**ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ**

ΗΛΙΑΣ ΝΤΟΝΤΟΡΟΣ

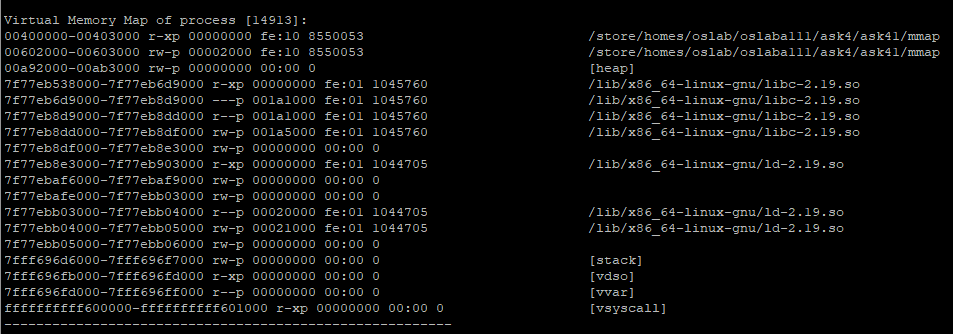
ΘΟΔΩΡΗΣ- ΙΩΑΝΝΗΣ ΚΙΤΣΗΣ

ΑΝΑΦΟΡΑ 4ης ΣΕΙΡΑΣ ΑΣΚΗΣΕΩΝ

ΑΣΚΗΣΗ 1Η

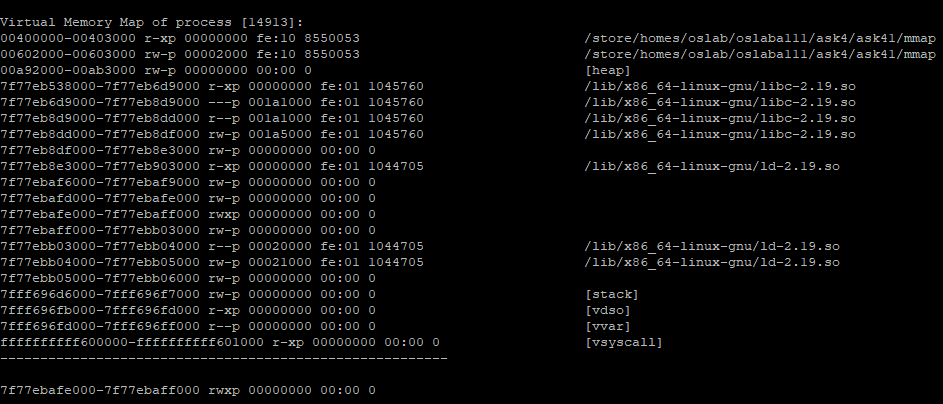
* + 1. **Print the virtual address space map of this process [9283].**

Με την εντολή show\_maps() τυπώνουμε το χάρτη της εικονικής μνήμης της τρέχουσας διεργασίας.



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

Παρατηρούμε ότι στον χάρτη που τυπώσαμε, στην γραμμή 10 έχει συγχωνευτεί μία σελίδα την οποία δεσμεύσαμε με την mmap. Το ΛΣ επέκτεινε την απεικόνιση αυτής της γραμμής κατά 1 σελίδα από την αρχή της επειδή σηματοδοτήσαμε την νέα απεικόνιση ως private για τη διεργασία αυτή.



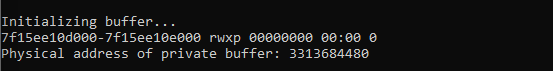
* + 1. **Step 3: Find and print the physical address of the buffer in main memory. What do you see?**

Αφού δεν αρχικοποιήσαμε τον buffer η δεσμευμένη εικονική μνήμη δεν αντιστοιχεί ακόμα σε κάποιο πεδίο διευθύνσεων στην πραγματική μνήμη. Αυτό συμβαίνει λόγω Demand Paging όταν μια διεργασία χρειαστεί να αποθηκεύσει δεδομένα σε μία σελίδα δημιουργείται το αντίστοιχο πλαίσιο στη φυσική μνήμη.



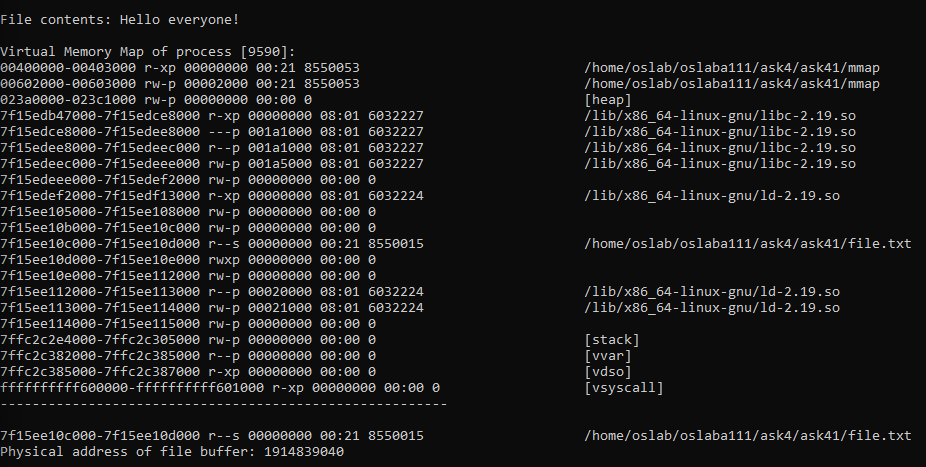
* + 1. **Step 4: Initialize your buffer with zeros and repeat Step 3. What happened?**

Από τη στιγμή που αρχικοποιήσαμε τον buffer παρατηρούμε ότι υπάρχει αντιστοίχιση της απεικόνισής μας σε πραγματική διεύθυνση φυσικής μνήμης.



