7fc1abc16000-7fc1abc17000 r--p 00000000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc17000-7fc1abc37000 r-xp 00001000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc37000-7fc1abc3f000 r--p 00021000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc3f000-7fc1abc40000 rw-p 00000000 00:00 0
7fc1abc40000-7fc1abc41000 r--p 00029000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc41000-7fc1abc42000 rw-p 0002a000 fe:01 144563                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fc1abc42000-7fc1abc43000 rw-p 00000000 00:00 0
7fff95bb6000-7fff95bd7000 rw-p 00000000 00:00 0                          [stack]
7fff95bec000-7fff95bf0000 r--p 00000000 00:00 0                          [vvar]
7fff95bf0000-7fff95bf2000 r-xp 00000000 00:00 0                          [vdso]
-------------------------------------------------------
```

Παρατηρούμε ότι το memory map για την διεργασία-πατέρα διαφέρει από αυτό για τη διεργασία-παιδί αφού για την ίδια διεύθυνση στην φυσική μνήμη(τον shared buffer) ο πατέρας έχει

όλα τα permissions ενώ το παιδί δεν έχει δικαίωμα εγγραφής(λείπει το w δηλαδή το write permission).

12.Αποδεσμεύστε όλους τους buffers στις δύο διεργασίες.

Χρησιμοποιήσαμε την συνάρτηση munmap() για να αποδεσμεύσουμε τα memory regions που κάναμε map για τον εκάστοτε buffer.

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

Άσκηση 1.2.1: Semaphores πάνω από διαμοιραζόμενη μνήμη

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


/*TODO header file for m(un)map*/
#include <semaphore.h>
#include <sys/mman.h>

#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;
```

```c
/*
* 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;

//φτιάχνουμε τον σημαφόρο
sem_t *sem;

/*
* A (distinct) instance of this structure
* is passed to each process
*/
struct process_info_struct {
   pid_t pid;

   //int* color_val; /* Pointer to array to manipulate */

   int mypid; /* Application-defined process id */
   int pcnt;
};

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 usage(char *argv0)
{
   fprintf(stderr, "Usage: %s thread_count array_size\n\n"
                   "Exactly two argument required:\n"
                   "    thread_count: The number of threads to create.\n"
                   "    array_size: The size of the array to run with.\n",
           argv0);
   exit(1);
}
/*
* 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(void *arg)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    struct process_info_struct *pr = arg;
    int i;
    //το κάθε process αναλαμβάνει τις γραμμές  i,i + n,i + 2×n,i+3×n,... όπου
n=pcnt=ο αριθμός των διεργασιών
    for(i=pr->mypid; i<y_chars; i+= pr->pcnt){
```

```
        //μπορούν και όλα παράλληλα να κάνουν το compute αλλά το output πρέπει
να γίνεται σειριακά,
        //οπότε εντοπίζουμε σαν κρίσιμο κομμάτι κώδικα το
output_mandel_line(1, color_val);
        compute_mandel_line(i, color_val);
        sem_wait(&sem[pr->mypid]);
        //λέω στον επόμενο σημαφόρο να αυξηθεί η τιμή του κατά 1 και άρα να
περάσει το wait_sem
        //χρησιμοποιούμε το mod διότι όταν φτάσουμε στο τελευταίο
        //όταν φτάσω στο i = mypid του τελευταίου process αυτό (pr->mypid)+1)
δεν υπάρχει
        output_mandel_line(1, color_val);
        sem_post(&sem[((pr->mypid)+1) % (pr->pcnt)]);
        //signal(SIGINT, signal_handler);
    }
}

/*
* 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(...)
    */
    //PROT_READ | PROT_WRITE: Τα memory protection flags.Τα δικαιώματα
πρόσβασης για τον συγκεκριμένο χώρο μνήμης εδώ είναι read και write.
    //MAP_SHARED | MAP_ANONYMOUS: Τα mapping type flags.
    // MAP_SHARED = το mapping είναι διαμοιραζόμενο μεταξύ των
διεργασιών,επιτρέποντας σε πολλές διεργασίες να μπαίνουν και να
επεξεργάζονται τον χώρο μνήμης
    // MAP_ANONYMOUS = λέει ότι το mapping που μόλις κάναμε δεν συνδέεται με
κάποιο αρχείο
-1: The file descriptor. This argument is ignored when MAP_ANONYMOUS is
