

IT [204] - OPERATING SYSTEMS

Project 2: Memory Allocation;

Project 3: Concurrency - Synchronization Problems;

Amina Srna, Belma Šehić**

*Department of Information Technologies- Faculty of Engineering, Natural and Medical Sciences- International
Burch University*

amina.srna@stu.ibu.edu.ba, belma.sehic@stu.ibu.edu.ba

13/06/2024, Sarajevo

Project 2: Memory

Purpose: The purpose of this project is to familiarize you with the principles of memory management by implementing a memory manager to dynamically allocate memory in a program.

Task 2.1. Memory management tools

- Understand how the `mmap` and `munmap` system calls work. Explore how to use `mmap` to obtain pages of memory from the OS, and allocate chunks from these pages dynamically when requested. Familiarize yourself with the various arguments to the `mmap` system call.

The **`mmap()`** system call in Linux maps files or devices into memory, following the POSIX standard. It allows a process to request the operating system to map a file into its address space or to allocate anonymous memory. Pages are not loaded into physical memory until they are referenced, a technique known as demand paging.

The function prototype for `mmap()`:

→ `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`

Arguments of `mmap()` system call:

- **addr**: This argument is a hint to the OS kernel for the starting address of the virtual mapping in the process's virtual memory. If specified as `NULL`, the kernel can place the virtual mapping anywhere it deems fit. If not `NULL`, **addr** should be a multiple of the page size.
- **length**: Specifies the number of bytes for the mapping. This length should be a multiple of the page size.
- **prot**: The protection for the mapped memory. The value of **prot** is the bitwise OR of the following single-bit values:
 - `PROT_READ`: Enable the contents of the mapped memory to be readable by the process.
 - `PROT_WRITE`: Enable the contents of the mapped memory to be writable by the process.
 - `PROT_EXEC`: Enable the contents of the mapped memory to be executable by the process as CPU instructions.
- **flags**: Various options controlling the mapping. Common flags include:
 - `MAP_ANONYMOUS` (or `MAP_ANON`): Allocate anonymous memory; the pages are not backed by any file.
 - `MAP_FILE`: The default setting; the mapped region is backed by a regular file.
 - `MAP_FIXED`: Do not interpret **addr** as a hint; place the mapping exactly at that address, which must be a multiple of the page size.
 - `MAP_PRIVATE`: Modifications to the mapped memory region are not visible to other processes mapping the same file.
 - `MAP_SHARED`: Modifications to the mapped memory region are visible to other processes mapping the same file and are eventually reflected in the file.
 - `MAP_SHARED_VALIDATE`: Similar to `MAP_SHARED` but ensures all passed flags are known, failing the mapping with the error `EOPNOTSUPP` for unknown flags. This is required to use some flags (e.g., `MAP_SYNC`).

- **fd**: The open file descriptor for the file from which to populate the memory region. If `MAP_ANONYMOUS` is specified, `fd` should be `-1`.
- **offset**: If not an anonymous mapping, the memory-mapped region will be populated with data starting at `offset` bytes from the beginning of the file opened as file descriptor `fd`. This should be a multiple of the page size.

To showcase our practical understanding of these system calls, we have written a C code file named **mmap_example.c** that utilizes them.

- Write a simple C program that runs for a long duration, say, by pausing for user input or by sleeping. While the process is active, use the `ps` or any other similar command with suitable options, to measure the memory usage of the process. Specifically, measure the virtual memory size (VSZ) of the process, and the resident set size (RSS) of the process (which includes only the physical RAM pages allocated to the process). You should also be able to see the various pieces of the memory image of the process in the Linux `proc` file system, by accessing a suitable file in the `proc` filesystem.

Pictures on the following page represent:

- C code of the task `mmap_example.c` (The program will create a 1 MB file in `/tmp`, map it to memory, and wait for you to press Enter. This simulates a long-running process.)
- After running this task, the output is mapped memory

```

Users > mop1 > Desktop > C c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/mman.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #define FILEPATH "/tmp/mmapfile"
7  #define FILESIZE (1024 * 1024) // 1 MB
8
9  int main() {
10     int fd;
11     void *map;
12
13     fd = open(FILEPATH, O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);
14     if (fd == -1) {
15         perror("Error opening file for writing");
16         exit(EXIT_FAILURE);
17     }
18
19     if (lseek(fd, FILESIZE - 1, SEEK_SET) == -1) {
20         close(fd);
21         perror("Error calling lseek() to stretch the file");
22         exit(EXIT_FAILURE);
23     }
24
25     if (write(fd, "", 1) == -1) {
26         close(fd);
27         perror("Error writing last byte of the file");
28         exit(EXIT_FAILURE);
29     }
30
31     map = mmap(0, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
32     if (map == MAP_FAILED) {
33         close(fd);
34         perror("Error mmaping the file");
35         exit(EXIT_FAILURE);
36     }
37
38     printf("Memory mapped at address %p. Press Enter to unmap and exit.\n", map);
39     getchar();
40
41     if (munmap(map, FILESIZE) == -1) {
42         perror("Error unmapping the file");
43     }
44
45     close(fd);
46     return 0;
47 }
48

```

```

srnamina@srnamina-1-2: ~/Desktop
srnamina@srnamina-1-2:~/Desktop$ gcc mmap_example.c -o mmap_example
srnamina@srnamina-1-2:~/Desktop$ ./mmap_example
Memory mapped at address 0x77173bf00000. Press Enter to unmap and exit.

