

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

4η Σειρά Ασκήσεων

Μάθημα: Λειτουργικά Συστήματα (Τμήμα 1°)

Εξάμηνο: 6°

Ονοματεπώνυμο: Αλεξοπούλου Γεωργία, Γκενάκου Ζωή

Στην παρούσα άσκηση θα ασχοληθούμε με τη μελέτη του μηχανισμού της εικονικής μνήμης και με τη χρήση της για αποδοτική διαδιεργασιακή επικοινωνία. Η άσκηση έχει δύο μέρη. Στο πρώτο μέρος μας ζητείται να πειραματιστούμε με κλήσεις συστήματος και να μελετήσετε βασικούς μηχανισμούς του ΛΣ (§ 1.1). Στο δεύτερο μέρος χρησιμοποιούμε την εικονική μνήμη για να μοιράσουμε πόρους μεταξύ διεργασιών (processes) και να υλοποιήσουμε διαδιεργασιακό σχήμα συγχρονισμού (§ 1.2).

Άσκηση 4.1:

Παρακάτω φαίνεται ο πηγαίος κώδικας:

```
/*  
 * 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;
off_t file_size;

/*
 * Child process' entry point.
 */
void child(void)
{
    uint64_t pa;
    /*
     * Step 7 - Child
     */

    if (0 != raise(SIGSTOP))
```

```

        die("raise(SIGSTOP)");

printf("VM Map of Child Process:\n");
show_maps();

/*
 * Step 8 - Child
 */

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

uint64_t virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address((uint64_t)heap_private_buf);

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

printf("VA info of private buffer in Child Process:\n");
show_va_info((uint64_t)heap_private_buf);

/*
 * Step 9 - Child
 */

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

memset(heap_private_buf, '!', buffer_size);

printf("VA info of private buffer in Child Process:\n");
show_va_info((uint64_t)heap_private_buf);

virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address(virt_adr);

```

```

        if (pa != 0) {
            printf("Physical Address of private buffer requested by
Child Process: %lu\n", pa);
        }
        else {
            printf("Physical Address of private buffer requested by
Child Process not found\n");
        }

/*
 * Step 10 - Child
 */

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

memset(heap_shared_buf, '!', buffer_size);

printf("VA info of shared buffer in Child Process:\n");
show_va_info((uint64_t)heap_shared_buf);

virt_adr = (uint64_t)heap_shared_buf;
pa = get_physical_address(virt_adr);

if (pa != 0) {
    printf("Physical Address of shared buffer requested by
Child Process: %lu\n", pa);
}
else {
    printf("Physical Address of shared buffer requested by
Child Process not found\n");
}

/*
 * Step 11 - Child
 */

```

```

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

    mprotect(heap_shared_buf, buffer_size, PROT_READ);

    printf("New VM Map of Child Process:\n");
    show_maps();

    printf("New VA info of shared buffer in Child Process:\n");
    show_va_info((uint64_t)heap_shared_buf);

    /*
     * Step 12 - Child
     */

    if (munmap(heap_private_buf, buffer_size) == -1) {
        die("munmap");
    }

    if (munmap(heap_shared_buf, buffer_size) == -1) {
        die("munmap");
    }

    if (munmap(file_shared_buf, file_size) == -1) {
        die("munmap");
    }

}

/*
 * 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();

printf("VM Map of Parent Process:\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();

uint64_t virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address((uint64_t)heap_private_buf);

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

printf("VA info of private buffer in Parent Process:\n");
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 Process:\n");
show_va_info((uint64_t)heap_private_buf);

virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address(virt_adr);

if (pa != 0) {
    printf("Physical Address of private buffer requested by
Parent Process: %lu\n", pa);
}

else {
    printf("Physical Address of private buffer requested by
Parent Process not found\n");
}

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 Process:\n");
show_va_info((uint64_t)heap_shared_buf);

virt_adr = (uint64_t)heap_shared_buf;
pa = get_physical_address(virt_adr);

if (pa != 0) {
    printf("Physical Address of shared buffer requested by
Parent Process: %lu\n", pa);
}

else {
    printf("Physical Address of shared buffer requested by
Parent Process not found\n");
}

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("New VM Map of Parent Process:\n");
    show_maps();

    printf("New VA info of shared buffer in Parent Process:\n");
    show_va_info((uint64_t)heap_shared_buf);

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

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

    if (munmap(heap_private_buf, buffer_size) == -1) {
        die("munmap");
    }

    if (munmap(heap_shared_buf, buffer_size) == -1) {
        die("munmap");
    }

    if (munmap(file_shared_buf, file_size) == -1) {
        die("munmap");
    }
}

int main(void){

```

```

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

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();

/*
 * 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);
if (heap_private_buf == MAP_FAILED) {
    die("mmap");
}

printf("Mapping:\n");
show_maps();

```

```

/*
 * Step 3: Find the physical address of the first page of your
buffer
 * in main memory. What do you see?
 */

printf(RED "\nStep 3: Find and print the physical address of the
"
        "buffer in main memory. What do you see?\n" RESET);
press_enter();

uint64_t virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address((uint64_t)heap_private_buf);
if (pa != 0) {
    printf("Physical Address: %lu\n", pa);
}

else {
    printf("Physical Address not found\n");
}

/*
 * 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();

memset(heap_private_buf, 0, buffer_size);
virt_adr = (uint64_t)heap_private_buf;
pa = get_physical_address(virt_adr);
if (pa != 0) {
    printf("Physical Address: %lu\n", pa);
}

else {

```

```

        printf("Physical Address not found\n");
    }

    /*
     * 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_size = lseek(fd, 0, SEEK_END);
    if(file_size == -1) {
        die("lseek");
    }

    file_shared_buf = mmap(NULL, file_size, PROT_READ, MAP_SHARED,
fd, 0);
    if (file_shared_buf == MAP_FAILED) {
        die("mmap");
    }

    printf("File Contents: \n%s\n", file_shared_buf);

    printf("\nVirtual Memory Map after memory mapping file.txt:\n");
    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_shared_buf == MAP_FAILED) {
    die("mmap");
}

memset(heap_shared_buf, '!', buffer_size);

printf("New Mapping:\n");
show_maps();

printf("New Physical Address Mapped:\n");
show_va_info((uint64_t)heap_shared_buf);

p = fork();
if (p < 0)
    die("fork");
if (p == 0) {
    child();
    return 0;
}

parent(p);

if (close(fd) == -1)
    perror("close");

```

```
    return 0;  
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται αφενός στα σχόλια, αφετέρου παρατίθεται και εκτενώς στην εκφώνηση της άσκησης.

Χρησιμοποιώντας το Makefile που παρατίθεται από κάτω, γίνεται η μεταγλώττιση και το κατάλληλο linking.