specified, so a value of -1 is passed.
    //-1: Το fd του υποτιθέμενου αρχείου. Αυτό το argument εδώ αγνοείται γιατί
έχουμε βάλει το MAP_ANONYMOUS
    //0: Το offset που δηλώνει από ποιο σημείο του υποτιθέμενου αρχείου
ξεκινάει το mapping. Εδώ προφανώς δεν μας νοιάζει.
    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);
    }
}


int main(int argc, char *argv[])
{
    int nprocs,i,status ;
    struct process_info_struct *pr;
    pid_t p;
    /*
     * Parse the command line
     */
    if (argc != 2)
        usage(argv[0]);
    //nprocs=ο αριθμός των processes που περνάμε σαν input
    if (safe_atoi(argv[1], &nprocs) < 0 || nprocs <= 0) {
        fprintf(stderr, "`%s' is not valid for `nprocs'\n", argv[1]);
        exit(1);
    }


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

    /*
     * draw the Mandelbrot Set, one line at a time.
     * Output is sent to file descriptor '1', i.e., standard output.
     */
    //πίνακας απο structs, sizeof(*pr) υπολογίζει το μέγεθος ενός
process_info_struct(το sizeof(pr) βρίσκει το μέγεθος ενός pointer)
    pr = create_shared_memory_area(nprocs * sizeof(*pr));
    //πίνακας από σημαφόρους οι οποίοι θα είναι όσοι και οι διεργασίες
    sem = create_shared_memory_area(nprocs * sizeof(sem_t));
    for (i = 0; i < nprocs; i++) {
        pr[i].mypid = i;
        pr[i].pcnt = nprocs;
        //το πρώτο proc είναι το μόνο που θα "μπει" στο sem_wait και
        // γι'αυτό του βάλαμε αρχική τιμή το 1(ώστε το wait να την κάνει
decrease κατά 1)
```

```
        // ενώ τις υπόλοιπες τις αρχικοποιούμε όλες να είναι σε locked state
        //int sem_init(sem_t *sem, int pshared, unsigned int value);
        //εδώ θέλουμε το pshared=1 γιατί ο σημαφορος μοιράζεται μεταξυ πολλών
διεργασιων
        if (i == 0) {
            sem_init(&sem[i], 1, 1); //unlocked-state
        } else {
            sem_init(&sem[i], 1, 0); //locked-state
        }
    }
    for (i = 0; i < nprocs; i++) {
        p = fork();
        if (p < 0) {
            perror("fork");
            exit(1);//die("fork");
        }
        if (p == 0) {
            pr[i].pid = getpid();
            compute_and_output_mandel_line(&pr[i]);
            return 0;
        }
    }
    //βάζουμε τον πατερα να περιμένει ολα του τα παιδια να πεθανουν
    for (i = 0; i < nprocs; i++){
        p= wait(&status);
    }
    for (i = 0; i < nprocs; i++){
        sem_destroy(&sem[i]);
    }
    //ελευθερώνουμε τον χώρο που δημιουργήσαμε
    destroy_shared_memory_area(sem, nprocs * sizeof(sem_t));
    destroy_shared_memory_area(pr, nprocs * sizeof(*pr));

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

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;**

   Εν γένει, το αν θα προτιμήσουμε η παραλληλοποιημένη υλοποίηση να γίνει με νήματα ή με διεργασίες εξαρτάται από το εκάστοτε πρόγραμμα. Στην συγκεκριμένη περίπτωση παρατηρούμε ότι απαιτείται συχνή επικοινωνία και διαμοιρασμός πόρων του ΛΣ, όσο και συχνή πρόσβαση σε κοινή μνήμη. Για αυτές τις περιστάσεις γνωρίζουμε ότι η χρήση νημάτων είναι προτιμητέα σε σχέση με τις διεργασίες.Παρατηρείται σημαντικά μικρότερο κόστος τόσο για την επικοινωνία μεταξύ τους όσο και για την δημιουργία και καταστροφή τους.
   Με την χρήση νημάτων γλυτώνουμε διάφορες ενέργειες που στην περίπτωση των διεργασιών κρίνονται απαραίτητες. Τέτοιες είναι η δημιουργία PCB για κάθε διεργασία, η αφαίρεση του δικαιώματος write απ' το page table ύστερα από κάθε fork(), η αντικατάσταση PCB σε περίπτωση context switch κλπ.

2. **(Προαιρετική) Μπορεί το mmap() interface να χρησιμοποιηθεί για τον διαμοιρασμό μνήμης μεταξύ διεργασιών που δεν έχουν κοινό ancestor; Αν όχι, γιατί;**

   Ναι, το mmap() interface μπορεί να χρησιμοποιηθεί μεταξύ διεργασιών ακόμα και αν δεν έχουν κοινό ancestor. Αυτό που απαιτείται μόνο είναι η χρήση του MAP_SHARED flag που επιτρέπει τον διαμοιρασμό του δημιουργούμενου χώρου μνήμης απ'όλες τις διεργασίες.

1.2.2: Υλοποίηση χωρίς semaphores

