Student name: Federico Watkins

CSIS 3810: Operating Systems

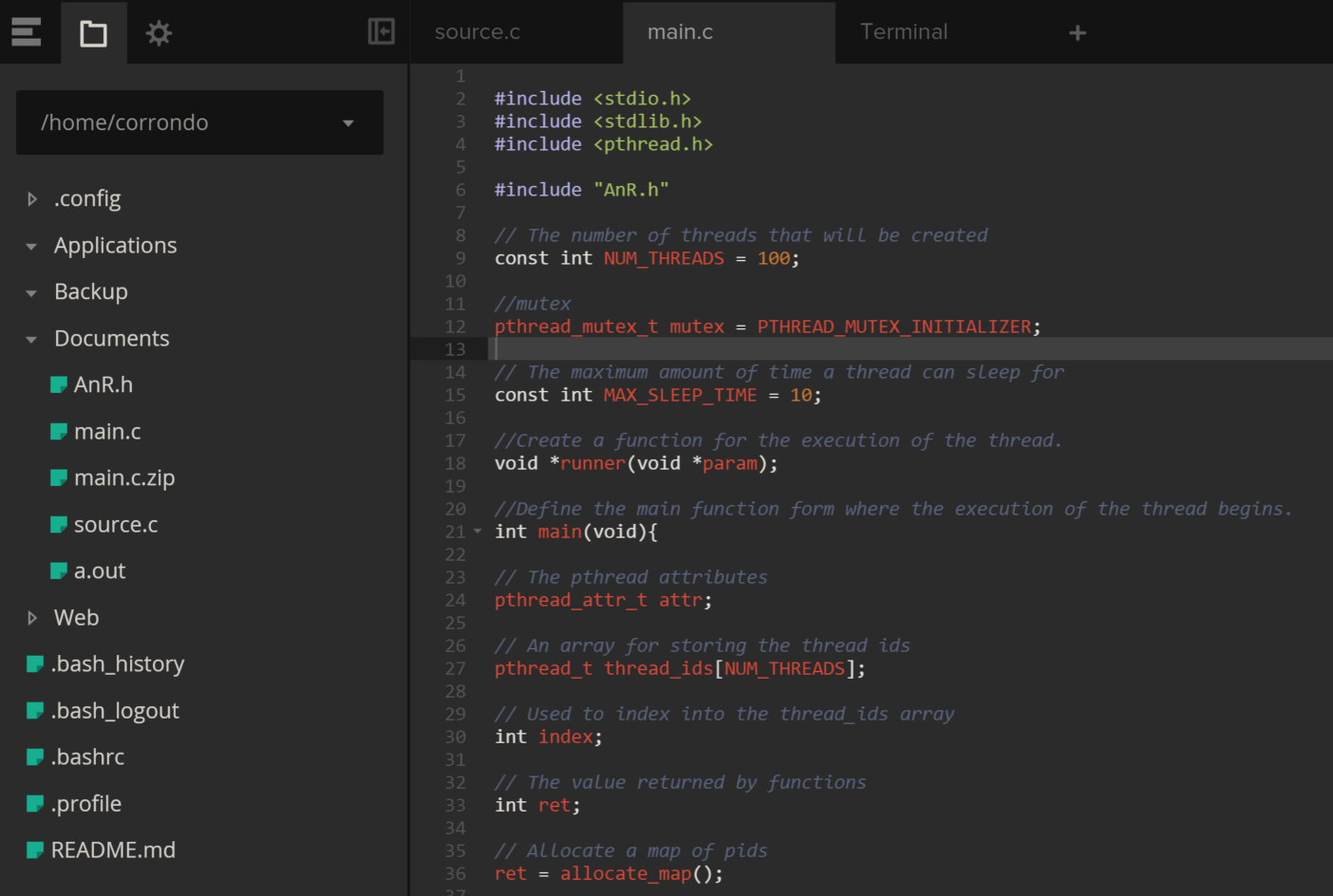