```
.PHONY: all clean  
  
all: mmap  
  
CC = gcc  
CFLAGS = -g -Wall -Wextra -O2  
SHELL= /bin/bash  
  
mmap: mmap.o help.o  
    $(CC) $(CFLAGS) $^ -o $@  
  
%.s: %.c  
    $(CC) $(CFLAGS) -S -fverbose-asm $<  
  
%.o: %.c  
    $(CC) $(CFLAGS) -c $<  
  
%.i: %.c  
    gcc -Wall -E $< | indent -kr > $@  
  
clean:  
    rm -f *.o mmap
```

Παίρνουμε ένα ένα τα βήματα που χρειάστηκαν προκειμένου να συμπληρώσουμε τον κώδικα.

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

- Η πρώτη στήλη αντιπροσωπεύει τις διευθύνσεις έναρξης και λήξης κάθε περιοχής μνήμης με τη μορφή <start_address>-<end_address>.
- Η δεύτερη στήλη δείχνει τα δικαιώματα για κάθε περιοχή (π.χ. r-xp για αναγνώσιμο και εκτελέσιμο).
- Η τρίτη στήλη εμφανίζει το offset της περιοχής.
- Η τέταρτη στήλη παρέχει τον αριθμό της συσκευής και το inode για οποιοδήποτε αρχείο που σχετίζεται με την περιοχή.
- Η πέμπτη στήλη καθορίζει τη διαδρομή και το όνομα του αρχείου (αν υπάρχει).

Step 1: Print the virtual address space map of this process [152602].

```
Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
```

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.

Μετά τη χρήση του `mmap(2)` για να κάνουμε allocate έναν buffer μεγέθους ίσου με 1 σελίδα, ο χάρτης εικονικής μνήμης της διεργασίας έχει ως εξής:

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

Mapping:

Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
```

Με βάση τον χάρτη, ο χώρος των εικονικών διευθύνσεων που δεσμεύσαμε με την `mmap` είναι ο παρακάτω:

```
Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
```

Το εύρος διευθύνσεων `557e3ae6a000-557e3ae70000` αντιστοιχεί στις διευθύνσεις έναρξης και τέλους του δεσμευμένου buffer. Αυτή είναι μια περιοχή μόνο για ανάγνωση (r--p) που ξεκινά με τη διεύθυνση `557e3ae6a000` και τελειώνει σε `557e3ae70000`. Οι επόμενες καταχωρήσεις εμφανίζουν τις σημαίες προστασίας μνήμης (r-xp, r--p, r--p, rw-p) για διαφορετικά τμήματα του buffer.

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[0x7f271e181000] is not mapped; no physical memory allocated.
Physical Address not found
```


Παραπάνω φαίνεται ότι η φυσική διεύθυνση του buffer στην κύρια μνήμη δεν είναι άμεσα προσβάσιμη ή δεν παρέχεται. Το μήνυμα "VA[0x7f271e181000] is not mapped; no physical memory allocated" υποδηλώνει ότι δεν υπάρχει ρητή αντιστοίχιση μεταξύ της εικονικής διεύθυνσης 0x7f271e181000 και μιας φυσικής διεύθυνσης.

Ενώ έχει δεσμευθεί η μνήμη όπως φαίνεται και στον χάρτη μνήμης δεν έχει γίνει ακόμα η απεικόνιση της εικονικής μνήμης στην φυσική. Αυτό συμβαίνει, γιατί η φυσική μνήμη δεσμεύεται on demand από το λειτουργικό σύστημα όταν πάει να προσπελαστεί. Η κατανομή μνήμης on demand, χρησιμοποιείται από λειτουργικά συστήματα για τη βελτιστοποίηση της χρήσης της μνήμης. Αντί να εκχωρείται φυσική μνήμη για όλες τις σελίδες εικονικής μνήμης τη στιγμή της αντιστοίχισης μνήμης, η μνήμη εκχωρείται μόνο όταν γίνεται πρόσβαση ή αναφορά σε αυτήν.

Δηλαδή, όταν πάμε να προσπελάσουμε εικονική μνήμη όπου δεν έχει απεικονιστεί ακόμα, θα γίνει page fault και αφού διαπιστωθεί ότι η εικονική μνήμη βρίσκεται στον χάρτη μνήμης τότε το λειτουργικό σύστημα θα επιλέξει ένα frame για να γίνει η απεικόνιση. Όταν προκύπτει page fault, το λειτουργικό σύστημα πρέπει να εκχωρήσει ένα physical frame στην κύρια μνήμη για να αποθηκεύσει τη σελίδα που λείπει.

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: 5787041792
```

Αφού γεμίσαμε το buffer με μηδενικά, παρατηρήσατε μια φυσική διεύθυνση: 5787041792. Αυτό υποδηλώνει ότι μετά την προετοιμασία του buffer, το λειτουργικό σύστημα εκχώρησε φυσική μνήμη στην αντίστοιχη εικονική διεύθυνση.

Η αλλαγή που παρατηρήσατε είναι ότι το page fault από το Βήμα 3 ενεργοποίησε την εκχώρηση φυσικής μνήμης για την εικονική διεύθυνση 0x7f271e181000. Ως αποτέλεσμα, η φυσική διεύθυνση 5787041792 αντιπροσωπεύει τώρα την αντιστοίχιση του buffer στην κύρια μνήμη.

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

Μετά τη χρήση του mmap(2) για τη χαρτογράφηση μνήμης του file.txt στο χώρο διευθύνσεων της διεργασίας και την εκτύπωση των περιεχομένων του, οι νέες πληροφορίες αντιστοίχισης εμφανίζονται στον χάρτη μνήμης ως εξής:

```
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
```

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

File Contents:
Hello everyone!

Virtual Memory Map after memory mapping file.txt:

```
Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
```

Αυτό υποδηλώνει ότι το file.txt έχει αντιστοιχιστεί στη μνήμη στο εύρος εικονικών διευθύνσεων 0x7f271e157000-0x7f271e158000 στο χώρο διευθύνσεων της διεργασίας. Η αντιστοίχιση είναι μόνο για ανάγνωση (r--) και αντιπροσωπεύει ένα shared mapping. Το ίδιο το file.txt βρίσκεται στη διαδρομή /home/oslab/oslab33/ex4/4.1/file.txt.

Αυτό το mapping μας επιτρέπει να έχουμε απευθείας πρόσβαση και να διαβάζουμε τα περιεχόμενα του file.txt σαν να ήταν μέρος της μνήμης της διαδικασίας.

6. Χρησιμοποιείτε 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.

Όταν χρησιμοποιείται το mmap(2) για το allocation ενός κοινόχρηστου buffer, το λειτουργικό σύστημα δημιουργεί μια αντιστοίχιση μεταξύ μιας περιοχής εικονικών διευθύνσεων στο χώρο διευθύνσεων της διεργασίας και μιας περιοχής φυσικής μνήμης. Σε αυτήν την περίπτωση, η νέα αντιστοίχιση εμφανίζεται ως:

```
New Physical Address Mapped:
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
```

Ακολουθεί μια ανάλυση των πληροφοριών σε αυτήν τη γραμμή:

- 7f271e156000-7f271e157000: Αυτό το εύρος εικονικών διευθύνσεων αντιπροσωπεύει την περιοχή μνήμης που αντιστοιχίζεται στην κοινόχρηστη προσωρινή μνήμη. Η διεργασία μπορεί να έχει πρόσβαση και να χειριστεί τα περιεχόμενα του buffer χρησιμοποιώντας αυτές τις εικονικές διευθύνσεις.