```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 <errno.h>
#include <signal.h>
#include <sys/wait.h>


/*TODO header file for m(un)map*/
#include <semaphore.h>
#include <sys/mman.h>

#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;

/*
* 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;
```

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

/*
* A (distinct) instance of this structure
* is passed to each process
*/
struct process_info_struct {
  pid_t pid;

  //int* color_val; /* Pointer to array to manipulate */

  int mypid; /* Application-defined process id */
  int pcnt;
};


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 usage(char *argv0)
{
  fprintf(stderr, "Usage: %s thread_count array_size\n\n"
             "Exactly two argument required:\n"
             "    thread_count: The number of threads to create.\n"
             "    array_size: The size of the array to run with.\n",
        argv0);
  exit(1);
}
/*
* 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; //πίνακας x_chars κελιών για κάθε γραμμή
  }
}

/*
* This function outputs an array of x_char color values
* to a 256-color xterm.
*/
void output_mandel_line(int fd, int *buffer)
{
  int i,j;

  char point ='@';
  char newline='\n';

  //στις προηγούμενες ασκήσεις αυτή η συνάρτηση τύπωνε γραμμή-γραμμή
  //τώρα όλα όσα πρέπει να αποθηκευτούν βρίσκονται στον "δισδιάστατο" buffer
οπότε απλά τυπώνουμε τα δεδομένα του με
  // διπλό for-loop
  //δεν υπάρχει το πρόβλημα της σειράς εκτύπωσης(και άρα της ανάγκης
συγχρονισμού) αφού πλέον αυτή η συνάρτηση
  // καλείται από τον πατέρα όλων των διεργασιών αφότου έχουν πεθάνει όλα τα
παιδιά του
  for(i = 0; i < y_chars; i++) {
     for (j = 0; j < x_chars; j++) {
        /* Set the current color, then output the point */
        set_xterm_color(fd, buffer[(i*x_chars)+j]);
        if (write(fd, &point, 1) != 1) {
           perror("compute_and_output_mandel_line: write point");
           exit(1);
        }
     }
     //βάζουμε την αλλαγή γραμμής
     printf("%c", newline);
  }
  /* 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_store_mandel_line(void *arg, int *buffer)
{
  /*
   * A temporary array, used to hold color values for the line being drawn
   */
  int color_val[x_chars];
  struct process_info_struct *pr = arg;
  int i,j;
```

```
  //πλέον αυτή η συνάρτηση υπολογίζει την κάθε γραμμή και την αποθηκεύει στην
σωστή θέση στον διαμοιραζόμενο μεταξύ
  // των διεργασιών buffer
  //η κάθε διεργασία υπολογίζει διαφορετική γραμμή και αποθηκεύει το
αποτέλεσμά της σε διαφορετικό "μέρος" στον buffer
  //έτσι δεν υπάρχει ο κίνδυνος κάποια διεργασία να κάνει overwrite τους
υπολογισμούς της άλλης
  for(i=pr->mypid; i<y_chars; i+= pr->pcnt) {
     compute_mandel_line(i, color_val);
     for(j=0; j<x_chars; j++){
        buffer[(i*x_chars)+j] = color_val[j];
     }
  }
}

/*
* 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(...)
  */
  //PROT_READ | PROT_WRITE: Τα memory protection flags.Τα δικαιώματα
πρόσβασης για τον συγκεκριμένο χώρο μνήμης εδώ είναι read και write.
  //MAP_SHARED | MAP_ANONYMOUS: Τα mapping type flags.
  // MAP_SHARED = το mapping είναι διαμοιραζόμενο μεταξύ των
διεργασιών,επιτρέποντας σε πολλές διεργασίες να μπαίνουν και να
επεξεργάζονται τον χώρο μνήμης
  // MAP_ANONYMOUS = λέει ότι το mapping που μόλις κάναμε δεν συνδέεται με
κάποιο αρχείο
-1: The file descriptor. This argument is ignored when MAP_ANONYMOUS is
specified, so a value of -1 is passed.
  //-1: Το fd του υποτιθέμενου αρχείου. Αυτό το argument εδώ αγνοείται γιατί
έχουμε βάλει το MAP_ANONYMOUS
