Jacobs University Bremen **Jovan Shandro**

OS homework 4

Problem 4.1

Solution:

a) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	3	3	1	1	1	1	1	1
frame 1		2	2	4	4	4	4	2	2	2
page fault	✓	✓	✓	✓	✓			✓		

Explanation: When the first 1 is accessed, since it is not already stored as both frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, it results in a page fault again and takes the place of 1 in frame 0 as it was inserted before 2 (because of FIFO strategy). Then 4 is accessed and same way a page fault is caused and it takes the place of 2 in frame 1. Same way when 1 is accessed, it results in a page fault and takes 3's place in frame 0 as 3 was inserted first. Then for the next 1, and 4, there are no page faults as they are already stored. The as 2 is accessed, it results in a page fault and takes 4's place in frame 1 as it was inserted before 1. Then the last two page accesses do not result in a page fault as both 1 and 2 are stored in the frames. So we get a total of 6 page faults.

b) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	1	4	4	4	4	4	4	4
frame 1		2	2	2	1	1	1	1	1	1
frame 2			3	3	3	3	3	2	2	2
page fault	✓	✓	✓	✓	✓			✓		

Explanation: When the first 1 is accessed, since it is not already stored as all frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, and frame 2 is empty it is stored in frame 2 resulting in a page fault. Then 4 is accessed and a page fault is caused as it is not already stored in the frames and it takes the place of 1 in frame 0 as it was inserted before the others. Same way when 1 is accessed, it results in a page fault and takes 2's place in frame 1 as 2 was inserted first. Then for the next 1, and 4, there are no page faults as they are already stored. The as 2 is accessed, it results in a page fault and takes 3's place in frame 2 as it was inserted before the others. Then the last two page accesses do not result in a page fault as both 1 and 2 are stored in the frames. So we get a total of 6 page faults.

c) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	1	1	1	1	1	1	1	1
frame 1		2	3	4	4	4	4	2	2	2
page fault	✓	✓	✓	✓				✓		

Explanation: When the first 1 is accessed, since it is not already stored as both frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, it results in a page fault again and takes the place of 2 in frame 1 as 2 will not be used for the longest period of time. Then 4 is accessed and same way a page fault is caused and it takes the place of 3 in frame 1 as 3 is never accessed again (in the given reference string). Then for the next 1, 1, and 4, there are no page faults as they are already stored. The as 2 is accessed, it results in a page fault and takes 4's place in frame 1 as 4 is not used again in our ref string whereas 1 is. Then the last two page accesses do not result in a page fault as both 1 and 2 are stored in the frames. So we get a total of 5 page faults.

d) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	1	1	1	1	1	1	1	1
frame 1		2	2	2	2	2	2	2	2	2
frame 2			3	4	4	4	4	4	4	4
page fault	✓	✓	✓	✓						

Explanation: When the first 1 is accessed, since it is not already stored as all frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, and frame 2 is empty it is stored in frame 2 resulting in a page fault. Then 4 is accessed and a page fault is caused as it is not already stored in the frames and it takes the place of 3 in frame 2 as it is never used again in our ref string. Then for all other page accesses we get no page faults as all 1 2 and 4 are stored in the three frames. So we get a total of 4 page faults.

e) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	3	3	1	1	1	2	2	2
frame 1		2	2	4	4	4	4	4	1	1
page fault	✓	✓	✓	✓	✓			✓	✓	

Explanation: When the first 1 is accessed, since it is not already stored as both frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, it results in a page fault again and takes the place of 1 in frame 0 as it was 1 was accessed for the last time before 2. Then 4 is accessed and same way a page fault is caused and it takes the place of 2 in frame 1 as 2 was accessed for the last time before 3. Same way when 1 is accessed, it results in a page fault and takes 3's place in frame 0 as 3 was last accessed before 4. Then for the next 1, and 4, there are no page faults as they are already stored. Then as 2 is accessed, it results in a page fault and takes 1's place in frame 0 as 1 it was accessed for the last time before 4. Then 1 is accessed and since it is not already stored it results in a page fault and takes 4's place in frame 1 as 4 was accessed for the last time before 2. Then the last page access does not result in a page fault as 2 is stored in the frames. So we get a total of 7 page faults.

f) The filled table will be:

reference string	1	2	3	4	1	1	4	2	1	2
frame 0	1	1	1	4	4	4	4	4	4	4
frame 1		2	2	2	1	1	1	1	1	1
frame 2			3	3	3	3	3	2	2	2
page fault	✓	✓	✓	✓	✓			✓		

Explanation: When the first 1 is accessed, since it is not already stored as all frames are empty, it results in a page fault and is then stored in frame 0. Then 2 is accessed, and same way since it is not already mapped and frame 1 is empty, it results in a page fault and is stored in frame 1. Then 3 is accessed and since it is not stored in the frames, and frame 2 is empty it is stored in frame 2 resulting in a page fault. Then 4 is accessed and a page fault is caused as it is not already stored in the frames and it takes the place of 1 in frame 0 as it was as 1 was accessed for the last time before the others. Same way when 1 is accessed, it results in a page fault and takes 2's place in frame 1 as 2 was accessed for the last time before the others. Then for the next 1, and 4, there are no page faults as they are already stored. Then as 2 is accessed, it results in a page fault and takes 3's place in frame 2 as 3 was accessed for the last time before the others. Then the last two page accesses do not result in a page fault as both 1 and 2 are stored in the frames. So we get a total of 6 page faults.

Problem 4.2

Solution:

a) The shell command will print "HELLO WORLD" in the standard output. First, it is important to say that the transformations will be applied in the same order as they appear in the arguments, so rot13 first, then upper, and in the end another rot13. This is because the transformation are applied in the same order as they are saved in our transformations array, and since argv is parsed using the opting of the getopt function, the transformations in the array will be in the same order as they are in the arguments. When the first rot13 is applied, it goes 13 characters forward for the characters that do not surpass 'z'(or 'Z') and 13 backwards for those that do. So after the first transformation the string "hello world" becomes "uryyb jbeyq". Then upper is applied which converts every character in uppercase, so the string now becomes "URYYB JBEYQ". Then another rot13 is applied and since the english alphabet has only 26 letters, it will do the inverse operatin compared to the first time, so the string now becomes "HELLO WORLD", which is the string that we see.