- **rw-s:** Αυτό υποδηλώνει ότι η περιοχή μνήμης είναι αναγνώσιμη και εγγράψιμη (rw) και μοιράζεται (s-shared) μεταξύ πολλαπλών διεργασιών. Αυτό σημαίνει ότι οποιεσδήποτε αλλαγές γίνονται στο buffer από μία διεργασία θα είναι ορατές σε άλλες διεργασίες που έχουν επίσης αντιστοιχίσει την ίδια κοινόχρηστη προσωρινή μνήμη.
- **00000000:** Αυτό το πεδίο αντιπροσωπεύει το offset εντός του αρχείου ή της συσκευής που αντιστοιχίζεται. Σε αυτήν την περίπτωση, η μετατόπιση είναι 0, υποδεικνύοντας ότι η αντιστοίχιση δεν βασίζεται σε αρχείο αλλά στη συσκευή /dev/zero.
- **00:01 4166:** Αυτό το τμήμα της γραμμής παρέχει πρόσθετες πληροφορίες σχετικά με την αντιστοιχισμένη περιοχή. Το 00:01 υποδεικνύει τον κύριο και τον δευτερεύοντα αριθμό της συσκευής και το 4166 αντιπροσωπεύει τον αριθμό εισόδου του αρχείου ή της συσκευής. Σε αυτήν την περίπτωση, η αντιστοίχιση σχετίζεται με τη συσκευή /dev/zero.

Η συσκευή /dev/zero είναι μια ειδική συσκευή που παρέχει απεριόριστη παροχή μηδενικών byte κατά την ανάγνωση από buffer. Χρησιμοποιείται συνήθως για την εκχώρηση περιοχών μνήμης με μηδενική προετοιμασία. Σε αυτό το σενάριο, το κοινό buffer εκχωρείται χρησιμοποιώντας τη συσκευή /dev/zero ως πηγή των περιεχομένων του buffer.

Με τη χρήση αυτής της κοινής αντιστοίχισης buffer, πολλαπλές διεργασίες μπορούν να έχουν πρόσβαση και να μοιράζονται την ίδια περιοχή φυσικής μνήμης, επιτρέποντας την αποτελεσματική επικοινωνία μεταξύ διεργασιών και την κοινή χρήση δεδομένων.

```
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.

New Mapping:

Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3e370000-557e3e391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]

-----
New Physical Address Mapped:
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
```

Σε αυτό το σημείο στο πρόγραμμα καλείται η `fork()` και δημιουργείται μια νέα διεργασία.

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.

VM Map of Parent Process:

Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----

VM Map of Child Process:

Virtual Memory Map of process [152614]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
```

Εξετάζοντας τους χάρτες εικονικής μνήμης της γονικής διαδικασίας και της διαδικασίας-παιδί μετά την κλήση `fork()`, μπορούμε να παρατηρήσουμε τα εξής:

1. Οι χάρτες εικονικής μνήμης τόσο της γονικής όσο και της θυγατρικής διαδικασίας είναι πανομοιότυποι. Οι αντιστοιχίσεις του κοινόχρηστου `buffer`, των βιβλιοθηκών, του εκτελέσιμου και άλλων περιοχών είναι ίδιες και στις δύο διεργασίες. Αυτό συμβαίνει επειδή η κλήση συστήματος `fork()` δημιουργεί μια διεργασία-παιδί που είναι ακριβές αντίγραφο της γονικής διαδικασίας, συμπεριλαμβανομένης της εικονικής της μνήμης.

2. Οι περιοχές μνήμης που επισημαίνονται ως [heap] αντιπροσωπεύουν τη δυναμικά εκχωρημένη μνήμη για κάθε διεργασία. Είναι ξεχωριστά και δεν μοιράζονται μεταξύ των διαδικασιών γονέα και θυγατρικού. Κάθε διεργασία έχει τη δική της περιοχή σωρού για δυναμική κατανομή μνήμης.
3. Οι περιοχές μνήμης που σχετίζονται με τις κοινόχρηστες βιβλιοθήκες μοιράζονται επίσης μεταξύ των διεργασιών γονέα και θυγατρικής. Αυτό συμβαίνει επειδή οι κοινόχρηστες βιβλιοθήκες φορτώνονται στη μνήμη μόνο μία φορά και μοιράζονται σε πολλαπλές διεργασίες, γεγονός που βοηθά στην αποτελεσματική χρήση της μνήμης.
4. Η περιοχή μνήμης που σχετίζεται με το κοινόχρηστο αρχείο (file.txt) είναι επίσης κοινόχρηστη μεταξύ των γονικών και θυγατρικών διεργασιών. Τυχόν τροποποιήσεις που πραγματοποιούνται σε αυτήν την περιοχή κοινόχρηστου αρχείου από μια διεργασία θα είναι ορατές στην άλλη διαδικασία.

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 of private buffer requested by parent: 5787041792
VA info of private buffer in Parent Process:
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by child: 5787041792
VA info of private buffer in Child Process:
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
```

Με βάση την παρεχόμενη έξοδο, φαίνεται ότι η φυσική διεύθυνση του private buffer τόσο για τη γονική όσο και για τη θυγατρική διεργασία είναι 5787041792.

Αμέσως μετά την κλήση συστήματος «fork()», δημιουργείται η διεργασία-παιδί ως αντίγραφο της γονικής διεργασίας. Η διεργασία-παιδί κληρονομεί τη διάταξη εικονικής μνήμης, συμπεριλαμβανομένου του private buffer, από τη γονική διεργασία. Η φυσική διεύθυνση του ιδιωτικού buffer παραμένει η ίδια τόσο για τη γονική όσο και για τη διεργασία-παιδί.

Στην έξοδο, το εύρος εικονικών διευθύνσεων `7f271e181000-7f271e182000` αντιστοιχεί στο ιδιωτικό buffer τόσο στη γονική όσο και στη διεργασία-παιδί. Ωστόσο, είναι σημαντικό να σημειωθεί ότι η εικονική διεύθυνση δεν παρέχει άμεσα πληροφορίες σχετικά με τη φυσική διεύθυνση. Η φυσική διεύθυνση είναι η πραγματική θέση των δεδομένων στην κύρια μνήμη (RAM).

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?