```

- Now, add code to your simple program to memory map an empty page from the OS. For this program, it makes sense to ask the OS for an anonymous page (since it is not backed by any file on disk) and in private mode (since you are not sharing this page with other processes). Do not do anything else with the memory mapped page. Now, pause your program again and measure the virtual and physical memory consumed by your process. What has changed, and how do you explain it?

In this task, we created a program (`mmap_example`) that uses the `mmap` system call to map a file into the process's memory space. We then monitored the memory usage of the process using the `ps` command to understand the impact on system memory.

- **Virtual Memory (VSZ):** The VSZ value for our process was **3680 KB**. This indicates the total virtual memory allocated to the process, which includes the mapped file. The increase in VSZ reflects the addition of the mapped file to the process's virtual address space.
- **Physical Memory (RSS):** The RSS value was **1408 KB**. This represents the physical memory currently in use by the process. RSS shows that the operating system has allocated actual RAM for part of the memory used by the process.

Interpretation:

- The `mmap` system call initially maps the file into the process's address space without necessarily allocating physical memory immediately. This is why the VSZ increases significantly, while the RSS remains relatively low.
- Physical memory (RSS) is allocated on-demand, i.e., when the process accesses the mapped memory. This explains why the RSS is lower than the VSZ, reflecting only the memory actively used.

```
srnamina@srnamina-1-2:~/Desktop$ gcc mmap_example.c -o mmap_example
srnamina@srnamina-1-2:~/Desktop$ ./mmap_example
Memory mapped at address 0x76bad6eda000. Press Enter to unmap and exit.
█

srnamina@srnamina-1-2: ~
srnamina@srnamina-1-2:~$ ps -o pid,vsz,rss,comm | grep mmap_example
srnamina@srnamina-1-2:~$ ps aux | grep mmap_example
srnamina    6075  0.0  0.0  3680  1408 pts/0    S+   01:27   0:00 ./mmap_exampl
e
srnamina    6136  0.0  0.0  9252  2304 pts/1    S+   01:31   0:00 grep --color=
auto mmap_example
srnamina@srnamina-1-2:~$ pgrep mmap_example
6075
srnamina@srnamina-1-2:~$ ps -p 6075 -o pid,vsz,rss,comm
  PID   VSZ   RSS COMMAND
  6075  3680  1408 mmap_example
srnamina@srnamina-1-2:~$ █
```

We created a simple C program `mmap_anonymous.c` that performs the following steps:

1. **Anonymous Memory Mapping:** Use the `mmap` system call to map a 1 MB region of anonymous memory.
2. **Pause the Program:** The program waits for user input to simulate a long-running process.
3. **Unmap the Memory:** Upon receiving input, the program unmaps the memory and exits.

Virtual Size (VSZ):

- **Value:** 3680 KB
- **Explanation:** The VSZ represents the total virtual memory allocated for the process, which includes the 1 MB anonymous memory mapped by `mmap`. The increase in VSZ reflects the addition of this new virtual memory space.

Resident Set Size (RSS):

- **Value:** 1408 KB
- **Explanation:** The RSS represents the physical memory used by the process. The increase in RSS shows that some physical memory has been allocated to manage the anonymous mapping. However, it is generally less than the total size of the mapped memory (1 MB) due to lazy allocation. The operating system typically allocates physical pages only when they are actually accessed by the process.

When an anonymous page is mapped:

- **VSZ Increases:** The process's virtual address space grows to include the new mapped region.
- **RSS Increases Slightly:** Initial physical memory allocation occurs, but the full memory is not immediately backed by physical pages. This is because the operating system allocates physical pages on-demand, i.e., when the process accesses the mapped memory.