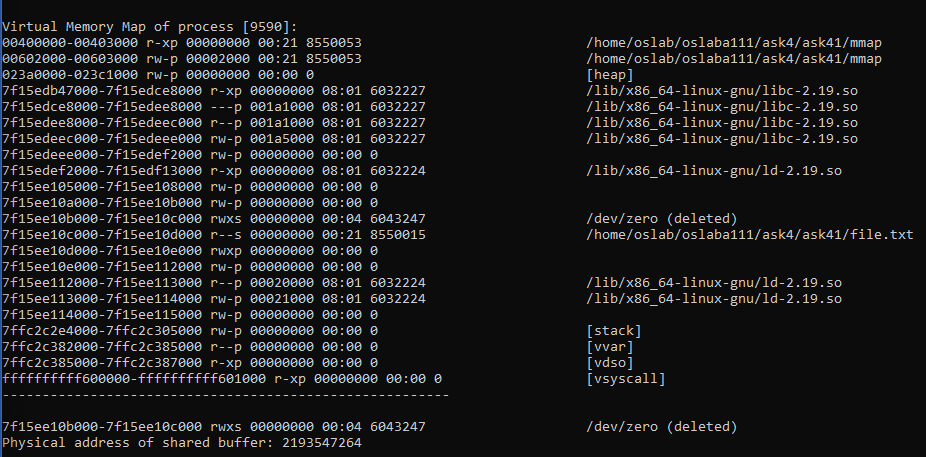
* + 1. **Step 5: Use mmap(2) to read and print file.txt. Print the new mapping information that has been created.**

Αφού χρησιμοποιήσαμε την open() για να ανοίξουμε το αρχείο δημιουργείται ένα shared mapping μεγέθους όσο και το μέγεθος του αρχείου με χρήση των κατάλληλων flag για ανάγνωση αρχείου. Τα περιεχόμενα του buffer είναι και οι χαρακτήρες του αρχείου. Στον χάρτη απεικονίσεων παρατηρούμε ότι έχει δημιουργηθεί ένα νέο record για αυτή την απεικόνιση.



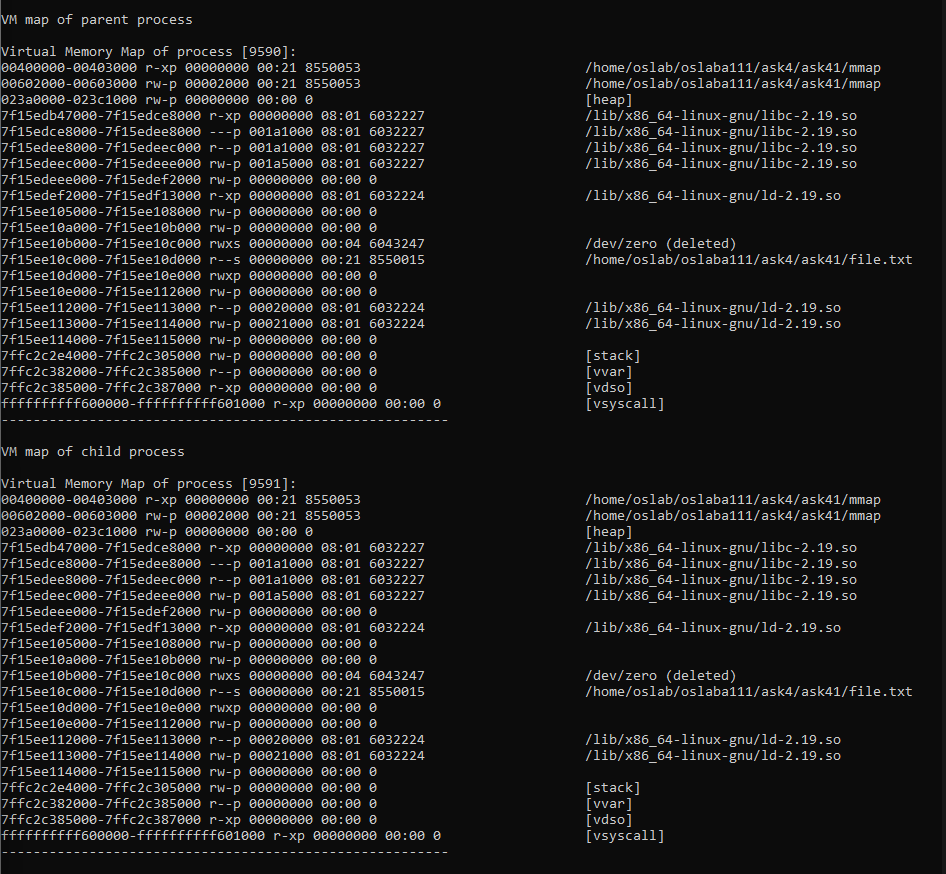
* + 1. **Step 6: Use mmap(2) to allocate a shared buffer of size equal to 1 page. Initialize the buffer and print the new mapping information that has been created.**

Το entry του νέου shared buffer που κατασκευάσαμε εμφανίζεται στον χάρτη απεικονίσεων αφού μετά την δημιουργία του ο buffer αρχικοποιείται.



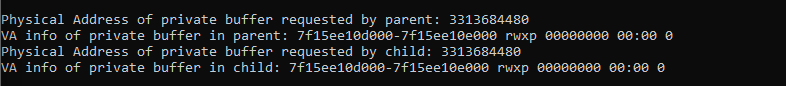
* + 1. **Step 7: Print parent's and child's map.**

Δημιουργείται μια νέα διεργασία ,η διεργασία γονέας που εκτελεί την parent() και η διεργασία παιδί που εκτελεί την child(). Αφού η μια είναι αντίγραφο της άλλης, οι χάρτες απεικονίσεών είναι πανομοιότυποι.