VA info of private buffer in Parent Process:
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by Parent Process: 5787041792
VA info of private buffer in Child Process:
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
Physical Address of private buffer requested by Child Process: 5258579968
```


Μετά την εγγραφή στο ιδιωτικό buffer από τη διεργασία-παιδί, το physical address του private buffer αλλάζει για τη διεργασία-παιδί.

Στην αρχική έξοδο του Βήματος 8, το physical address του private buffer τόσο για τη γονική όσο και για τη διεργασία-παιδί ήταν 5787041792. Ωστόσο, μετά την εγγραφή στο private buffer από τη διεργασία-παιδί, στο Βήμα 9, το physical address του private buffer για τη διεργασία-παιδί γίνεται 5258579968.

Ο λόγος για αυτήν την αλλαγή είναι η συμπεριφορά Copy-on-Write (COW). Όταν η διεργασία-παιδί εγγράφει στο private buffer, το λειτουργικό σύστημα δημιουργεί ένα ξεχωριστό αντίγραφο της σελίδας που περιέχει τα τροποποιημένα δεδομένα. Αυτό το αντίγραφο είναι συγκεκριμένο για τη διεργασία-παιδί και επιτρέπει στη διεργασία-παιδί να έχει το δικό της ανεξάρτητο αντίγραφο των τροποποιημένων δεδομένων. Ως αποτέλεσμα, το physical address του private buffer για τη θυγατρική διαδικασία αλλάζει για να αντικατοπτρίζει το νέο αντίγραφο των τροποποιημένων δεδομένων.

Είναι σημαντικό να σημειωθεί ότι η φυσική διεύθυνση του ιδιωτικού buffer για τη γονική διαδικασία παραμένει η ίδια, καθώς η γονική διαδικασία δεν τροποποίησε τα δεδομένα. Ο μηχανισμός COW διασφαλίζει ότι οι διεργασίες γονέα και θυγατρικές μοιράζονται αρχικά τις ίδιες σελίδες φυσικής μνήμης, αλλά μόλις μία από αυτές τροποποιήσει τα δεδομένα, δημιουργείται ένα ξεχωριστό αντίγραφο για τη διατήρηση της απομόνωσης της διαδικασίας.

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

```
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 Process:  
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)  
Physical Address of shared buffer requested by Parent Process: 5133127680  
VA info of shared buffer in Child Process:  
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)  
Physical Address of shared buffer requested by Child Process: 5133127680
```

Κατά την εγγραφή στο κοινό buffer από τη θυγατρική διεργασία, η φυσική διεύθυνση του κοινόχρηστου buffer παραμένει η ίδια τόσο για τη γονική όσο και για τη διεργασία-παιδί.

Στο Βήμα 10, η φυσική διεύθυνση του κοινόχρηστου buffer τόσο για τη γονική όσο και για τη διεργασία-παιδί είναι 5133127680 και τα εύρη εικονικών διευθύνσεων είναι επίσης τα ίδια.

Σε αντίθεση με το ιδιωτικό buffer, το οποίο παρουσίαζε συμπεριφορά Copy-on-Write (COW), το κοινό buffer είναι προσβάσιμο τόσο από τη γονική όσο και από τη θυγατρική διεργασία χωρίς να δημιουργεί ξεχωριστά αντίγραφα των δεδομένων. Και οι δύο διεργασίες μοιράζονται τις ίδιες σελίδες φυσικής μνήμης για το κοινό buffer. Αυτή η συμπεριφορά αναμένεται για περιοχές κοινής μνήμης. Τυχόν τροποποιήσεις που γίνονται από μια διεργασία στην κοινόχρηστη προσωρινή μνήμη είναι άμεσα ορατές στην άλλη διεργασία, καθώς και οι δύο έχουν πρόσβαση στην ίδια θέση φυσικής μνήμης. Κατά συνέπεια, η φυσική διεύθυνση του κοινόχρηστου buffer παραμένει αμετάβλητη τόσο για τη γονική όσο και για τη θυγατρική διαδικασία.

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.

New VM Map of Parent Process:

```
Virtual Memory Map of process [152602]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e156000-7f271e157000 r--s 00000000 00:01 4166 /dev/zero (deleted)
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
```

New VA info of shared buffer in Parent Process:

```
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166 /dev/zero (deleted)
```

New VM Map of Child Process:

```
Virtual Memory Map of process [152614]:
557e3ae6a000-557e3ae6b000 r--p 00000000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6b000-557e3ae6c000 r-xp 00001000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6c000-557e3ae6d000 r--p 00002000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6e000-557e3ae6f000 r--p 00003000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3ae6f000-557e3ae70000 rw-p 00004000 00:26 7619875 /home/oslab/oslab33/ex4/4.1/mmap
557e3c370000-557e3c391000 rw-p 00000000 00:00 0 [heap]
7f271df7d000-7f271df9f000 r--p 00000000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271df9f000-7f271e0f8000 r-xp 00022000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e0f8000-7f271e147000 r--p 0017b000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e147000-7f271e14b000 r--p 001c9000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14b000-7f271e14d000 rw-p 001cd000 fe:01 144567 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f271e14d000-7f271e153000 rw-p 00000000 00:00 0
7f271e156000-7f271e157000 r--s 00000000 00:01 4166 /dev/zero (deleted)
7f271e157000-7f271e158000 r--s 00000000 00:26 7618868 /home/oslab/oslab33/ex4/4.1/file.txt
7f271e158000-7f271e159000 r--p 00000000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e159000-7f271e179000 r-xp 00001000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e179000-7f271e181000 r--p 00021000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e181000-7f271e182000 rw-p 00000000 00:00 0
7f271e182000-7f271e183000 r--p 00029000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e183000-7f271e184000 rw-p 0002a000 fe:01 144563 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f271e184000-7f271e185000 rw-p 00000000 00:00 0
7ffdd16b5000-7ffdd16d6000 rw-p 00000000 00:00 0 [stack]
7ffdd16ee000-7ffdd16f2000 r--p 00000000 00:00 0 [vvar]
7ffdd16f2000-7ffdd16f4000 r-xp 00000000 00:00 0 [vdso]
-----
```

New VA info of shared buffer in Child Process:

```
7f271e156000-7f271e157000 r--s 00000000 00:01 4166 /dev/zero (deleted)
```

Από την παραπάνω έξοδο απομονώνουμε δύο κομμάτια που μας επιβεβαιώνουν την απαγόρευση για εγγραφή στον shared buffer για την διεργασία παιδί.

```
New VA info of shared buffer in Parent Process:
7f271e156000-7f271e157000 rw-s 00000000 00:01 4166
```