Chapter: 5

Assignment: 3

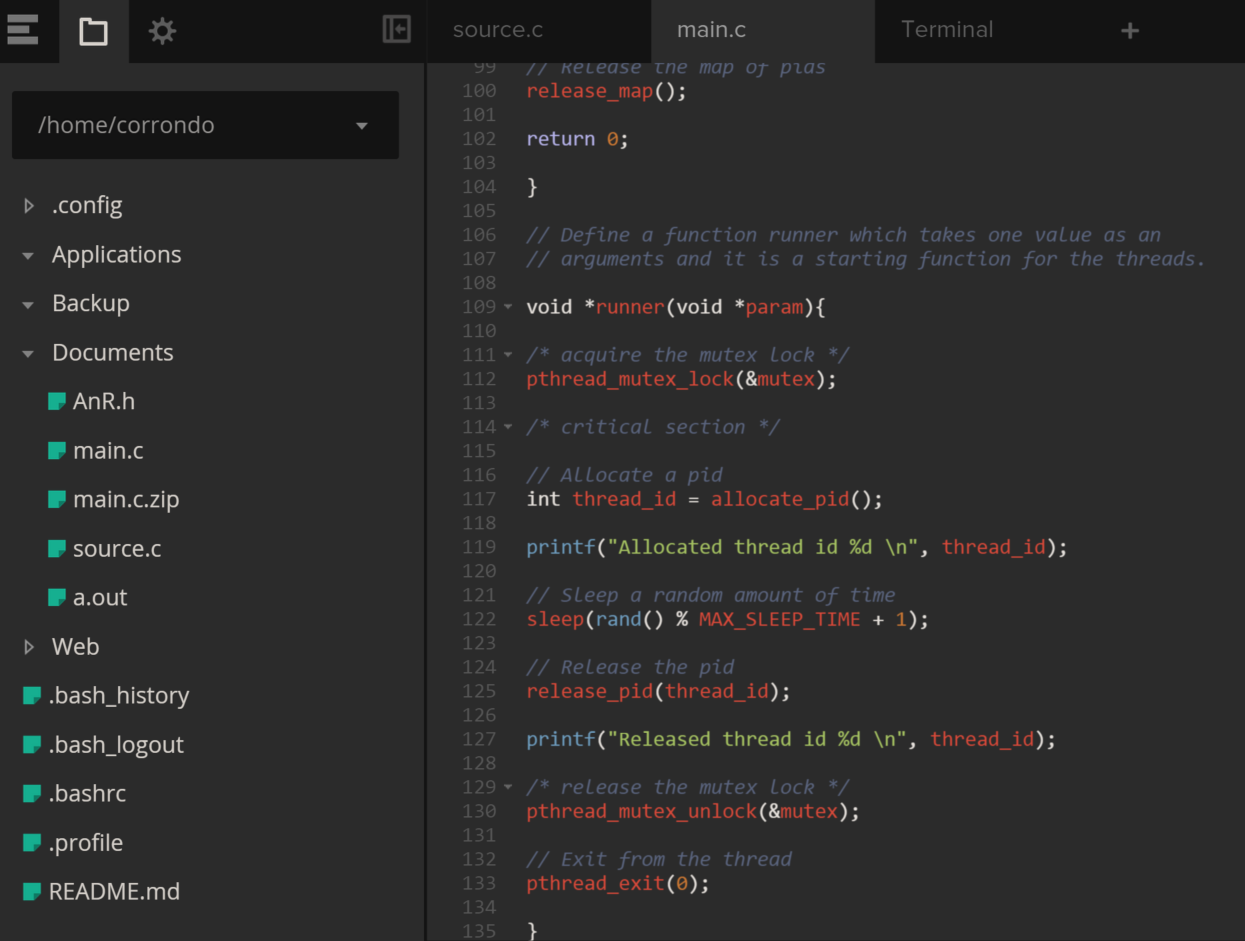
Problem Statement:

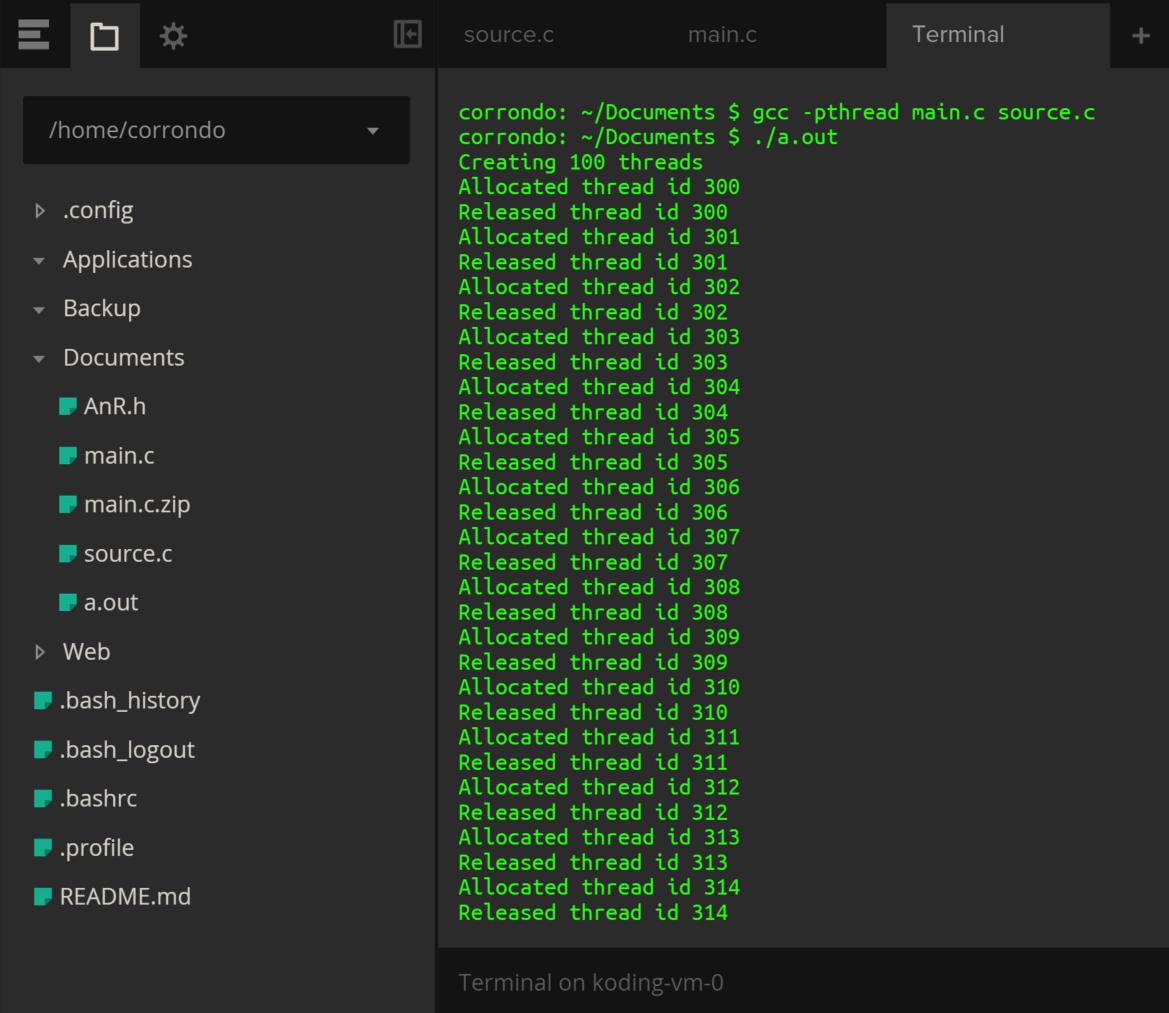
Programming Exercise 3.20 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.20 required you to modify your solution to Exercise 3.20 by writing a program that created a number of threads that requested and released process identifiers. Now modify your solution to Exercise 4.20 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions. Use Pthreads mutex locks, described in Section 5.9.4.

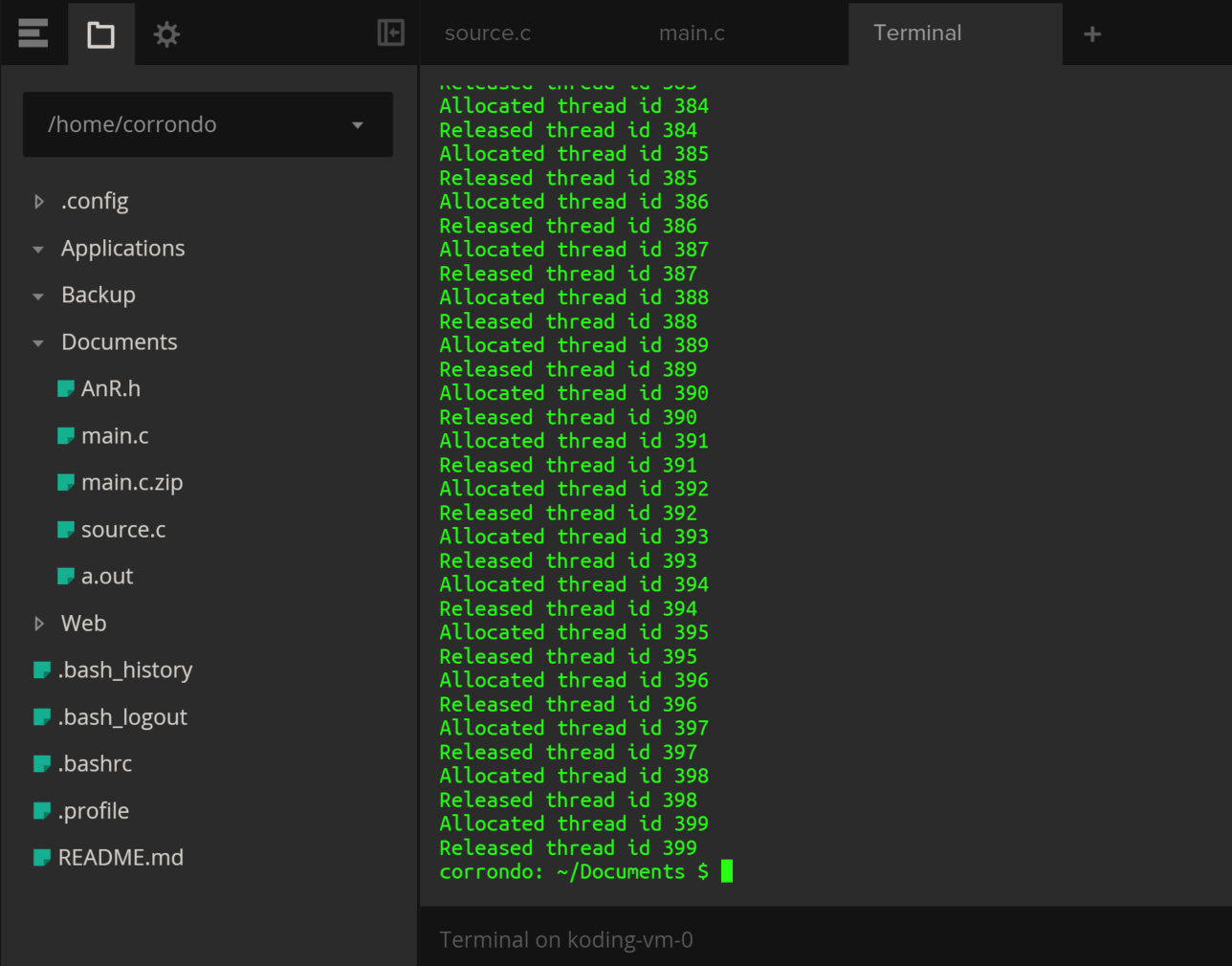
Solution:



1. Using my solution to Exercise 3.20 and 4.20 I modify main.c to include protection for the critical sections of code by means of a mutex. Thus, threads will acquire lock before entering a critical section and releases it upon exiting the critical section. Mutex is implemented with the pthread\_mutex\_init() function. Its first parameter is a pointer to the mutex. The second parameter is NULL which initializes the mutex to its default attributes.



1. The mutex is acquired and released with the pthread\_mutex\_lock() and pthread\_mutex\_unlock() functions. In the above screenshot I invoke pthread\_mutex\_lock() before the critical section of code and pthread\_mutex\_unlock() after.
2. 



To ensure that the code is functioning, here I enter the commands gcc -pthread main.c source.c to run the files. As shown in the screen clipping, an output file is created (./a.out). Once entered into the command line to run it begins to create 100 threads. It also reports that it is allocating and releasing thread id’s. All id’s are unique for every thread. As opposed to allocating all thread id’s at once and then releasing all thread id’s at once as in programming problem 4.20, here the mutex causes each thread id to be allocated and released individually.

References:

Operating System Concepts 9th Edition, Ch. 5, p. 212-237

http://man.yolinux.com/cgi-bin/man2html?cgi\_command=pthread\_mutex\_init

---Things learned in Chapter 5:

Chapter 5 Covered detailed information on classical process-synchronization problems and the tools that are used to solve them. The critical-section problem was presented and different solutions were discussed. The fact that each process has a segment of code considered to be its critical section in which the process may be changing common variables, updating a table, writing a file, and so on. I learned about mutual exclusion, progress and bounded waiting which are the requirements that a solution to the critical-section problem must satisfy. Within the assignment we explored a software solution. Through in-class lectures as well as the book’s chapter reading we learned about hardware solutions. Covered semaphores which is an integer variable that is accessed only through atomic operations wait() and signal() and is one of the more sophisticated ways for processes to synchronize their activities. Semaphore’s implementation, although sophisticated, does result in a situation where two or more processes are waiting indefinitely for an event, this is known as deadlocked. Another related problem to deadlocks is starvation or indefinite blocking, in which case processes wait indefinitely within the semaphore. Even more complex is the use of high-level synchronization constructs such as monitor types.