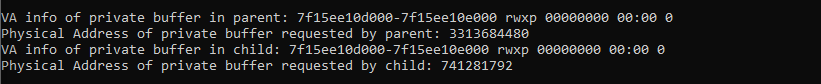
* + 1. **Step 8: Find the physical address of the private heap buffer (main) for both the parent and the child.**

Αμέσως μετά το fork, η διεύθυνση στην οποία απεικονίζεται ο private buffer είναι η ίδια και στις δύο διεργασίες. Ακολουθείται Copy on Write, δηλαδή, ενώ αρχικά οι δείκτες δείχνουν στην ίδια διεύθυνση, όταν μία από τις δύο διεργασίες προσπαθήσει να γράψει στον private buffer,τότε το λειτουργικό αντιγράφει τα περιεχόμενα της μνήμης σε διαφορετικό πεδίο διευθύνσεων ώστε να αποφεύγονται οι άσκοπες αντιγραφές.



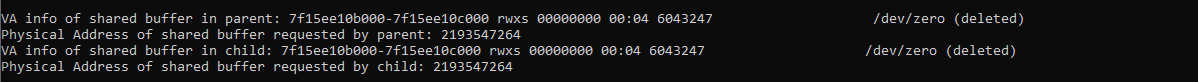
* + 1. **Step 9: Write to the private buffer from the child and repeat step 8. What happened?**

Αφού γράψουμε στον private buffer από το παιδί , η διεύθυνσή του μεταφέρεται σε άλλη διεύθυνση φυσικής μνήμης λόγω COW.



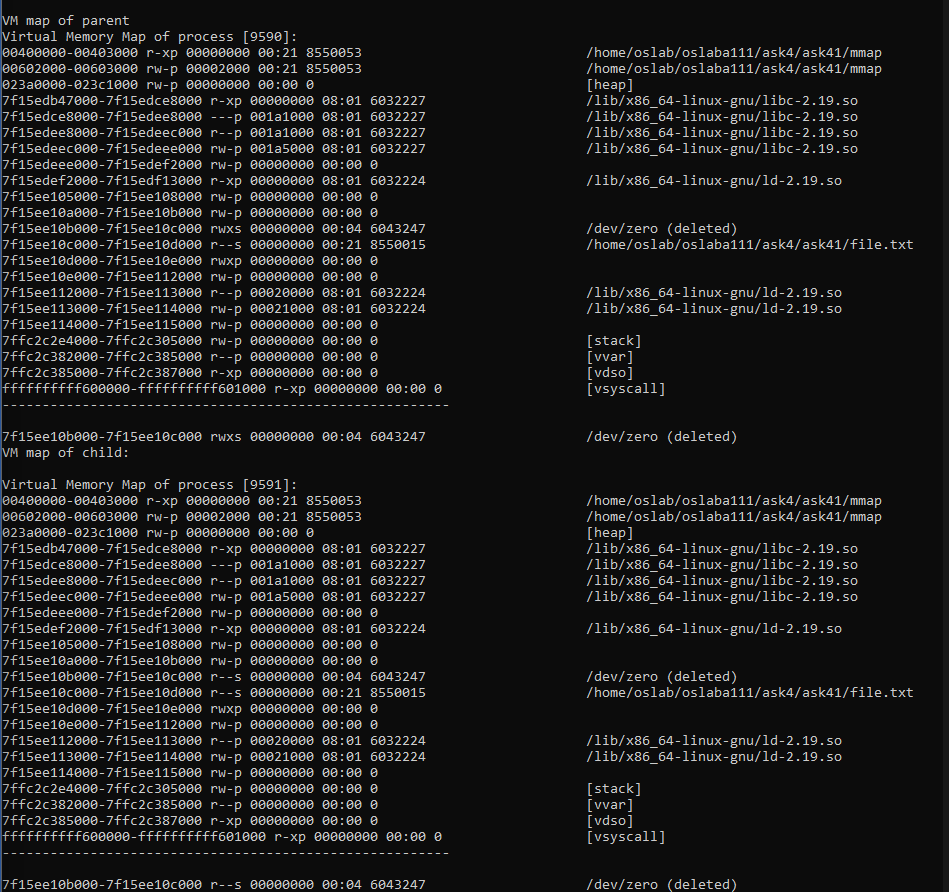
* + 1. **Step 10: Write to the shared heap buffer (main) from child and get the physical address for both the parent and the child. What happened?**

Ο shared buffer είναι στις ίδιες θέσεις μνήμης και για τις δύο διεργασίες.



* + 1. **Step 11: Disable writing on the shared buffer for the child. Verify through the maps for the parent and the child.**

Αλλάζοντας τα δικαιώματα του buffer στη διεργασία παιδί, παρατηρούμε ότι οι χάρτες των δύο διεργασιών διαφέρουν ως προς τα δικαιώματα αυτά. Δηλαδή, το παιδί έχει μόνο δικαίωμα ανάγνωσης (r) ενώ ο πατέρας συνεχίζει να έχει δικαιώματα ανάγνωσης και εγγραφής (rw).



* + 1. Τέλος, αποδεσμεύουμε τις απεικονίσεις και κλείνουμε το αρχείο “file.txt”.

