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

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 3.1: At the Barbershop

Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs and awaits his turn. The barber moves onto the next waiting seated customer after he finishes one haircut. If there are no customers to be

served, the barber goes to sleep. If the barber is asleep when a customer arrives, the customer wakes up the barber to give him a haircut. A waiting customer vacates his chair after his hair cut completes.
Your goal is to write the pseudocode for the customer and barber threads below with suitable synchronization. You must use only semaphores to solve this problem. Use the standard notation of invoking up/down functions on a semaphore variable.

The following variables (3 semaphores and a count) are provided to you for your solution. You must use these variables and declare any additional variables if required.

```
semaphore mutex = 1, customers = 0, barber = 0;
int waiting_count = 0;
```

Some functions to invoke in your customer and barber threads are:

A customer who finds the waiting room full should call the function `leave()` to exit the shop permanently. This function does not return.
A customer should invoke the function `getHairCut()` in order to get his haircut. This function returns when the hair cut completes.
The barber thread should call `cutHair()` to give a haircut. When the barber invokes this function, there should be exactly one customer invoking `getHairCut()` concurrently.