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: 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: Server-Client

Here we are considering an application server which needs to handle requests of many clients. The server can serve only one request (thread) at a time. The other threads (requests) that arrive while the server is busy, must wait to be served using a synchronization primitive (semaphore or condition variable).

To make sure the waiting times are not too excessive, the server keeps a maximum of N requests (threads) in the system, and you may suppose N > 2. The thread that arrives in the system will have to first check if N other requests are already in the system: if yes, the thread will exit without waiting and return an error value to the client, by calling the function `thread_exitFailure()`. This function will terminate the thread and it does not return.
When a thread is ready to be served, it must call the function `receiveService()`. No more than one thread should call this function at any point in time. This function blocks the thread during the service time.
Note that, while the thread which is receiving the service is blocked, other arriving threads must be free to join the queue, or exit if the system is overloaded.
After a thread returns from `receiveService()`, it must enable one of the waiting threads to seek service (if any are waiting), and then terminate itself successfully by calling the function `thread_exitSuccess()`. This function terminates the thread and does not return.

Your task is to write pseudocode of the function to be run by the request threads in this system. For a solution, you can only use locks and condition variables, but you may use other variables if your solution asks for it (clearly state all the variables used and their initial values at the start of your solution).