Ο κώδικας για την πρώτη άσκηση :

1. /\*
2. \* mmap.c
3. \*
4. \* Examining the virtual memory of processes.
5. \*
6. \* Operating Systems course, CSLab, ECE, NTUA
7. \*
8. \*/
10. #include <stdlib.h>
11. #include <string.h>
12. #include <stdio.h>
13. #include <sys/mman.h>
14. #include <unistd.h>
15. #include <sys/types.h>
16. #include <sys/stat.h>
17. #include <fcntl.h>
18. #include <errno.h>
19. #include <stdint.h>
20. #include <signal.h>
21. #include <sys/wait.h>
23. #include "help.h"
25. #define RED     "\033[31m"
26. #define RESET   "\033[0m"
28. char \*heap\_private\_buf;
29. char \*heap\_shared\_buf;
30. char \*file\_shared\_buf;
32. uint64\_t buffer\_size;
34. /\*
35. \* Child process' entry point.
36. \*/
37. void child(void)
38. {
39. //      uint64\_t pa;
41. /\*
42. \* Step 7 - Child
43. \*/
44. if (0 != raise(SIGSTOP))
45. die("raise(SIGSTOP)");
46. /\*
47. \* TODO: Write your code here to complete child's part of Step 7.
48. \*/
49. printf("VM map of child process\n");
50. show\_maps();
52. /\*
53. \* Step 8 - Child
54. \*/
55. if (0 != raise(SIGSTOP))
56. die("raise(SIGSTOP)");
57. /\*
58. \* TODO: Write your code here to complete child's part of Step 8.
59. \*/
61. printf("Physical Address of private buffer requested by child: %ld\n", get\_physical\_address((uint64\_t)heap\_private\_buf));
62. printf("VA info of private buffer in child: ");
63. show\_va\_info((uint64\_t) heap\_private\_buf);
65. /\*
66. \* Step 9 - Child
67. \*/
68. if (0 != raise(SIGSTOP))
69. die("raise(SIGSTOP)");
70. /\*
71. \* TODO: Write your code here to complete child's part of Step 9.
72. \*/
73. int j;
74. for (j = 0; j < (int) buffer\_size; j++) {
75. heap\_private\_buf[j] = j;
76. }
77. printf("VA info of private buffer in child: ");
78. show\_va\_info((uint64\_t) heap\_private\_buf);
80. printf("Physical Address of private buffer requested by child: %ld\n", get\_physical\_address((uint64\_t)heap\_private\_buf));
82. /\*
83. \* Step 10 - Child
84. \*/
85. if (0 != raise(SIGSTOP))
86. die("raise(SIGSTOP)");
87. /\*
88. \* TODO: Write your code here to complete child's part of Step 10.
89. \*/
91. for (j=0; j<(int) buffer\_size; j++) {
92. heap\_shared\_buf[j] = j;
93. }
94. printf("VA info of shared buffer in child: ");
95. show\_va\_info((uint64\_t) heap\_shared\_buf);
97. printf("Physical Address of shared buffer requested by child: %ld\n", get\_physical\_address((uint64\_t)heap\_shared\_buf));
99. /\*
100. \* Step 11 - Child
101. \*/
103. //printf("Physical Address requested by child: %ld", get\_physical\_address(heap\_shared\_buf));
105. if (0 != raise(SIGSTOP))
106. die("raise(SIGSTOP)");
108. /\*
109. \* TODO: Write your code here to complete child's part of Step 11.
110. \*/
111. mprotect(heap\_shared\_buf,buffer\_size,PROT\_READ);
112. printf("VM map of child:\n");
113. show\_maps();
114. show\_va\_info((uint64\_t)heap\_shared\_buf);
116. /\*
117. \* Step 12 - Child
118. \*/
119. /\*
120. \* TODO: Write your code here to complete child's part of Step 12.
121. \*/
122. munmap(heap\_shared\_buf,buffer\_size);
123. munmap(heap\_private\_buf,buffer\_size);
124. munmap(file\_shared\_buf,buffer\_size);
126. }
128. /\*
129. \* Parent process' entry point.
130. \*/
131. void parent(pid\_t child\_pid)
132. {
133. //uint64\_t pa;
134. int status;
136. /\* Wait for the child to raise its first SIGSTOP. \*/
137. if (-1 == waitpid(child\_pid, &status, WUNTRACED))
138. die("waitpid");
140. /\*
141. \* Step 7: Print parent's and child's maps. What do you see?
142. \* Step 7 - Parent
143. \*/
145. printf(RED "\nStep 7: Print parent's and child's map.\n" RESET);
146. press\_enter();
148. printf("VM map of parent process\n");
149. show\_maps();
150. /\*
151. \* TODO: Write your code here to complete parent's part of Step 7.
152. \*/
154. if (-1 == kill(child\_pid, SIGCONT))
155. die("kill");
156. if (-1 == waitpid(child\_pid, &status, WUNTRACED))
157. die("waitpid");
159. /\*
160. \* Step 8: Get the physical memory address for heap\_private\_buf.
161. \* Step 8 - Parent
162. \*/
164. printf(RED "\nStep 8: Find the physical address of the private heap "
165. "buffer (main) for both the parent and the child.\n" RESET);
166. press\_enter();
168. /\*
169. \* TODO: Write your code here to complete parent's part of Step 8.
170. \*/
172. printf("Physical Address of private buffer requested by parent: %ld\n", get\_physical\_address((uint64\_t)heap\_private\_buf));
173. printf("VA info of private buffer in parent: ");
174. show\_va\_info((uint64\_t) heap\_private\_buf);
176. if (-1 == kill(child\_pid, SIGCONT))
177. die("kill");
178. if (-1 == waitpid(child\_pid, &status, WUNTRACED))
179. die("waitpid");
181. /\*
182. \* Step 9: Write to heap\_private\_buf. What happened?
183. \* Step 9 - Parent
184. \*/
186. printf(RED "\nStep 9: Write to the private buffer from the child and "
187. "repeat step 8. What happened?\n" RESET);
188. press\_enter();
189. printf("VA info of private buffer in parent: ");
190. show\_va\_info((uint64\_t) heap\_private\_buf);
192. printf("Physical Address of private buffer requested by parent: %ld\n", get\_physical\_address((uint64\_t)heap\_private\_buf));
193. /\*
194. \* TODO: Write your code here to complete parent's part of Step 9.
195. \*/
197. if (-1 == kill(child\_pid, SIGCONT))
198. die("kill");
199. if (-1 == waitpid(child\_pid, &status, WUNTRACED))
200. die("waitpid");
202. /\*
203. \* Step 10: Get the physical memory address for heap\_shared\_buf.
204. \* Step 10 - Parent
205. \*/
206. printf(RED "\nStep 10: Write to the shared heap buffer (main) from "
207. "child and get the physical address for both the parent and "
208. "the child. What happened?\n" RESET);
209. press\_enter();
211. /\*
212. \* TODO: Write your code here to complete parent's part of Step 10.
213. \*/
214. printf("VA info of shared buffer in parent: ");
215. show\_va\_info((uint64\_t) heap\_shared\_buf);
217. printf("Physical Address of shared buffer requested by parent: %ld\n", get\_physical\_address((uint64\_t)heap\_shared\_buf));