b) Running the pmap command with the -XX option (which shows all the information the kernel can give), I got the following output (I have not included all sections, but only the ones we need, as they were a lot):

```
11389:   ./cat++ -l ./librot13.so -l ./libupper.so -l ./librot13.so
         Address Perm   Offset Device   Inode Size KernelPageSize MMUPageSize  Rss Shared_Clean ...  Mapping
    5609d19ff000 r--p 00000000  08:03 6826534    4              4           4    4            0 ...  cat++
    5609d1a00000 r-xp 00001000  08:03 6826534    4              4           4    4            0 ...  cat++
    5609d1a01000 r--p 00002000  08:03 6826534    4              4           4    4            0 ...  cat++
    5609d1a02000 r--p 00002000  08:03 6826534    4              4           4    4            0 ...  cat++
    5609d1a03000 rw-p 00003000  08:03 6826534    4              4           4    4            0 ...  cat++
    5609d20bc000 rw-p 00000000  00:00       0  132              4           4   20            0 ...  [heap]
    7f7eefb5b000 rw-p 00000000  00:00       0   12              4           4    8            0 ...
    7f7eefb5e000 r--p 00000000  08:03  788573  148              4           4  140          140 ...  libc-2.29.so
    7f7eefb83000 r-xp 00025000  08:03  788573 1484              4           4 1248         1248 ...  libc-2.29.so
    7f7eefcf6000 r--p 00198000  08:03  788573  292              4           4  168          168 ...  libc-2.29.so
    7f7eefd3f000 r--p 001e0000  08:03  788573   12              4           4   12            0 ...  libc-2.29.so
    7f7eefd42000 rw-p 001e3000  08:03  788573   12              4           4   12            0 ...  libc-2.29.so
    7f7eefd45000 rw-p 00000000  00:00       0   16              4           4   16            0 ...
    7f7eefd49000 r--p 00000000  08:03  788575    4              4           4    4            4 ...  libdl-2.29.so
    7f7eefd4a000 r-xp 00001000  08:03  788575    8              4           4    8            8 ...  libdl-2.29.so
    7f7eefd4c000 r--p 00003000  08:03  788575    4              4           4    0            0 ...  libdl-2.29.so
    7f7eefd4d000 r--p 00003000  08:03  788575    4              4           4    4            0 ...  libdl-2.29.so
    7f7eefd4e000 rw-p 00004000  08:03  788575    4              4           4    4            0 ...  libdl-2.29.so
    7f7eefd4f000 rw-p 00000000  00:00       0    8              4           4    8            0 ...
    7f7eefd85000 r--p 00000000  08:03  788568    4              4           4    4            4 ...  ld-2.29.so
    7f7eefd86000 r-xp 00001000  08:03  788568  132              4           4  132          132 ...  ld-2.29.so
    7f7eefda7000 r--p 00022000  08:03  788568   32              4           4   32           32 ...  ld-2.29.so
    7f7eefdaf000 r--p 00029000  08:03  788568    4              4           4    4            0 ...  ld-2.29.so
    7f7eefdb0000 rw-p 0002a000  08:03  788568    4              4           4    4            0 ...  ld-2.29.so
    7f7eefdb1000 rw-p 00000000  00:00       0    4              4           4    4            0 ...
    7fffbea19000 rw-p 00000000  00:00       0  132              4           4   12            0 ...  [stack]
    7fffbeaa0000 r--p 00000000  00:00       0   12              4           4    0            0 ...  [vvar]
    7fffbeaa3000 r-xp 00000000  00:00       0    4              4           4    4            4 ...  [vdso]
ffffffffff600000 r-xp 00000000  00:00       0    4              4           4    0            0 ...  [vsyscall]
```

Now we answer the given questions. As we see in the image above, whenever a library is loaded, 5 segments are added to the process. As for the question

on what happens when a library is loaded multiple times, we first start with a small analysis of the results of pmap. On the address section we have the pages of the logic memory which in my system are of size 4096 each. Notice that even though we may expect certain parts of the library that was called twice to be the same, they appear to be different and this is because of a security feature also mentioned by the professor that randomizes them on purpose. The Size section shows the actual size in kb of different parts of the libraries. Notice the difference in Size and resident size (RSS) which shows that the entire library segment has not been loaded. At last, observe the Shared_Clean column. We see that the values for some of the segments in the first library (3 of them) equal to the RSS column. This means that the physical pages used to store these parts are shared between multiple processes, and since it equals RSS, all of that part of the library that is in memory is shared.

c) We first consider cat -n. Our cat++ program does indeed allow us to implement a transform the emulates it. The code below is possible implementation I made for the catn.c file to create the libcatn.so library

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *
transform(char *line)
{
    // static counter
    static int i = 1;
    // digit to be added in prefix
    char digit = i + '0';
    // get prefix
    char a[] = { digit, ' ', ' '};
    char * prefix = a;
    // copy current value of line
    int k = (int)strlen(line);
    char copy[k];
    for(int j = 0; j < k; j++)
        copy[j] = line[j];
    // reallocate enough space for new line
    line = (char *) realloc(line, strlen(prefix) + k + 1);
    // change the value of line
    sprintf(line, "%s%s", prefix, copy);
    // increment counter
    i++;
    return line;
}
```

To test it run

```
'gcc -fPIC -c catn.c'
```

then

```
'gcc -shared -o libcatn.so catn.o'
```

and then you can test it with

```
'echo -e "hello\nworld" \ | ./cat++ -l ./libcatn.so'
```

Now considering wc. Our current version of cat++ does not allow us to implement a transform that emulates it, as our current version performs the transform on every line of the file and then prints it in stdout. To implement wc we would have to keep track of the current line number, total nr of words and chars for the moment, and even though we may be able to get these values using static variables, there is no way we can know when we reached the last line so that we can print the output that wc prints. So it is impossible to be implemented.