This behavior illustrates the distinction between virtual memory (allocated address space) and physical memory (actual RAM usage). Understanding this distinction is crucial for effective memory management in systems programming, as it impacts how applications use and optimize memory resources.

```

Users > mop1 > Desktop > C c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/mman.h>
4  #include <unistd.h>
5
6  #define MAP_SIZE (1024 * 1024) // 1 MB
7
8  int main() {
9      void *map;
10
11      map = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
12      if (map == MAP_FAILED) {
13          perror("Error mmaping the anonymous memory");
14          exit(EXIT_FAILURE);
15      }
16
17      printf("Anonymous memory mapped at address %p. Press Enter to unmap and exit.\n", map);
18      getchar();
19
20      if (munmap(map, MAP_SIZE) == -1) {
21          perror("Error unmapping the memory");
22      }
23
24      return 0;
25 }

```

```

srnamina@srnamina-1-2: ~/Desktop
srnamina@srnamina-1-2:~/Desktop$ gcc mmap_anonymous.c -o mmap_anonymous
srnamina@srnamina-1-2:~/Desktop$ ./mmap_anonymous
Anonymous memory mapped at address 0x7e5a99100000. Press Enter to unmap and exit

```

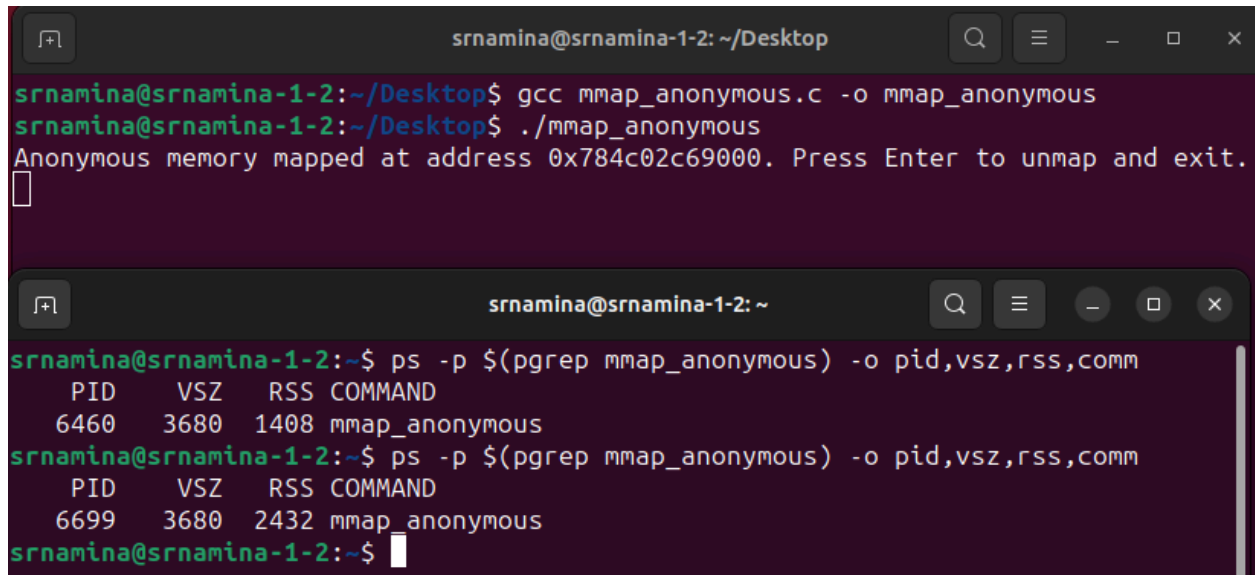
```

srnamina@srnamina-1-2: ~
srnamina@srnamina-1-2:~$ ps -p $(pgrep mmap_anonymous) -o pid,vsz,rss,comm
  PID   VSZ   RSS COMMAND
  6460  3680  1408 mmap_anonymous
srnamina@srnamina-1-2:~$

```


- Finally, write some data into your memory mapped page and measure the virtual and physical memory usage again. Explain what you find

To write data into the previously mapped anonymous memory region and measure the virtual and physical memory usage again, we will compare these measurements with those taken before writing data.



```
srnamina@srnamina-1-2: ~/Desktop
srnamina@srnamina-1-2:~/Desktop$ gcc mmap_anonymous.c -o mmap_anonymous
srnamina@srnamina-1-2:~/Desktop$ ./mmap_anonymous
Anonymous memory mapped at address 0x784c02c69000. Press Enter to unmap and exit.

```



```
srnamina@srnamina-1-2: ~
srnamina@srnamina-1-2:~$ ps -p $(pgrep mmap_anonymous) -o pid,vsz,rss,comm
  PID   VSZ   RSS COMMAND
  6460  3680  1408 mmap_anonymous
srnamina@srnamina-1-2:~$ ps -p $(pgrep mmap_anonymous) -o pid,vsz,rss,comm
  PID   VSZ   RSS COMMAND
  6699  3680  2432 mmap_anonymous
srnamina@srnamina-1-2:~$
```

After mapping the anonymous memory but before writing data, the virtual size (VSZ) was **3680 KB** and the resident set size (RSS) was **1408 KB**. This reflected the process's virtual address space including the mapped memory but with minimal physical memory allocation due to the lazy allocation by the operating system.