219. if (-1 == kill(child\_pid, SIGCONT))
220. die("kill");
221. if (-1 == waitpid(child\_pid, &status, WUNTRACED))
222. die("waitpid");
224. /\*
225. \* Step 11: Disable writing on the shared buffer for the child
226. \* (hint: mprotect(2)).
227. \* Step 11 - Parent
228. \*/
229. printf(RED "\nStep 11: Disable writing on the shared buffer for the "
230. "child. Verify through the maps for the parent and the "
231. "child.\n" RESET);
232. press\_enter();
234. /\*
235. \* TODO: Write your code here to complete parent's part of Step 11.
236. \*/
237. printf("VM map of parent");
238. show\_maps();
239. show\_va\_info((uint64\_t)heap\_shared\_buf);
240. if (-1 == kill(child\_pid, SIGCONT))
241. die("kill");
242. if (-1 == waitpid(child\_pid, &status, 0))
243. die("waitpid");
245. /\*
246. \* Step 12: Free all buffers for parent and child.
247. \* Step 12 - Parent
248. \*/
250. /\*
251. \* TODO: Write your code here to complete parent's part of Step 12.
252. \*/
253. munmap(heap\_shared\_buf,buffer\_size);
254. munmap(heap\_private\_buf,buffer\_size);
255. munmap(file\_shared\_buf,buffer\_size);
256. }
258. int main(void)
259. {
260. pid\_t mypid, p;
261. int fd = -1;
262. //      uint64\_t pa;
264. mypid = getpid();
265. buffer\_size = 1 \* get\_page\_size();
267. /\*
268. \* Step 1: Print the virtual address space layout of this process.
269. \*/
270. printf(RED "\nStep 1: Print the virtual address space map of this "
271. "process [%d].\n" RESET, mypid);
272. press\_enter();
273. /\*
274. \* TODO: Write your code here to complete Step 1.
275. \*/
276. //mmap ( NULL, sizeof(int),PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, 1, 0 );
277. show\_maps();
278. /\*
279. \* Step 2: Use mmap to allocate a buffer of 1 page and print the map
280. \* again. Store buffer in heap\_private\_buf.
281. \*/
282. printf(RED "\nStep 2: Use mmap(2) to allocate a private buffer of "
283. "size equal to 1 page and print the VM map again.\n" RESET);
284. press\_enter();
285. /\*
286. \* TODO: Write your code here to complete Step 2.
287. \*/
288. heap\_private\_buf = mmap(heap\_private\_buf, buffer\_size, PROT\_READ | PROT\_WRITE| PROT\_EXEC, MAP\_PRIVATE | MAP\_ANONYMOUS, -1,0);
290. if (heap\_private\_buf == MAP\_FAILED)
291. {
292. perror("error on step 2\n");
293. exit(-1);
294. }
295. show\_maps();
296. show\_va\_info((uint64\_t)heap\_private\_buf);
297. /\*
298. \* Step 3: Find the physical address of the first page of your buffer
299. \* in main memory. What do you see?
300. \*/
301. printf(RED "\nStep 3: Find and print the physical address of the "
302. "buffer in main memory. What do you see?\n" RESET);
303. press\_enter();
304. /\*
305. \* TODO: Write your code here to complete Step 3.
306. \*/
307. get\_physical\_address((uint64\_t)heap\_private\_buf);
308. //show\_va\_info(heap\_private\_buf);
309. //munmap(heap\_private\_buf);
310. /\*
311. \* Step 4: Write zeros to the buffer and repeat Step 3.
312. \*/
313. printf(RED "\nStep 4: Initialize your buffer with zeros and repeat "
314. "Step 3. What happened?\n" RESET);
315. press\_enter();
316. /\*
317. \* TODO: Write your code here to complete Step 4.
318. \*/
319. //heap\_zeros\_buf = mmap(heap\_zeros\_buf, buffer\_size,PROT\_READ | PROT\_WRITE, MAP\_PRIVATE| MAP\_ANONYMOUS,-1,0);
320. //show\_va\_info(heap\_zeros\_buf);
321. //get\_physical\_address(heap\_private\_buf);
322. int i;
323. //heap\_zeros\_buf[0] = 0;
324. printf("Initializing buffer...\n");
325. for (i=0; i < (int) buffer\_size; i++) {
326. \*(heap\_private\_buf+i)= 0;
327. }
328. /\*for (i=0; i <(int) buffer\_size; i++) {
329. printf("%d", \*(heap\_private\_buf+i));
330. }\*/
332. show\_va\_info((uint64\_t)heap\_private\_buf);
333. printf("Physical address of private buffer: %ld\n",get\_physical\_address((uint64\_t)heap\_private\_buf));
335. /\*
336. \* Step 5: Use mmap(2) to map file.txt (memory-mapped files) and print
337. \* its content. Use file\_shared\_buf.
338. \*/
339. printf(RED "\nStep 5: Use mmap(2) to read and print file.txt. Print "
340. "the new mapping information that has been created.\n" RESET);
341. press\_enter();
342. /\*
343. \* TODO: Write your code here to complete Step 5.
344. \*/
345. fd = open("file.txt",O\_RDONLY);
346. if (fd == -1) {
347. perror("error opening file");
348. }
349. //show\_maps();
350. char c;
351. file\_shared\_buf = mmap(NULL, buffer\_size, PROT\_READ, MAP\_SHARED,fd,0);
353. if (file\_shared\_buf == MAP\_FAILED)
354. {
355. perror("error on step 5\n");
356. exit(-1);
357. }
359. printf("File contents: ");
360. for (i=0; i<(int) buffer\_size; i++){
361. c = \*(file\_shared\_buf+i);
362. if (c !=  EOF)
363. putchar(c);
364. else
365. break;
366. }
367. //printf("\n");
368. show\_maps();
369. show\_va\_info((uint64\_t)file\_shared\_buf);
371. printf("Physical address of file buffer: %ld\n",get\_physical\_address((uint64\_t)file\_shared\_buf));
372. /\*
373. \* Step 6: Use mmap(2) to allocate a shared buffer of 1 page. Use
374. \* heap\_shared\_buf.
375. \*/
376. printf(RED "\nStep 6: Use mmap(2) to allocate a shared buffer of size "
377. "equal to 1 page. Initialize the buffer and print the new "
378. "mapping information that has been created.\n" RESET);
379. press\_enter();
380. /\*
381. \* TODO: Write your code here to complete Step 6.
382. \*/
383. heap\_shared\_buf = mmap(heap\_shared\_buf, buffer\_size, PROT\_READ| PROT\_WRITE| PROT\_EXEC, MAP\_SHARED| MAP\_ANONYMOUS,-1,0);
385. if (heap\_private\_buf == MAP\_FAILED)
386. {
387. perror("error on step 6\n");
388. exit(-1);
389. }
391. for(i=0; i<(int) buffer\_size; i++){
392. \*(heap\_shared\_buf+i)=0;
393. }
394. show\_maps();
395. show\_va\_info((uint64\_t)heap\_shared\_buf);
396. printf("Physical address of shared buffer: %ld\n",get\_physical\_address((uint64\_t)heap\_shared\_buf));
397. p = fork();
398. if (p < 0)
399. die("fork");
400. if (p == 0) {
401. child();
402. return 0;
403. }
405. parent(p);
407. if (-1 == close(fd))
408. perror("close");
409. return 0;
410. }