```
/dev/zero (deleted)
```

```
New VA info of shared buffer in Child Process:
7f271e156000-7f271e157000 r--s 00000000 00:01 4166 /dev/zero (deleted)
oslab33@os-node2:~/ex4/4.1$
```

Φαίνεται ότι οι δύο διεργασίες έχουν πλέον διαφορετικά δικαιώματα πάνω στον shared buffer. Αυτή η αλλαγή υποδεικνύει ότι οι εγγραφές στο κοινόχρηστο buffer έχουν απενεργοποιηθεί για τη διεργασία παιδί. Το δικαίωμα "r--s" στον χάρτη μνήμης της διεργασίας παιδί σημαίνει ότι η κοινόχρηστη προσωρινή μνήμη είναι πλέον μόνο για ανάγνωση για την διεργασία αυτή, αποτρέποντας τυχόν λειτουργίες εγγραφής.

12. Αποδεσμεύστε όλους τους buffers στις δύο διεργασίες.

Βλέπουμε και στον πηγαίο κώδικα, ότι για το Βήμα 12 αποδεσμεύουμε όλους τους buffers στις δύο εργασίες με την χρήση της munmap().

Άσκηση 4.2:

Στην άσκηση αυτή, τροποποιούμε το πρόγραμμα Mandelbrot set, από την άσκηση 3, ώστε αντί να χρησιμοποιούμε νήματα για την παραλληλοποίηση του υπολογισμού του, να χρησιμοποιούμε διεργασίες.

4.2.1 Semaphores πάνω από διαμοιραζόμενη μνήμη

Όταν χρησιμοποιούνται διεργασίες για παραλληλοποίηση αντί για νήματα, ένα πρόβλημα που προκύπτει σχετικά με την κοινόχρηστη μνήμη είναι ότι κάθε διεργασία έχει το δικό της ξεχωριστό χώρο διευθύνσεων. Σε αντίθεση με τα νήματα, τα οποία μοιράζονται τον ίδιο χώρο διευθύνσεων και μπορούν να έχουν άμεση πρόσβαση σε κοινόχρηστες μεταβλητές, οι διεργασίες έχουν τον δικό τους ανεξάρτητο χώρο μνήμης.

Αυτό σημαίνει ότι εάν θέλουμε να μοιράζουμε δεδομένα μεταξύ διεργασιών, πρέπει να χρησιμοποιήσουμε μηχανισμούς επικοινωνίας μεταξύ διεργασιών. Για παράδειγμα, η κοινόχρηστη μνήμη επιτρέπει σε πολλές διεργασίες να έχουν πρόσβαση σε μια κοινή περιοχή της μνήμης. Αυτή η περιοχή μπορεί να χρησιμοποιηθεί για την αποτελεσματική κοινή χρήση δεδομένων μεταξύ των διαδικασιών. Ωστόσο, δεδομένου ότι οι διεργασίες έχουν ξεχωριστούς χώρους διευθύνσεων, πρέπει να αντιστοιχίσουν την περιοχή κοινόχρηστης μνήμης στον δικό τους χώρο διευθύνσεων για πρόσβαση σε αυτήν.

Όταν χρησιμοποιούμε κοινόχρηστη μνήμη, πρέπει να διασφαλίσουμε τους κατάλληλους μηχανισμούς συγχρονισμού για να αποφύγουμε τα race conditions και να διασφαλίσουμε την ακεραιότητα των δεδομένων. Τεχνικές όπως οι σημαφόροι, που χρησιμοποιούμε σε αυτή την άσκηση, μπορούν να χρησιμοποιηθούν για τον έλεγχο της πρόσβασης στην κοινόχρηστη μνήμη και την αποφυγή συγκρούσεων.

Είναι σημαντικό να σημειωθεί ότι η χρήση διεργασιών για παραλληλοποίηση αντί για νήματα μπορεί να έχει επιπλέον επιβάρυνση λόγω της ανάγκης για επικοινωνία μεταξύ των διεργασιών. Ωστόσο, οι διαδικασίες παρέχουν καλύτερη απομόνωση και ανοχή σφαλμάτων σε σύγκριση με τα νήματα.

Ο πηγαίος κώδικος 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 "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

sem_t *mySem;

/*
 * 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 function to convert string to integer
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;
}

/*
 * 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);
    }
}

```

```

// Function to display usage information
void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s processes_count\n\n"
        "Exactly one argument required:\n"
        "        processes_count: The number of processes 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 signal_handler(int sign)
{
    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;
}

// Destroy the shared memory area
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);
    }
}

/*
 * This function computes and outputs a Mandelbrot line for a given
 * process.
 * Each process is responsible for a different set of lines.
 */
void compute_and_output_mandel_line(int line, int procnt){

    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(&mySem[line]) < 0) {
            perror("sem_wait");
            exit(1);
        }
        output_mandel_line(1, color_val);
        if (sem_post(&mySem[(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 = signal_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
`processes_count'\n", argv[1]);
    exit(1);
}

mySem = create_shared_memory_area(procnt * sizeof(sem_t)); //
Allocate shared memory to store the semaphore array

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

if (sem_post(&mySem[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) {
            compute_and_output_mandel_line(i, procnt);
            exit(1);
        }
    }
    for (i = 0; i < procnt; i++) {
        child_pid = wait(&status);
    }

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

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

    reset_xterm_color(1);
    return 0;
}

```

Η λειτουργία του κώδικα φαίνεται από τα επεξηγηματικά σχόλια του κώδικα.

Πιο συγκεκριμένα, ο συγχρονισμός των διεργασιών με χρήση σηματοφόρων και ο χειρισμός της κοινόχρηστης μνήμης υλοποιούνται στις ακόλουθες ενότητες:

- Χειρισμός κοινής μνήμης:

Η περιοχή κοινόχρηστης μνήμης δημιουργείται χρησιμοποιώντας τη συνάρτηση `create_shared_memory_area()`. Καλεί το `mmap()` για να εκχωρήσει μια κοινόχρηστη περιοχή μνήμης συγκεκριμένου μεγέθους.

Η περιοχή κοινόχρηστης μνήμης καταστρέφεται χρησιμοποιώντας τη συνάρτηση `destroy_shared_memory_area()`, η οποία καλεί τη `munmap()` για να κατανείμει την περιοχή κοινής μνήμης.

- Χειρισμός σηματοφόρου:

Οι σηματοφόροι αρχικοποιούνται στη συνάρτηση `main()` χρησιμοποιώντας `sem_init()`. Αποθηκεύονται στον πίνακα `mySem`, ο οποίος εκχωρείται στην περιοχή κοινόχρηστης μνήμης.

Οι λειτουργίες σηματοφόρου εκτελούνται στη συνάρτηση `compute_and_output_mandel_line()`. Η `sem_wait()` χρησιμοποιείται για την απόκτηση του σηματοφόρου πριν από τον υπολογισμό και την έξοδο μιας γραμμής και η `sem_post()` χρησιμοποιείται για την απελευθέρωση του σηματοφορέα μετά την επεξεργασία της γραμμής.

Η καταστροφή του σηματοφόρου γίνεται στη συνάρτηση `main()` χρησιμοποιώντας `sem_destroy()`.

Επομένως, ο χειρισμός της κοινόχρηστης μνήμης γίνεται κυρίως στις συναρτήσεις `create_shared_memory_area()` και `struct_shared_memory_area()`, ενώ ο χειρισμός σηματοφορέα γίνεται κυρίως στις συναρτήσεις `main()` και `compute_and_output_mandel_line()`.

