Introduction

TABLE OF CONTENTS

For this portion of the homework, you'll need to extend Pintos with the `sbrk` system call, so that your dynamic memory allocator can request memory from the operating system.

Compilation

To build the code and run the tests for this part:

```
cd ~/code/personal/hw-memory/pintos/src/memory
make check
```

Skeleton

Code for this part can be found in the `pintos/src` subdirectory of the `hw-memory` directory.

This homework builds on the Pintos userspace that you implemented in Project 1 (among some other functionality implemented which is mentioned below). Unlike Project 1, however, this is an individual assignment. You should NOT share code with your group members to complete this assignment! To simplify the logistics, we are providing a minimal implementation of the requisite functionality from Project 1 to complete this assignment. This implementation is built on an older version of Pintos, and is not a thorough implementation of the functionality from Project 1 — as such, you should complete this assignment by building on the userspace implementation that we provide you in the starter code, NOT using your group's implementation from Project 1. That said, the nature of this assignment's dependency on Project 1 is that, while certain features of the Pintos userspace must be working in order for this assignment to be well-formed (e.g. you have to be able to start a user process before it can request heap memory from the kernel), the code you write for this assignment will not depend directly on the implementation of Project 1 features.

The starter code implements the following.

1. The one-line `do-nothing` hack needed to complete the Project Pregame.
2. The `write` system call for file descriptor 1 (`stdout`).
3. The ability to `open`, `read`, `write`, and `close` a *single* file per process.
4. Argument validation for the provided system calls using the page fault handler.
5. The implementation for malloc/calloc/realloc/free in Pintos.

Notably, the provided starter code does not provide the ability to pass arguments on the command line, exec a new process, or open multiple files. We chose to implement only the above mentioned items to minimize the amount of code you would need to look over and simplify some of the implementation (discussed below).

The following are some files you may work with throughout this assignment.

`threads/palloc.c`

> Page allocator.

`threads/vaddr.h`

> Helper functions for working with virtual addresses in Pintos.

`userprog/process.c`

> Loads ELF binaries, starts processes, and switches page tables on context switch. You should be familiar with this code based on your experience in Project User Programs.

`userprog/pagedir.c`

> Manages the page tables. You probably won't need to modify this code, but you may want to call some of these functions.

`userprog/syscall.c`

> This is a basic system call handler that implements the system calls mentioned above.

`lib/user/syscall.c`

> Provides library functions for user programs to invoke system calls from a C program. Each function uses inline assembly code to prepare the syscall arguments and invoke the system call.

> We do expect you to understand the calling conventions used for syscalls.

`lib/user/stdlib.S`

> Provides library functions for the C standard library linked into Pintos user applications, namely implementing `malloc`, `calloc`, `realloc`, and `free`.

`lib/syscall-nr.h`

> This file defines the syscall numbers for each syscall.

You should carefully read through the functions in `thread/vaddr.h` and `userprog/pagedir.h`, as many of them will be useful to you in this assignment.