ΑΣΚΗΣΗ 2Η

* + 1. Η υλοποίηση με νήματα περιμένουμε να έχει καλύτερη επίδοση από την υλοποίηση με διεργασίες.Με την δημιουργία μίας διεργασίας γίνεται αντιγραφή όλων των δεδομένων (program counter, registers, virtual memory κ.λπ.) και αυτό σε συνδυασμό με την mmap() κοστίζουν πολύ σε χρόνο. Αντίθετα, με νήματα δεν έχουμε αντιγραφή, το οποίο σε πιο σύνθετα προβλήματα θα είχε αισθητή διαφορά στην επίδοση παρόλα επειδή το πρόβλημα είναι απλό δεν παρατηρούμε μεγάλη αλλαγή στην επίδοση .
    2. Αυτή τη φορά κατασκευάσαμε ένα shared buffer μεγέθους ο οποίο αποθηκεύει τη χρωματική τιμή κάθε χαρακτήρα. Αφού δημιουργηθούν οι διεργασίες παιδία, κάνουν τους υπολογισμούς των γραμμών και ανανεώνουν τον buffer και η διεργασία γονέας τυπώνει τον buffer χρησιμοποιώντας τη συνάρτηση output\_mandel\_line(). Ο συγχρονισμός εδώ επιτυγχάνεται απλά με τη χρήση πολλών wait() που είναι σε αριθμό όσες και οι διεργασίες παιδία που αναλαμβάνουν τον υπολογισμό.Εάν ο buffer μας είχε μέγεθος NPROCS \* x\_chars κάθε διεργασία μπορεί να αναλάβει NPROCS γραμμές. Με αυτό τον τρόπο, αφού μια διεργασία υπολογίσει και τυπώσει μία από τις γραμμές που της αντιστοιχούν θα κάνει raise(SIGSTOP) για να αναστείλει τον εαυτό της. Όταν όλες οι διεργασίες έχουν κάνει raise(SIGSTOP) τότε ο buffer θα γεμίσει με συνεχόμενες N γραμμές οι οποίες είναι έτοιμες για εκτύπωση. Η διεργασία γονιός περιμένει τα παιδιά της να σταματήσουν και μετά τυπώνει τα περιεχόμενα του buffer. Στη συνέχεια στέλνει σήμα SIGCONT στα παιδία ώστε να υπολογιστούν οι επόμενες Ν γραμμές. Αυτή η διαδικασία θα επαναληφθεί NPROCS / y\_chars φορές.

Ο κώδικας για την άσκηση 2.1 :

1. /\*
2. \* mandel.c
3. \*
4. \* A program to draw the Mandelbrot Set on a 256-color xterm.
5. \*
6. \*/
8. #include <assert.h>
9. #include <math.h>
10. #include <semaphore.h>
11. #include <stdio.h>
12. #include <stdlib.h>
13. #include <string.h>
14. #include <sys/mman.h>
15. #include <sys/wait.h>
16. #include <unistd.h>
18. #include "mandel-lib.h"
20. #define MANDEL\_MAX\_ITERATION 100000
22. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
23. \* Compile-time parameters \*
24. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
26. /\*
27. \* Output at the terminal is is x\_chars wide by y\_chars long
28. \*/
29. int y\_chars = 50;
30. int x\_chars = 90;
32. /\*
33. \* The part of the complex plane to be drawn:
34. \* upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
35. \*/
36. double xmin = -1.8, xmax = 1.0;
37. double ymin = -1.0, ymax = 1.0;
39. /\*
40. \* Every character in the final output is
41. \* xstep x ystep units wide on the complex plane.
42. \*/
43. double xstep;
44. double ystep;
46. sem\_t \*sem;
48. /\*
49. \* This function computes a line of output
50. \* as an array of x\_char color values.
51. \*/
52. void compute\_mandel\_line(int line, int color\_val[]) {
53. /\*
54. \* x and y traverse the complex plane.
55. \*/
56. double x, y;
58. int n;
59. int val;
61. /\* Find out the y value corresponding to this line \*/
62. y = ymax - ystep \* line;
64. /\* and iterate for all points on this line \*/
65. for (x = xmin, n = 0; n < x\_chars; x += xstep, n++) {
67. /\* Compute the point's color value \*/
68. val = mandel\_iterations\_at\_point(x, y, MANDEL\_MAX\_ITERATION);
69. if (val > 255)
70. val = 255;
72. /\* And store it in the color\_val[] array \*/
73. val = xterm\_color(val);
74. color\_val[n] = val;
75. }
76. }
78. /\*
79. \* This function outputs an array of x\_char color values
80. \* to a 256-color xterm.
81. \*/
82. void output\_mandel\_line(int fd, int color\_val[]) {
83. int i;
85. char point = '@';
86. char newline = '\n';
88. for (i = 0; i < x\_chars; i++) {
89. /\* Set the current color, then output the point \*/
90. set\_xterm\_color(fd, color\_val[i]);
91. if (write(fd, &point, 1) != 1) {
92. perror("compute\_and\_output\_mandel\_line: write point");
93. exit(1);
94. }
95. }
97. /\* Now that the line is done, output a newline character \*/
98. if (write(fd, &newline, 1) != 1) {
99. perror("compute\_and\_output\_mandel\_line: write newline");
100. exit(1);
101. }
102. }
104. void compute\_and\_output\_mandel\_line(int fd, int line, int N) {
106. /\*
107. \* A temporary array, used to hold color values for the line being drawn
108. \*/
109. int color\_val[x\_chars];
110. compute\_mandel\_line(line, color\_val);
111. sem\_wait(&sem[line % N]);
112. output\_mandel\_line(fd, color\_val);
113. sem\_post(&sem[(line + 1) % N]);
114. }
116. /\*
117. \* Create a shared memory area, usable by all descendants of the calling
118. \* process.
119. \*/
120. void \*create\_shared\_memory\_area(unsigned int numbytes) {
121. int pages;
122. void \*addr;
124. if (numbytes == 0) {
125. fprintf(stderr, "%s: internal error: called for numbytes == 0\n", \_\_func\_\_);
126. exit(1);
127. }
129. /\*
130. \* Determine the number of pages needed, round up the requested number of
131. \* pages
132. \*/
133. pages = (numbytes - 1) / sysconf(\_SC\_PAGE\_SIZE) + 1;
135. /\* Create a shared, anonymous mapping for this number of pages \*/
136. addr = mmap(NULL, sysconf(\_SC\_PAGE\_SIZE) \* pages, PROT\_READ | PROT\_WRITE,
137. MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);
139. return addr;
140. }
142. void destroy\_shared\_memory\_area(void \*addr, unsigned int numbytes) {
143. int pages;
145. if (numbytes == 0) {
146. fprintf(stderr, "%s: internal error: called for numbytes == 0\n", \_\_func\_\_);
147. exit(1);
148. }
150. /\*
151. \* Determine the number of pages needed, round up the requested number of
152. \* pages
153. \*/
154. pages = (numbytes - 1) / sysconf(\_SC\_PAGE\_SIZE) + 1;
156. if (munmap(addr, pages \* sysconf(\_SC\_PAGE\_SIZE)) == -1) {
157. perror("destroy\_shared\_memory\_area: munmap failed");
158. exit(1);
159. }
160. }
162. int main(int argc, char \*\*argv) {
163. int line, i, p, status;
165. if (argc != 2) {
166. perror("usage");
167. exit(1);
168. }
170. int NPROCS = atoi(argv[1]);
171. sem = (sem\_t \*)create\_shared\_memory\_area(sizeof(sem\_t) \* NPROCS);
173. xstep = (xmax - xmin) / x\_chars;
174. ystep = (ymax - ymin) / y\_chars;
176. /\*
177. \* draw the Mandelbrot Set, one line at a time.
178. \* Output is sent to file descriptor '1', i.e., standard output.
179. \*/
181. for (i = 0; i < NPROCS; ++i)
182. sem\_init(&sem[i], 1, 0);
183. sem\_post(&sem[0]);
185. for (i = 0; i < NPROCS; ++i) {
186. p = fork();
187. if (p == 0) {
188. for (line = i; line < y\_chars; line += NPROCS)
189. compute\_and\_output\_mandel\_line(1, line, NPROCS);
190. exit(0);
191. }
192. }
194. for (i = 0; i < NPROCS; ++i)
195. wait(&status);
197. destroy\_shared\_memory\_area(sem, NPROCS \* sizeof(sem\_t));
198. reset\_xterm\_color(1);
199. return 0;
200. }

Ο κώδικα για την άσκηση 2.2 :

1. /\*
2. \* mandel.c
3. \*
4. \* A program to draw the Mandelbrot Set on a 256-color xterm.
5. \*
6. \*/
8. #include <assert.h>
9. #include <math.h>
10. #include <semaphore.h>
11. #include <stdio.h>
12. #include <stdlib.h>
13. #include <string.h>
14. #include <sys/mman.h>
15. #include <sys/wait.h>
16. #include <unistd.h>
17. /\*TODO header file for m(un)map\*/
19. #include "mandel-lib.h"
21. #define MANDEL\_MAX\_ITERATION 100000
23. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
24. \* Compile-time parameters \*
25. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
27. /\*
28. \* Output at the terminal is is x\_chars wide by y\_chars long
29. \*/
30. int y\_chars = 50;
31. int x\_chars = 90;
33. /\*
34. \* The part of the complex plane to be drawn:
35. \* upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
36. \*/
37. double xmin = -1.8, xmax = 1.0;
38. double ymin = -1.0, ymax = 1.0;
40. /\*
41. \* Every character in the final output is
42. \* xstep x ystep units wide on the complex plane.
43. \*/
44. double xstep;
45. double ystep;
46. int \*buffer;
48. /\*
49. \* This function computes a line of output
50. \* as an array of x\_char color values.
51. \*/
52. void compute\_mandel\_line(int line, int color\_val[]) {
53. /\*
54. \* x and y traverse the complex plane.
55. \*/
56. double x, y;
58. int n;
59. int val;
61. /\* Find out the y value corresponding to this line \*/
62. y = ymax - ystep \* line;
64. /\* and iterate for all points on this line \*/
65. for (x = xmin, n = 0; n < x\_chars; x += xstep, n++) {
67. /\* Compute the point's color value \*/
68. val = mandel\_iterations\_at\_point(x, y, MANDEL\_MAX\_ITERATION);
69. if (val > 255)
70. val = 255;
72. /\* And store it in the color\_val[] array \*/
73. val = xterm\_color(val);
74. color\_val[n] = val;
75. }
76. }
78. /\*
79. \* This function outputs an array of x\_char color values
80. \* to a 256-color xterm.
81. \*/
82. void output\_mandel\_line(int fd, int color\_val[]) {
83. int i;
85. char point = '@';
86. char newline = '\n';
88. for (i = 0; i < x\_chars; i++) {
89. /\* Set the current color, then output the point \*/
90. set\_xterm\_color(fd, color\_val[i]);
91. if (write(fd, &point, 1) != 1) {
92. perror("compute\_and\_output\_mandel\_line: write point");
93. exit(1);
94. }
95. }
97. /\* Now that the line is done, output a newline character \*/
98. if (write(fd, &newline, 1) != 1) {
99. perror("compute\_and\_output\_mandel\_line: write newline");
100. exit(1);
101. }
102. }
104. void compute\_and\_output\_mandel\_line(int fd, int line, int N) {
105. /\*
106. \* A temporary array, used to hold color values for the line being drawn
107. \*/
108. int j;
109. int color\_val[x\_chars];
110. compute\_mandel\_line(line, color\_val);
111. for (j = 0; j < x\_chars; j++)
112. buffer[x\_chars \* line + j] = color\_val[j];
113. // output\_mandel\_line(fd, color\_val);
114. }
116. /\*
117. \* Create a shared memory area, usable by all descendants of the calling
118. \* process.
119. \*/
120. void \*create\_shared\_memory\_area(unsigned int numbytes) {
121. int pages;
122. void \*addr;
124. if (numbytes == 0) {
125. fprintf(stderr, "%s: internal error: called for numbytes == 0\n", \_\_func\_\_);
126. exit(1);
127. }
129. /\*
130. \* Determine the number of pages needed, round up the requested number of
131. \* pages
132. \*/
133. pages = (numbytes - 1) / sysconf(\_SC\_PAGE\_SIZE) + 1;
135. /\* Create a shared, anonymous mapping for this number of pages \*/
136. /\* TODO: \*/
137. addr = mmap(NULL, sysconf(\_SC\_PAGE\_SIZE) \* pages, PROT\_READ | PROT\_WRITE,
138. MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);
140. return addr;
141. }
143. void destroy\_shared\_memory\_area(void \*addr, unsigned int numbytes) {
144. int pages;
146. if (numbytes == 0) {
147. fprintf(stderr, "%s: internal error: called for numbytes == 0\n", \_\_func\_\_);
148. exit(1);
149. }
151. /\*
152. \* Determine the number of pages needed, round up the requested number of
153. \* pages
154. \*/
155. pages = (numbytes - 1) / sysconf(\_SC\_PAGE\_SIZE) + 1;
157. if (munmap(addr, pages \* sysconf(\_SC\_PAGE\_SIZE)) == -1) {
158. perror("destroy\_shared\_memory\_area: munmap failed");
159. exit(1);
160. }
161. }
163. int main(int argc, char \*\*argv) {
164. int line, i, p, status;
166. if (argc != 2) {
167. perror("usage");
168. exit(1);
169. }
171. int NPROCS = atoi(argv[1]);
172. buffer = create\_shared\_memory\_area(sizeof(int) \* x\_chars \* y\_chars);
174. xstep = (xmax - xmin) / x\_chars;
175. ystep = (ymax - ymin) / y\_chars;
177. /\*
178. \* draw the Mandelbrot Set, one line at a time.
179. \* Output is sent to file descriptor '1', i.e., standard output.
180. \*/
182. for (i = 0; i < NPROCS; ++i) {
183. p = fork();
184. if (p < 0) {
185. perror("Error:Fork");
186. exit(-1);
187. }
188. if (p == 0) {
189. for (line = i; line < y\_chars; line += NPROCS)
190. compute\_and\_output\_mandel\_line(1, line, NPROCS);
191. exit(0);
192. }
193. }
194. for (i = 0; i < NPROCS; ++i)
195. wait(&status);
197. for (i = 0; i < y\_chars; ++i)
198. output\_mandel\_line(1, buffer + x\_chars \* i);
200. destroy\_shared\_memory\_area(buffer, sizeof(int) \* x\_chars \* y\_chars);
201. reset\_xterm\_color(1);
202. return 0;
203. }