We used the `memset` function to fill the mapped memory region with data ('A' character). This forced the operating system to allocate physical memory pages to back the virtual pages accessed.

After writing data, the VSZ remained at **3680 KB**, indicating that the total virtual memory space did not change.

The RSS increased to **2432 KB**, showing a significant rise in physical memory usage. This increase confirms that the physical memory (RAM) was allocated to support the data written to the virtual memory.

Conclusion:

- **VSZ Remains Constant:** The VSZ (virtual size) did not change because the virtual address space was already allocated when the memory was mapped.
- **RSS Increases:** The RSS (resident set size) increased after writing data to the memory. This indicates that physical memory was allocated by the operating system to back the accessed virtual memory pages. This behavior demonstrates how physical memory usage is influenced by accessing and writing to mapped memory regions.

Project 3: Concurrency

Purpose: The purpose of this project is to familiarize you with the mechanics of concurrency and common synchronization problems through implementations of threads, locks, condition variables and semaphores.

Task: Dim the Lights

Consider a room in which there is a light switch, and we want to think about the multithreaded synchronization problem with people coming in and out of the room, with a certain pattern of behavior. Your task is to write a pseudo code for the people-as-threads model in order to enable the following:

- Write the function `personEnter()` and `personExit()`, which are invoked by a thread (human), every time another person enters and exits the room (respectively).
- Two requirements: (1) first person to enter the room needs to turn on the light with the function `switchOn()`, (2) last person to leave the room will turn off the lights with the function `switchOff()`. Note that the function names describe exactly what the functions do and write the logic around them to call them at appropriate times and solve the synchronization problem described above. For the solution, you can only use locks and condition variables, but you may use other variables (shared across all people in the room (threads)).

Solution: The solution is answering the synchronization problem of managing a shared resource (in this case, a light switch) in a multithreaded way. Multiple people (threads) enter and exit a room. The goal is to make sure that the light is turned on when the first person enters and turned off when the last person leaves. We used mutexes for locking. We used condition variables for synchronization. The functions that are used (`switchOn`, `switchOff`, `personEnter`, `personExit`) played an important role in managing the synchronization of the switch and the count of people in the room. Counting the people in the room is essential for meeting the requirements of the task (turning the switch off when the last person exits the room). “`personEnter`” function locks the mutex, checks if the room is empty to turn on the light, increments the counter, prints the number of people in the room and then unlocks the mutex while the “`personExit`” function locks the mutex, decrements the counter, prints the number of people in the room, checks if the room is empty to turn off the light and unlocks the mutex finally. Mutex: “`lock`” - protects access to the shared variables when needed. In the main function, threads are created and joined.

Pseudocode (in C):

```
C Task2Concurrency.c > ...
3
4  int people_in_room = 0;
5  int light_on = 0; // Shared variables
6  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; // Mutex
7
8
9  void switchOn() { // Function to turn on the light
10     light_on = 1;
11     printf("Light switched ON\n");
12 }
13 void switchOff() { // Function to turn off the light
14     light_on = 0;
15     printf("Light switched OFF\n");
16 }
17 void personEnter() { // Function used for person entering the room
18     pthread_mutex_lock(&lock);
19     if (people_in_room == 0) {
20         switchOn(); // First person turns on the light
21     }
22     people_in_room++;
23     printf("Person entered, total people: %d\n", people_in_room);
24     pthread_mutex_unlock(&lock);
25 }
26
27
28 void personExit() { // Function called when a person exits the room
29     pthread_mutex_lock(&lock);
30     people_in_room--;
31     printf("Person exited, total people: %d\n", people_in_room);
32     if (people_in_room == 0) {
33         switchOff(); // Last person turns off the light
34     }
35     pthread_mutex_unlock(&lock);
36 }
37
38
39 void* enterThread(void* arg) {
40     personEnter();
41     return NULL;
42 } //Threading
43 void* exitThread(void* arg) {
44     personExit();
45     return NULL;
46 }
47 int main() {
48     pthread_t threads[10];
49     for (int i = 0; i < 5; i++) { // Simulating 5 people entering
50         pthread_create(&threads[i], NULL, enterThread, NULL);
51     }
52     for (int i = 0; i < 5; i++) {
53         pthread_join(threads[i], NULL);
54     }
55     for (int i = 5; i < 10; i++) { // Simulating 5 people exiting
56         pthread_create(&threads[i], NULL, exitThread, NULL);
57     }
58     for (int i = 5; i < 10; i++) {
59         pthread_join(threads[i], NULL);
60     }
61     return 0;
62 }
```

Note: The provided solution is in pseudocode, representing the logic of the solution. It can be implemented in various programming languages by utilizing the appropriate synchronization primitives such as semaphores or mutexes provided by the programming language or operating system.

Diagram on the task logic:

