Project 2: Memory

T1: *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

Linux implements the POSIX-standard `mmap()` system call for mapping files into memory. `mmap()` is a system call that can be used by a user process to ask the operating system kernel to *map* a file into the memory (address space) of that process. The `mmap()` system call can also be used to allocate memory (an anonymous mapping), which was the subject of the first assignment. It is important to remember that the mapped pages are not actually brought into physical memory until they are referenced. Therefore, `mmap()` lazily loads pages into memory (also 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);
```

As you can see in the function prototype, there exist arguments which are used together with this one system call. And their explanation is right after this part.

Arguments of mmap() system call

There are six arguments to the `mmap()` system call:

- `addr` - This argument is a hint to the operating system kernel to use this address at which the virtual mapping should start in the virtual memory (the virtual address space) of the process. The value can be specified as NULL to indicate that the kernel can place the virtual mapping anywhere it sees fit. If not NULL, then addr should be a multiple of the page size. - znaci to be divisible
- `length` - This argument specifies the length as 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 various 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 machine instructions.
- flags - Various options controlling the mapping. Some of the more common `flags` values are described below:
    - `MAP_ANONYMOUS (or MAP_ANON)` - Allocate anonymous memory; the pages are not backed by any file.
    - `MAP_FILE` - The default setting; it need not be specified. The mapped region is backed by a regular file.
    - `MAP_FIXED` - Don't interpret addr as a hint: place the mapping at exactly 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 - This flag provides the same behavior as MAP_SHARED except that MAP_SHARED mappings ignore unknown flags in *flags*. By contrast, when creating a mapping using MAP_SHARED_VALIDATE, the kernel verifies all passed flags are known and fails the mapping with the error EOPNOTSUPP for unknown flags. This mapping type is also required to be able to use some mapping 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, then fd should be given as -1.
- offset - If this is not an anonymous mapping, the memory mapped region will be populated with data starting at position offset bytes from the beginning of the file open as file descriptor fd. Should be a multiple of the page size.

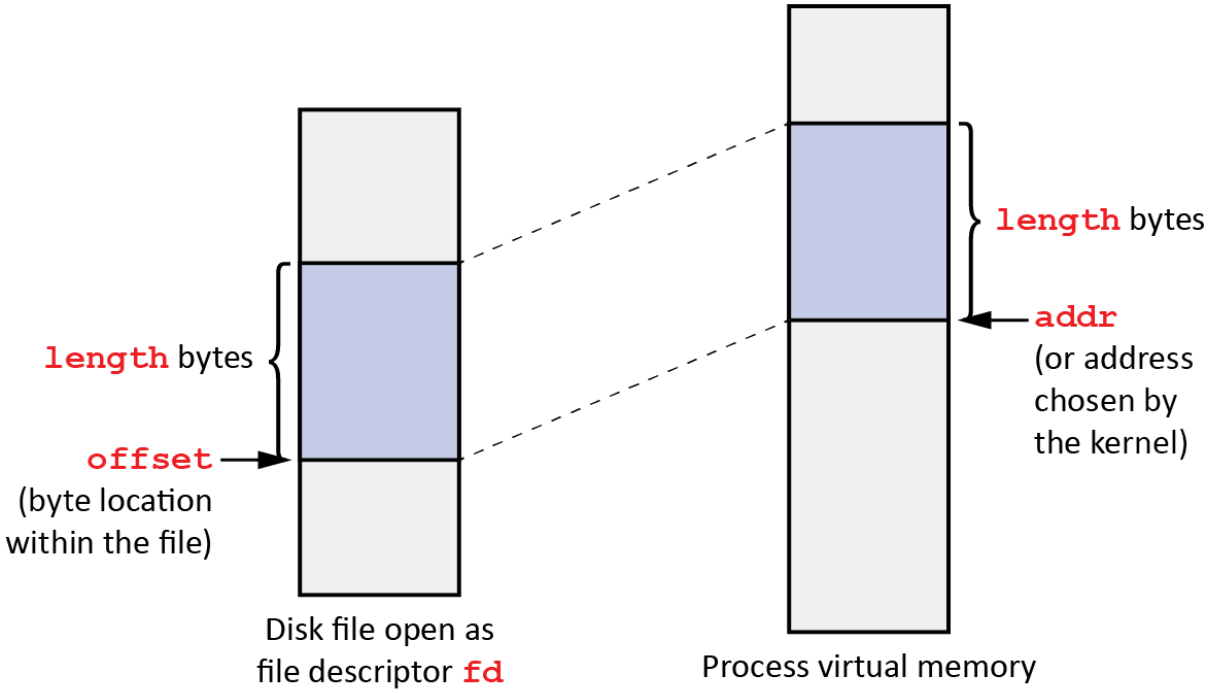
On success, `mmap()` returns a pointer to the mapped area in virtual memory. On error, the value `MAP_FAILED` (i.e., `(void *)(-1)`) is returned and errno is set to indicate the reason.

In order to demonstrate to you, our practical understanding of these system calls, we have written a C Code that uses these system calls. The name of the file is **Functionalities_of_mmap_and_munmap.c,** and at the end of the file we also summarized what we did. Here, you can also find the screenshot of the result.

```
ajla@ajla-VirtualBox: ~
ajla@ajla-VirtualBox:~$ gcc -o map map.c -Wall
ajla@ajla-VirtualBox:~$ ./map
Value before unmapping: 42
Segmentation fault (core dumped)
ajla@ajla-VirtualBox:~$
```

Here, you can also see the illustrative principle of mmap() system call.

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



Obtaining pages & Dynamic Allocation

The mmap system call can be used to allocate memory dynamically by requesting anonymous mappings and then managing those mappings manually. This can be done by using the returned pointer from mmap and applying memory allocation techniques like creating data structures within the mapped memory to manage chunks. One example for obtaining pages of memory from the operating system and allocating chunks dynamically when requested would be:

1. First open a file or use the anonymous mapping option (then you would use -1 for fd) to request memory from the OS. If you're using a file, open it using open() system call and obtain the file descriptor.

2. Use the mmap() system call to map the desired number of pages into the process's virtual memory address space. You should use following arguments with the mmap():

- Set the addr parameter to NULL to let the system choose a suitable address.

- Specify the length as the size of the memory region you want to allocate in bytes. For example, if you want to allocate 4 pages (4 * page size), set the length to 4096 * 4.

- Set prot to the desired memory protection flags using constants like PROT_READ, PROT_WRITE, and PROT_EXEC.

- Set flags to MAP_PRIVATE if you want a private copy of the mapped memory, or MAP_SHARED if you want to share the mapping with other processes. If you want an anonymous mapping, use MAP_ANONYMOUS flag.

- If you're using a file, pass the file descriptor (fd) obtained from open(). For anonymous mapping, pass -1 as fd.

- Set offset to the offset within the file where the mapping should begin. For anonymous mapping, set it to 0.

3. The mmap() call returns a pointer to the mapped memory region. You can assign this pointer to a variable for later use.

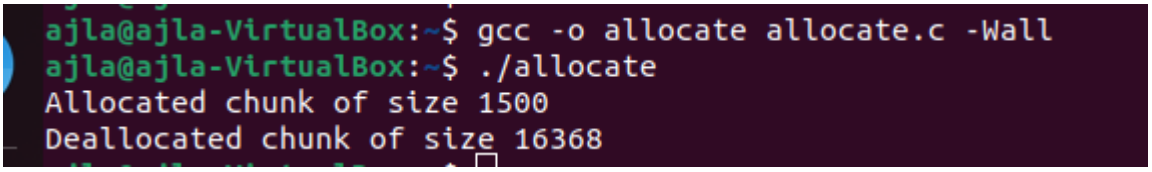
4. To allocate chunks from the mapped memory region, you can implement your own memory management mechanism. For example, you can use a linked list or a bitmap to keep track of free and allocated chunks within the mapped memory.

5. When a chunk is requested, search for a free chunk using your chosen memory management mechanism. Once a free chunk is found, mark it as allocated and return a pointer to that chunk to the caller.

6. To release a chunk and make it available for reuse, mark it as free in your memory management mechanism.

7. Finally, when you're done with the memory mapping, use the munmap() system call to release the entire mapped memory region. Pass the starting address of the mapped memory region and its length as arguments to munmap().

To give you an example of this, we have also provided you with a C file (Obtaining_pages_and_allocating_chunks.c) which shows you the simple code for these operations. Here, we have used the principle of a linked list to show you how to manage free and allocate chunks within mapped memory. Or implement a simple memory management mechanism using a linked list. The allocate_chunk() function traverses the linked list to find a free chunk of a suitable size and marks it as allocated. The deallocate_chunk() function searches for an allocated chunk of the specified size and marks it as deallocated.



The munmap system call

munmap() is a system call used to unmap memory previously mapped with mmap(). The call removes the mapping for the memory from the address space of the calling process process:

```
int munmap(void *addr, size_t length);
```

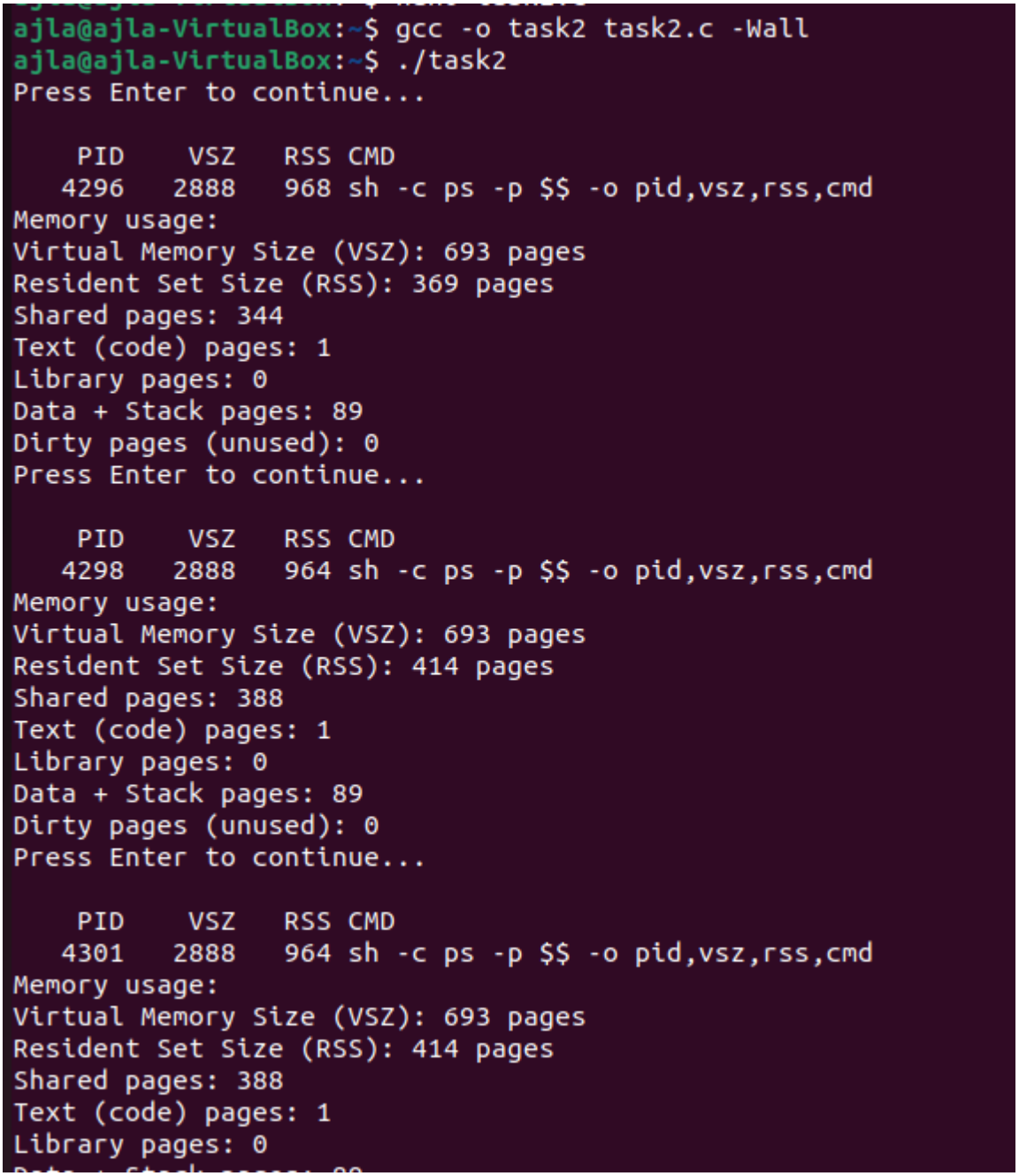
There are two arguments to the munmap() system call:

- addr - The address of the memory to unmap from the calling process's virtual mapping. Should be a multiple of the page size.
- length - The length of the memory (number of bytes) to unmap from the calling process's virtual mapping. Should be a multiple of the page size.

On success, munmap() returns 0, on failure -1 and errno is set to indicate the reason (The <errno.h> header file defines the integer variable errno, which is set by system calls and some library functions in the event of an error to indicate what went wrong) If operation performed by munmap() was successful, future accesses to the unmapped memory area will result in a segmentation fault (SIGSEGV).

-T2: 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.

After writing a C program that runs for a long duration, and can be stopped by the user input, we have measured the memory usage of the process. The C code, you can find in the file **Memory_usage.c**, and here are the results of our observation.



After analyzing the results of this C program, we have concluded that:

1) **PID (Process ID)** changes every time because we are creating new instances of the process, and one PID is a unique identifier of one process. Every time you run the program, a new instance of the process is created, resulting in a different PID for each run. Therefore, it is expected that the PID changes with each execution.
2) **VSZ (Virtual Memory Size)** represents the total amount of virtual memory allocated for the process, including both physical RAM and swap space. In our program, the VSZ always stays the same at 2888. This means since we are running the same program multiple times, that each time same amount of memory is allocated for the process.
3) **RSS (Resident Set Size)** represents the actual physical memory (RAM) used by the process. It includes the pages that are currently in memory. In our first two runs the RSS value decreased from 968 to 964. The decrease in RSS suggests that the process consumed more physical memory during the first run, which could be due to various factors such as memory allocation, data storage, or program execution.
4) **Shared Pages** represents the number of memory pages shared among different processes. In our case, the value changed from 344 to 388 between the first and second run. This indicates that more memory pages were shared among processes during the second run compared to the first run.
5) **Text (Code) Pages** represent the memory pages that hold the executable code of the program. This value remained the same in all our instances of the process which indicates that the executable code did not change between runs.
6) **Library Pages** represent the memory pages used by shared libraries loaded by the program. This value remained the same, suggesting that the shared libraries used by the program did not change.
7) **Data + Stack Pages** represent the memory pages used for storing data and the stack of the program. This value remained the same, indicating that the data and stack usage did not change significantly between runs.
8) **Dirty Pages (Unused)** represent the memory pages that have been modified but not yet written back to disk. In our case, this value remained the same(zero), indicating that there were no significant modifications to the memory pages.

   Generally speaking, the behavior which we observed, where only the PID and RSS values changed between runs, is expected. The other values, such as VSZ, shared pages, and memory segments, remained the same because the code and memory usage pattern of the program did not vary significantly across runs.

PROJECT 3 CONCURRENCY
tobaco task

The code provided in the question does not solve the synchronization problem correctly. The main issue is that the agent selects two ingredients randomly without considering the specific needs of each smoker. This can lead to deadlocks or incorrect synchronization, where a smoker may not be able to acquire the required ingredients to smoke a cigarette.

A correct solution to the cigarette smokers problem can be implemented using semaphores and the following approach:

Define three semaphores: tobacco, paper, and matches, initialized to 0.
Define a semaphore agentMutex initialized to 1 to control access to the agent's code.
Define a semaphore smokerMutex initialized to 1 to control access to the smoker's code.
Define a semaphore doneSmoking initialized to 1.
Define variables to represent the missing ingredients for each smoker.
Create four threads: one agent and three smokers.
Pseudocode for the agent's code:

plaintext
Copy code

```
Agent:
  while true:
    down(doneSmoking)
    agentMutex.acquire()

    // Randomly select two different ingredients
    randomIngredients = getRandomIngredients()
    ingredient1 = (randomIngredients + 1) % 3
    ingredient2 = (randomIngredients + 2) % 3

    // Release the corresponding semaphores for the selected ingredients
    if ingredient1 == 0 and ingredient2 == 1:
      tobacco.release()
    else if ingredient1 == 1 and ingredient2 == 0:
      tobacco.release()
    else if ingredient1 == 1 and ingredient2 == 2:
      paper.release()
    else if ingredient1 == 2 and ingredient2 == 1:
      paper.release()
    else if ingredient1 == 0 and ingredient2 == 2:
      matches.release()
    else if ingredient1 == 2 and ingredient2 == 0:
      matches.release()

    agentMutex.release()
```

Pseudocode for each smoker's code:

plaintext
Copy code

```
Smoker with tobacco:
  while true:
    smokerMutex.acquire()
    paper.acquire()
    matches.acquire()
    // Smoke the cigarette
    up(doneSmoking)

Smoker with paper:
  while true:
    smokerMutex.acquire()
    tobacco.acquire()
    matches.acquire()
    // Smoke the cigarette
    up(doneSmoking)

Smoker with matches:
  while true:
    smokerMutex.acquire()
    tobacco.acquire()
    paper.acquire()
    // Smoke the cigarette
    up(doneSmoking)
```

In this solution, the agent ensures that only one smoker can access their code at a time. Each smoker waits for the other two ingredients they need, acquires them using the corresponding semaphores, smokes the cigarette, and signals the agent that they are done. The agent then repeats the process by randomly selecting two different ingredients.

This approach guarantees that each smoker will receive the missing ingredients they need to smoke a cigarette and avoids deadlocks. The use of semaphores ensures proper synchronization among the threads.

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