Για την μεταγλώτιση του προγράμματος και το linking, χρησιμοποιούμε το παρακάτω Makefile, που μεταγλωττίζει και το πρόγραμμα της επόμενης άσκησης:

```
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-fork-sem

## Mandel

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

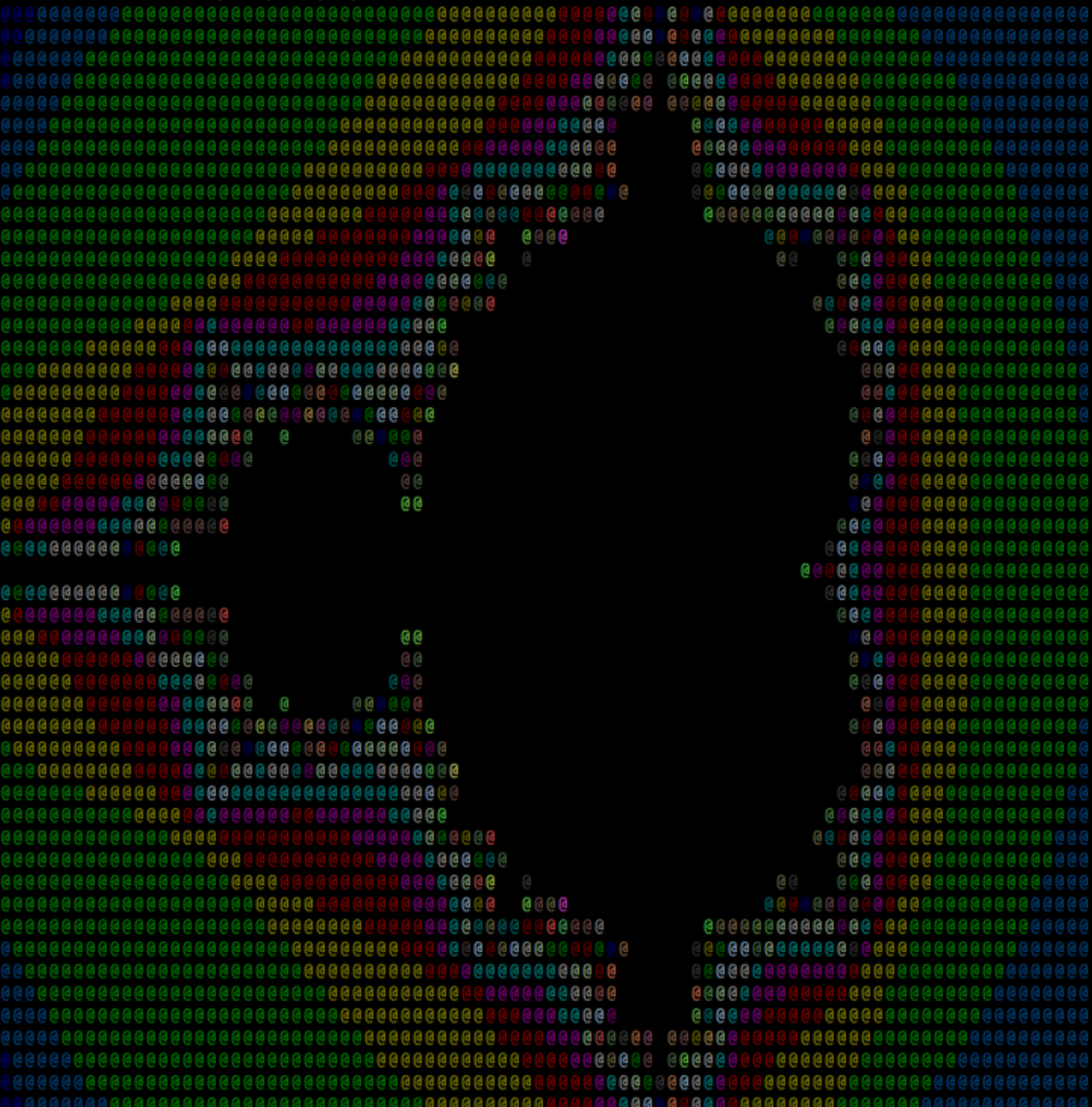
mandel-fork-sem: mandel-lib.o mandel-fork-sem.o
    $(CC) $(CFLAGS) -o mandel-fork-sem mandel-lib.o
mandel-fork-sem.o $(LIBS)
```

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

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

Μετά την μεταγλώττιση, εκτελούμε το πρόγραμμα:

```
oslab33@os-node1: ~/ex4/4.2
oslab33@os-node1:~/ex4/4.2$ ./mandel-fork-sem 50
```



```
oslab33@os-node1:~/ex4/4.2$
```

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

Η επιλογή μεταξύ της χρήσης νημάτων ή διεργασιών για παραλληλοποίηση εξαρτάται από διάφορους παράγοντες, συμπεριλαμβανομένων των ειδικών απαιτήσεων του προγράμματος, της φύσης του φόρτου εργασίας και της υποκείμενης αρχιτεκτονικής υλικού.

Στην περίπτωση ενός προγράμματος συνόλου Mandelbrot, το οποίο περιλαμβάνει υπολογιστικές τιμές για μεμονωμένα pixel του συνόλου, η διαφορά απόδοσης μεταξύ της χρήσης νημάτων και διεργασιών μπορεί να είναι πιο διαφοροποιημένη.

- Παραλληλοποίηση βάσει νημάτων: Η χρήση νημάτων για παραλληλοποίηση επιτρέπει την πρόσβαση σε κοινόχρηστη μνήμη, η οποία μπορεί να είναι επωφελής στο πλαίσιο του υπολογισμού του συνόλου Mandelbrot. Κάθε νήμα μπορεί να λειτουργήσει σε ένα συγκεκριμένο υποσύνολο εικονοστοιχείων και να ενημερώσει απευθείας το κοινόχρηστο buffer εικόνας, χωρίς την ανάγκη ρητής επικοινωνίας μεταξύ νημάτων. Αυτή η άμεση πρόσβαση στην κοινόχρηστη μνήμη μπορεί να οδηγήσει σε αποτελεσματική κοινή χρήση και συγχρονισμό δεδομένων, οδηγώντας ενδεχομένως σε καλύτερη απόδοση σε σύγκριση με την παραλληλοποίηση βάσει διεργασιών.
- Παραλληλισμός βάσει διεργασιών: Κατά τη χρήση διεργασιών, κάθε διεργασία θα έχει το δικό της αντίγραφο του buffer εικόνας στον ξεχωριστό χώρο διευθύνσεων. Αυτό σημαίνει ότι κάθε διεργασία θα υπολογίζει το τμήμα της εικόνας ανεξάρτητα και θα αποθηκεύει τα αποτελέσματα στο δικό της χώρο μνήμης. Για να συνδυαστούν τα αποτελέσματα από όλες τις διεργασίες σε μια τελική εικόνα, είναι απαραίτητη η επικοινωνία μεταξύ των διεργασιών. Σε αυτήν την περίπτωση, η χρήση κοινόχρηστης μνήμης και σηματοφόρων μπορεί να διευκολύνει τον συντονισμό και τον συγχρονισμό των διαδικασιών κατά την ενημέρωση του κοινόχρηστου buffer εικόνας.

Ωστόσο, η επιβάρυνση της επικοινωνίας και του συγχρονισμού μεταξύ των διεργασιών μπορεί να επηρεάσει την απόδοση της προσέγγισης παραλληλοποίησης που βασίζεται στη διαδικασία. Η απόκτηση και η απελευθέρωση σηματοφορέων για πρόσβαση σε κοινόχρηστη μνήμη εισάγει πρόσθετο λανθάνοντα χρόνο σε σύγκριση με την άμεση πρόσβαση στη μνήμη στην παραλληλοποίηση βασισμένη σε νήματα. Επιπλέον, μπορεί να προκύψει διαμάχη για τους κοινόχρηστους σηματοφόρους, οδηγώντας σε πιθανές καθυστερήσεις καθώς οι διαδικασίες περιμένουν την πρόσβαση στην κοινόχρηστη μνήμη.

Για να εκτιμηθεί με ακρίβεια η διαφορά απόδοσης μεταξύ παραλληλισμού βάσει νήματος και παραλληλισμού βάσει διεργασιών για το πρόγραμμα συνόλου Mandelbrot, θα ήταν απαραίτητο να

ληφθούν υπόψη παράγοντες όπως ο αριθμός των νημάτων ή των διαδικασιών που χρησιμοποιούνται, το μέγεθος της εικόνας, οι συγκεκριμένες λεπτομέρειες υλοποίησης (π. , την ευαισθησία της διαίρεσης εργασίας) και τα χαρακτηριστικά της πλατφόρμας υλικού.

Γενικά, η παραλληλοποίηση βάσει νημάτων είναι συχνά πιο κατάλληλη για λεπτομερείς παράλληλες εργασίες, όπου απαιτείται συχνή και αποτελεσματική κοινή χρήση δεδομένων. Ωστόσο, η παραλληλοποίηση βάσει διεργασιών μπορεί να προσφέρει πλεονεκτήματα όσον αφορά την ανοχή σφαλμάτων και την απομόνωση. Για υπολογιστικά εντατικές εργασίες όπως ο υπολογισμός του συνόλου Mandelbrot, όπου ο φόρτος εργασίας μπορεί να χωριστεί ομοιόμορφα σε ανεξάρτητες περιοχές, η παραλληλοποίηση βάσει νημάτων ευνοείται συνήθως λόγω της χαμηλότερης επιβάρυνσης και των πιο αποτελεσματικών δυνατοτήτων κοινής χρήσης δεδομένων.

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

Όχι, η mmap() δεν μπορεί να χρησιμοποιηθεί για την κατανομή της μνήμης μεταξύ διεργασιών που δεν έχουν κοινό ancestor.

Το mmap() είναι μια συνάρτηση που χρησιμοποιείται για τη χαρτογράφηση μνήμης, η οποία επιτρέπει σε μια διαδικασία να αντιστοιχίσει ένα αρχείο ή μια συσκευή στον εικονικό χώρο διευθύνσεών της. Δημιουργεί μια αντιστοίχιση μεταξύ μιας περιοχής της εικονικής μνήμης της διεργασίας και του καθορισμένου αρχείου ή συσκευής. Η αντιστοιχισμένη περιοχή μνήμης μπορεί να μοιράζεται μεταξύ πολλών νημάτων εντός της ίδιας διεργασίας, αλλά δεν μπορεί να κοινοποιηθεί απευθείας μεταξύ άσχετων διεργασιών.

Όταν πολλές διεργασίες χρειάζεται να μοιράζονται τη μνήμη, συνήθως χρησιμοποιούν μηχανισμούς επικοινωνίας μεταξύ διεργασιών, όπως κοινόχρηστη μνήμη. Αυτοί οι μηχανισμοί επιτρέπουν στις διαδικασίες να ανταλλάσσουν δεδομένα και να συγχρονίζουν τις δραστηριότητές τους.

Συνοψίζοντας, ενώ η mmap() είναι χρήσιμη για αντιστοίχιση μνήμης μέσα σε μια μεμονωμένη διεργασία ή μεταξύ νημάτων της ίδιας διεργασίας, δεν μπορεί να χρησιμοποιηθεί απευθείας για την κοινή χρήση μνήμης μεταξύ άσχετων διεργασιών. Η κοινόχρηστη μνήμη, μαζί με κατάλληλους μηχανισμούς συγχρονισμού, χρησιμοποιείται συνήθως για την κοινή χρήση μνήμης μεταξύ διεργασιών.

4.2.2 Υλοποίηση χωρίς semaphores

Σε αυτή την άσκηση, μας ζητείται να αποφύγουμε την χρήση σηματοφόρων. Επομένως, απλά δεσμεύουμε έναν πίνακα μεγέθους όσο αυτού που θέλουμε να τυπώσουμε. Με χρήση κοινής μνήμης θα δημιουργήσουμε ένα buffer που κρατάει ότι υπολογίζει η κάθε διεργασία. Στην συνέχεια, δημιουργούμε

τις διεργασίες και η κάθε μια από αυτές, υπολογίζει και γεμίζει την αντίστοιχη γραμμή στο buffer που της αντιστοιχεί. Τελικά η αρχική διαδικασία αναλαμβάνει να τυπώσει τον δισδιάστατο buffer.

Ο πηγαίος κώδικας 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 "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

int **buffer; // 2D Mandelbrot set output

// Output at the terminal 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 function to safely convert string to integer
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;
}

/*
 * 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;

    // 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;

        // Store the color value 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);
        }
    }

    // Output a newline character to mark the end of the line
    if (write(fd, &newline, 1) != 1) {
        perror("compute_and_output_mandel_line: write newline");
        exit(1);
    }
}

// Print usage information for the program
void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s processes_count\n\n"
        "Exactly one argument required:\n"

```

```

        "        processes_count: The number of processes
to create.\n",
        argv0);
    exit(1);
}

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

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

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

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

// Destroy a shared memory area

```



```

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);
    }
}

// Signal handler to catch SIGINT (Ctrl-C)
void signal_handler(int signum)
{
    reset_xterm_color(1);
    exit(1);
}

// Compute and output the Mandelbrot line for a given process
void compute_and_output_mandel_line(int line, int proccount)
{
    int line_count;
    // Each process writes to the buffer
    for (line_count = line; line_count < y_chars; line_count +=
proccount) {
        compute_mandel_line(line_count, buffer[line_count]);
    }
    return;
}

```

```

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

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

    // Check command-line arguments
    if (argc != 2)
        usage(argv[0]);
    if (safe_atoi(argv[1], &proccount) < 0 || proccount <= 0) {
        fprintf(stderr, "`%s' is not valid for `processes_count'\n",
argv[1]);
        exit(1);
    }

    /*
     * Signal handling: Catch SIGINT (Ctrl-C) to ensure the prompt is
not
     * drawn in a funny color if the user "terminates" the execution
with Ctrl-C.
     */
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGINT, &sa, NULL) < 0) {
        perror("sigaction");
        exit(1);
    }

    // Create shared memory for the buffer
    buffer = create_shared_memory_area(y_chars * sizeof(int *));
    for (i = 0; i < y_chars; i++) {
        buffer[i] = create_shared_memory_area(x_chars * sizeof(int));
    }

    // Create processes and call the execution function
    pid_t child_pid;

```

```

for (i = 0; i < proccount; i++) {
    child_pid = fork();
    if (child_pid < 0) {
        perror("error with creation of child");
        exit(1);
    }
    if (child_pid == 0) {
        compute_and_output_mandel_line(i, proccount);
        exit(1);
    }
}

// Wait for all child processes to finish
for (i = 0; i < proccount; i++) {
    child_pid = wait(&status);
}