  //0: Το offset που δηλώνει από ποιο σημείο του υποτιθέμενου αρχείου
ξεκινάει το mapping. Εδώ προφανώς δεν μας νοιάζει.
  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);
  }
}

int main(int argc, char *argv[])
{
  int line;
  int nprocs,i,status ;
  struct process_info_struct *pr;
  pid_t p;
  int *buf;

  //έλεγχος για λανθασμένη ποσότητα arguments
  if (argc != 2)
     usage(argv[0]);
  //nprocs=ο αριθμός των processes που περνάμε σαν input
  if (safe_atoi(argv[1], &nprocs) < 0 || nprocs <= 0) {
     fprintf(stderr, "`%s' is not valid for `nprocs'\n", argv[1]);
     exit(1);
  }

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

  //πίνακας απο structs, sizeof(*pr) υπολογίζει το μέγεθος ενός
process_info_struct(το sizeof(pr) βρίσκει το μέγεθος ενός pointer)
  pr = create_shared_memory_area(nprocs * sizeof(*pr));
  //διαμοιραζόμενος buffer μεταξύ των διεργασιών
  buf = create_shared_memory_area(y_chars * x_chars * sizeof(int));

  for (i = 0; i < nprocs; i++) {
     pr[i].mypid = i;
     pr[i].pcnt = nprocs;

  }
  for (i = 0; i < nprocs; i++) {
     p = fork();
     if (p < 0) {
        perror("fork");
        exit(1);//die("fork");
     }
```

```
    if (p == 0) {
       pr[i].pid = getpid();
       compute_and_store_mandel_line(&pr[i], buf);
       return 0;
    }
  }
  //βάζουμε τον πατερα να περιμένει ολα του τα παιδια να πεθανουν
  for (i = 0; i < nprocs; i++){
     p = wait(&status);
  }

  //αφού έχει φτιαχτεί ο buffer τον εκτυπώνουμε
  output_mandel_line(1, buf);

  //ελευθερώνουμε τον χώρο που δημιουργήσαμε
  destroy_shared_memory_area(pr, nprocs * sizeof(*pr));
  destroy_shared_memory_area(buf, y_chars * x_chars * sizeof(int));



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

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;

Στην συγκεκριμένη υλοποίηση, παρόλο που έχουμε διαφορετικές διεργασίες που επεξεργάζονται κοινά δεδομένα, δεν τίθεται πρόβλημα συγχρονισμού τους με κάποια απ' τις γνωστές τεχνικές(locks,semaphores κλπ). Αυτό συμβαίνει καθώς ο buffer έχει μέγεθος ώστε να χωρέσει τα δεδομένα(χρώματα κλπ) για όλη την εικόνα. Έτσι, οι διαφορετικές διεργασίες κάνουν παράλληλα το compute και αποθηκεύουν τα αποτελέσματά τους σε διαφορετικό, προκαθορισμένο για την καθεμιά, "μέρος" του buffer. Έτσι, όταν φτάνουμε στο στάδιο της εκτύπωσης, δεν υπάρχει πρόβλημα στην σειρά(και άρα να πρέπει οι διεργασίες να συγχρονοιστούν), καθώς αυτή λαμβάνει χώρα μόνο μέσω μίας διεργασίας, αυτής του αρχικού πατέρα όλων.
Αυτό δεν θα συνέβαινε στην περίπτωση ενός μικρότερου buffer μεγέθους NPROCS x x_chars οπότε η συμπλήρωση της εικόνας και η εκτύπωσή της θα λάμβανε χώρα σε στάδια, πράγμα που προφανώς θα απαιτούσε τον συγχρονισμό των διαφόρων διεργασιών.

Παρατίθεται το Makefile για την άσκηση 1.2:

```
#
# Makefile
#

CC = gcc

# CAUTION: Always use '-pthread' when compiling POSIX threads-based
# applications, instead of linking with "-lpthread" directly.
CFLAGS = -Wall -O2 -pthread
```

```
LIBS =

all: mandel-fork mandel-sem


## Mandel

mandel-fork: mandel-lib.o mandel-fork.o
  $(CC) $(CFLAGS) -o mandel-fork mandel-lib.o mandel-fork.o $(LIBS)

mandel-sem: mandel-lib.o mandel-sem.o
  $(CC) $(CFLAGS) -o mandel-sem mandel-lib.o mandel-sem.o $(LIBS)

mandel-lib.o: mandel-lib.h mandel-lib.c
  $(CC) $(CFLAGS) -c -o mandel-lib.o mandel-lib.c $(LIBS)

mandel-fork.o: mandel-fork.c
  $(CC) $(CFLAGS) -c -o mandel-fork.o mandel-fork.c $(LIBS)

mandel-sem.o: mandel-sem.c
  $(CC) $(CFLAGS) -c -o mandel-sem.o mandel-sem.c $(LIBS)

clean:
  rm -f *.s *.o mandel-fork
  rm -f *.s *.o mandel-sem
```