Project 1

第一个任务是重新实现`timer_sleep`函数。系统之前的实现是忙等待。为了提高效率，我尝试了几种写法，最终采用了优先级队列的写法。我首先在`thread`结构体里添加了`wakeup_time`变量，用来记录该线程睡眠后什么时候被唤醒。每次调用`timer_sleep()`时，我将线程的`wakeup_time`大小作为排序标准，有序地插入`sleep_list`队列中。每次`thread_tick()`时，我都从头开始检查时候当前的`timer_ticks()`是否大于`wkeup_time`，如果大于则不再检查，否则就将`sleep_list`中的该线程唤醒。这样避免了每次`thread_tick()`都要把所有线程都检查一遍所造成的效率上的降低。

第二个任务是实现优先级调度。我首先更改了`ready_list`中的插入，保证了每次插入`ready_list`时是按照线程优先级大小来插入的，使得`ready_list`是一个优先级队列。对于优先级捐赠，我首先在thread结构体里加入了`old_priority`，用来记录捐赠前的优先级；`locks`用来记录该线程持有的锁；`lock_waiting`该线程等待的锁。Lock结构体里加入了`max_priority`，用来记录所有请求锁线程的最大优先级。我在`lock_acquire`中通过循环递归实现优先级捐赠，并且维护`lock`中的`max_priority`。最后把sema和cond的序列按锁的最大优先级排好序即可。

第三个任务是实现多级反馈调度。我这个就按照doc中所写的那样，在`thread`结构体里加入了`nice`和`recent_cpu`变量，在全局加了`load_avg`。我自己写了一个浮点数运算逻辑，每个`timer_tick()`更新`recent_cpu`，每个`TIMER_FREQ`更新`load_avg`，每4个`timer_tick()`更新priority。

Project1我一个人从开始看doc到真正写完花了一周多近两周的时间，因为我尝试了多种写法，我觉得我最终的版本写法比较简洁，效率也很高。之中也遇到了很多问题，比如第二个任务的优先级捐赠没有递归捐赠下去，刚加进来的线程没有比较优先级，以及第三个的浮点数逻辑运算等等。但我还是通过与预期结果比对，gdb等调试方式成功找到问题最终顺利写完。

Project 2

---- Terminating Message ----

This is a very easy part of project 2, the only need to do is to find where every process exit. So just by searching the name we can find the function `process_exit()`. Then we just add a line of printf code to output the name of the process before we finally exit and do schedule.

---- Argument Passing ----

This is also a part which the implmentation is almost determined accurately by the document. We find `process_execute` and a few other functions where we need the name of the process. And we can use the `strtok_r()` function to parse the input and then get the real file name of the process to be exxcuted and start the process. For the arguments to be pushed into stack, we need to do that when the process start so we do this in `start_process()`. In this function, we get all the arguments by using the `strtok_r()` function and push them in to the stack.

This is the very first part we need to look at the frame, we find that we only need to substract the esp pointer and we will get the answer.

---- System Call ----

This is the hardest part in project 2.

We are supposed to get some syscall in `syscall_handler()`, and call some functions already written. And we need to slightly change a few functions to make it run in the correct order.

I use `syscall_handler()` to get a syscall, and check if all the argument passed here can be fetched and is not given some kernel address. In this part, I parse the frame given and check the possible number of arguments(i.e. 1,2,3), and in the same time, we get them in int type to make coding easier. By using `is_user_addr()`, we get check if they follow the memory. After checking, we send them to the correspondent function in `syscall.c` to finish doing them.

For `halt`, we call `shutdown_power_off()`.

For `exit`, we need to erase all the things we created new.

For `wait` and `exec`, we need to make sure that even if the child process has already died before we continue in father process, we can still get the answer. So we use a list to save the information of the son. And by using semaphore, we can make it truly wait for the child process. For `exec()`, we only need to use `process_create()` and make sure we have saved the things I mentioned above. In this part, all the syscalls which is related to file system is very easy, we only need to use a lock to make sure we do not use them in the same time. And then we only need to call the functions.

---- Denying Writes to Excutables ---- When we start a process, we have to deny write to it, so we do this in `load()` function where we load an excutable to kernel and make it run. And when a thread exit, we change it to writeable.

Project 3

过程中将面向用户程序的虚拟页、面向整个系统的帧设计如下结构设计

```
struct page {
    void* key;                  //页的地址
    void* value;                //帧的地址 或 交换区位置 或 文件位置
    enum page_status status;    //描述该页当前位置
    bool writable;              //是否指出写回
    void* origin;               //文件状态
};

struct frame {
    void* frame;                //帧的物理地址
    void* upage;                //与用户程序而言的页地址
    struct thread* t;           //所属进程
    bool swapable;              //是否允许进入交换区
};
```

在构建功能的过程中需要增加一些syscall函数方便使用。例如用来注册并建立映射的mmap、以及用来释放的unmap。

工具齐全之后对于置换算法使用了时钟算法、思想和内容比较简单清晰、性能也算有效。

在完成结构和算法的搭建之后就可以着手换掉之前访问存储空间的算法、使为了防止因实现的错误导致之前的功能不能使用，用`#ifdef VM`的方法保证原功能不被毁坏

在实现过程中，前期因为对工程理解不多，多处功能实现时因没有`acquire_lock()`和`free_lock()`导致了一些系统崩溃。此后还因对前面一些工程的不熟悉而进行了重复的劳动或是遗漏了一些需要更改的部分。