// Output the Mandelbrot set line by line
for (i = 0; i < y_chars; i++) {
    output_mandel_line(1, buffer[i]);
}

// Free shared memory resources
for (i = 0; i < y_chars; i++) {
    destroy_shared_memory_area(buffer[i], sizeof(buffer[i]));
}
destroy_shared_memory_area(buffer, sizeof(buffer));

// Reset terminal color and exit
reset_xterm_color(1);
return 0;
}

```

Ο κώδικας επεξηγείται στα σχόλια του προγράμματος.

Αλλά πιο συγκεκριμένα ας δούμε πώς εφαρμόζεται ο συγχρονισμός και η χρήση κοινής μνήμης για παράλληλους υπολογισμούς και εξόδους του συνόλου Mandelbrot.

- Συγχρονισμός:

Ο κώδικας χρησιμοποιεί πολλαπλές διεργασίες-παιδιά για τον υπολογισμό και την παραγωγή διαφορετικών γραμμών του συνόλου Mandelbrot παράλληλα. Για να διασφαλιστεί ο σωστός συγχρονισμός μεταξύ αυτών των διεργασιών, το πρόγραμμα χρησιμοποιεί μια απλή δομή βρόχου. Σε κάθε διεργασία εκχωρείται ένας αριθμός γραμμής για εργασία, ξεκινώντας από τον δείκτη διεργασίας της και αυξάνοντας κατά τον συνολικό αριθμό διεργασιών. Αυτή η κατανομή της εργασίας διασφαλίζει ότι κάθε διαδικασία χειρίζεται ένα μοναδικό σύνολο γραμμών.

- Κοινόχρηστη μνήμη:

Για να διευκολυνθεί η επικοινωνία και η κοινή χρήση δεδομένων μεταξύ των διεργασιών, ο κώδικας χρησιμοποιεί κοινόχρηστη μνήμη. Η συνάρτηση `create_shared_memory_area` χρησιμοποιείται για τη δημιουργία μιας κοινόχρηστης περιοχής μνήμης για την αποθήκευση των δεδομένων του συνόλου Mandelbrot. Αυτή η περιοχή μνήμης είναι προσβάσιμη από όλες τις επακόλουθες διεργασίες της γονικής διαδικασίας.

Στην κύρια συνάρτηση, προτού διακλαδιστούν οι θυγατρικές διεργασίες, ο κώδικας δημιουργεί την κοινόχρηστη μνήμη για το `buffer`. Η μεταβλητή `buffer` είναι ένας δισδιάστατος πίνακας ακεραίων και κάθε διεργασία θα έχει πρόσβαση σε αυτήν την κοινόχρηστη περιοχή μνήμης. Κάθε θυγατρική διαδικασία υπολογίζει στη συνέχεια τις γραμμές συνόλου Mandelbrot που της έχουν αντιστοιχιστεί και εγγράφει τα αποτελέσματα απευθείας στην προσωρινή μνήμη κοινόχρηστης μνήμης.

Αφού όλες οι διεργασίες-παιδιά ολοκληρώσουν τους υπολογισμούς τους, η κύρια διεργασία επαναλαμβάνεται πάνω από τις γραμμές στο `buffer` και τις εξάγει χρησιμοποιώντας τη συνάρτηση `output_mandel_line`. Δεδομένου ότι η προσωρινή μνήμη μοιράζεται μεταξύ όλων των διεργασιών, κάθε διεργασία γράφει την υπολογισμένη γραμμή της στην κατάλληλη θέση μνήμης στην προσωρινή μνήμη, επιτρέποντας στην κύρια διεργασία να έχει πρόσβαση στα πλήρη δεδομένα του συνόλου Mandelbrot.

Συνολικά, ο συγχρονισμός επιτυγχάνεται με την κατανομή της εργασίας μεταξύ των διεργασιών-παιδιά με βάση τους δείκτες τους και η κοινή μνήμη επιτρέπει στις διεργασίες να επικοινωνούν και να μοιράζονται αποτελεσματικά τα αποτελέσματά τους. Αυτός ο συνδυασμός συγχρονισμού και χρήσης κοινής μνήμης επιτρέπει τον παράλληλο υπολογισμό και την έξοδο του συνόλου Mandelbrot, βελτιώνοντας την απόδοση αξιοποιώντας τις δυνατότητες πολλαπλών διεργασιών.

Χρησιμοποιούμε το `Makefile` που παραθέσαμε παραπάνω για την μεταγλώττιση και το `linking`.

Μεταγλωττίζοντας και εκτελώντας το πρόγραμμα έχουμε το εξής:

3. Μετά τον υπολογισμό των γραμμών συνόλου Mandelbrot, κάθε διεργασία-παιδί εγγράφει τα αποτελέσματά της απευθείας στο buffer κοινόχρηστης μνήμης.
4. Μόλις όλες οι θυγατρικές διεργασίες ολοκληρώσουν τους υπολογισμούς τους, η κύρια διεργασία επαναλαμβάνεται πάνω από τις γραμμές στο buffer και τις εξάγει χρησιμοποιώντας τη συνάρτηση `output_mandel_line`.

Σε αυτό το σημείο, όλες οι διεργασίες-παιδιά έχουν ολοκληρώσει την εργασία που τους έχει ανατεθεί και η κύρια διεργασία διασφαλίζει τον συγχρονισμό περιμένοντας την έξοδο από κάθε θυγατρική διαδικασία χρησιμοποιώντας τη λειτουργία αναμονής. Αυτό εγγυάται ότι όλες οι θυγατρικές διεργασίες έχουν ολοκληρώσει τους υπολογισμούς τους πριν προχωρήσουν στην έξοδο του συνόλου Mandelbrot.

Εάν το buffer είχε διαστάσεις `NPROCS x x_chars`, όπου `NPROCS` είναι ο αριθμός των διεργασιών, το σχήμα συγχρονισμού θα πρέπει να τροποποιηθεί για να προσαρμοστεί η αλλαγή στη δομή του buffer. Κάθε θυγατρική διεργασία θα πρέπει να υπολογίζει και να εξάγει τις γραμμές που αντιστοιχούν στον δείκτη διεργασίας της, αντί να αυξάνονται με βάση τον υπολογισμό. Η δομή βρόχου στη συνάρτηση `compute_and_output_mandel_line` θα προσαρμοστεί ανάλογα για να διασφαλιστεί ότι κάθε διεργασία λειτουργεί στο εκχωρημένο σύνολο γραμμών εντός της τροποποιημένης δομής buffer. Η κύρια διεργασία θα εξακολουθεί να περιμένει να ολοκληρωθούν όλες οι θυγατρικές διεργασίες προτού προχωρήσει στην έξοδο του συνόλου Mandelbrot.

Εάν δεν υπάρχουν τροποποιήσεις στο σχήμα συγχρονισμού, αλλά η προσωρινή μνήμη έχει διαστάσεις `NPROCS x x_chars`, όπου `NPROCS` είναι ο αριθμός των διεργασιών, θα οδηγούσε σε λανθασμένους και αλληλοεπικαλυπτόμενους υπολογισμούς.

Μια άλλη 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.