Braden Fisher and Evan Brown

Professor Barker

CSCI 3310

18 April 2021

Project 3: Thread Library Writeup

Introduction

The goal of this project was to implement the user-level thread library that was previously provided for the disk scheduler project. The library was to provide the exact same API that was used by the disk scheduler (outlined in the thread.h header file) and to support any concurrent program that uses this API. Given some C++ program, the thread library would compile with this program and provide it with functionalities to support synchronization, such as creating new threads, forcing the currently-running thread to yield the CPU to the next available thread, acquiring and releasing locks, and waiting, signaling, and broadcasting on condition variables (following Mesa semantics). The thread library was not supposed to produce any output (other than "Thread library exiting." when all threads finish executing), which meant that a major component of the project would be to handle all errors silently, whether they be due to misbehaving user programs or memory or hardware errors coming from resources used by the operating system. The other major challenges of this project were to understand the Linux infrastructure that supports user-level thread libraries and to ensure atomicity within the thread operations in order to provide clients with safe synchronization.

Major Design Choices/Data Structures Used

One of the first decisions to make was how to represent the threads, locks, and condition variables, and we decided to define structs for each. Every thread contains a pointer to a ucontext_t struct, which is how Linux stores the information pertaining to a particular thread (stack, program counter, etc). We decided to put this within a “Thread” struct because we knew that each thread should also have some identifier in order to keep track of threads that own locks or are waiting on locks and/or condition variables. Each thread also keeps a boolean variable to indicate whether its execution is finished (which occurs when the function called by thread_create returns). The “Lock” struct also contains an identifier field, where the identifier is the unsigned integer value used by the client to define the lock. We also included a queue of Thread pointers, which would be the queue of threads that have tried to acquire the particular lock but are now waiting because another thread currently owns it, which is also stored as a Thread pointer within the Lock struct. The ConditionVariable struct contains both an identifier for the condition variable and an identifier for its associated lock, as the specification declares that a condition variable is defined by a unique tuple of (lock number, condition variable number). This struct also contains a queue in which all threads that have called thread_wait with this condition variable will be placed while they are “asleep” and waiting to be signaled.
We decided to use structs for each of these components because it was the simplest way to store all of the information and create new threads, locks, and condition variables with the arguments provided by users.

Outside of these structs, we used a queue of Thread pointers to implement the ready queue, which is the queue of threads that are ready to run but are not running. We also defined lists of Lock and Condition Variable pointers that contain all of the locks and condition variables created by the user; this was done so we would be able to determine if a reference to a lock or

condition variable referred to an existing lock/condition variable or if a new one needed to be created, and also to be able to iterate through each one and delete them when the thread library is finished so there are no memory leaks. Finally, we defined one ucontext_t pointer as the "main thread," which would create the first thread and then run until the ready queue is empty, therefore being the "scheduler" and what determines when the library is finished.

To ensure safe synchronization for the client, we made use of the provided interrupt.a library and made calls to the functions defined in the interrupt.h header file. At the beginning of each function within the thread library, interrupts are disabled so that the entire function runs atomically and that context switches occurring inside of the thread library code will not cause synchronization issues for the user. The major challenge was to ensure that interrupts would be enabled whenever user code runs, as user code is less trustworthy. We ensured that whenever an error was caught, interrupts would be enabled before -1 was returned. We also put calls to interrupt_enable() at the end of each thread library function and after the calls to swapcontext() that are in thread_yield, thread_lock, and thread_wait so that when the calling thread, which gets put to "sleep" or goes to the back of the ready queue, is awoken and runs again, the first thing it will do in its execution is enable interrupts before returning from the library function back to user code. With this approach, any thread in the ready queue that is run by the main thread will enable interrupts before it runs the user code.

Implementation and Structure

The following is a high-level description of our thread library code, which can be found in the thread.cc file. This library is meant only to support the use of threads in other programs

and thus does not have a main function or any other code that is automatically run. We implemented the following library functions:

thread_libinit(): This function initializes the thread library and will run until all threads are finished. It must be called before any other thread functionality can be used and cannot be called more than once. It takes in as input a user function and an argument to be passed to it; these will be used to create the first thread using the thread_create() function, which thread_libinit() will then context switch to. If any aspects of this process go awry, it will return -1, otherwise 0. After all of the threads have finished, this function uses the delete_current_thread() helper function to clear the threads' memory. It also clears all the memory held by the create locks and condition variables. Finally, it prints out "Thread library exiting\n".

thread_create(): This atomic function creates a new thread and places it on the ready queue with a given function to be called and a parameter to be called with. It can only be used successfully after thread_libinit() has already been called (any such errors will return -1, otherwise it will return 0). If there is sufficient memory, the function will create a Thread struct and initialize a corresponding ucontext pointer using the getcontext() function. This thread is assigned a stack array and an ID number, along with other necessary intializations. The new thread will not run immediately: instead, it will be placed at the end of the ready queue, from which it will eventually be given permission to run.

thread_yield(): This atomic function causes the currently running thread to give up the CPU and give control to the next thread on the readyQueue. If the readyQueue is empty, the function will do nothing, otherwise it will swap context to the next thread and push the current thread to the back of the readyQueue. Returns 0 on success and -1 on failure.

thread_lock(): This function is the equivalent of the acquire() method we used in class, as it is called when a thread is attempting to acquire a lock. If the lock is free, threads are automatically granted the lock, otherwise they are put on a lock-specific waiting queue from which they will eventually be granted access. Any thread which attempts to acquire a lock it already owns will return an error (-1)--the same is true if there is no memory left for the lock and its corresponding thread (returns 0 otherwise). This function uses the find_lock() helper to identify if a requested lock number has already been initialized; it will allocate a new lock and corresponding queue if not.

thread_unlock(): This function is called by a thread that wishes to release a lock. It relies heavily on the unlock_interrupts_disabled() helper function to accomplish this task--this work was put inside a helper as it is also useful to the thread_wait() function. This helper must be run with interrupts disabled, and it makes sure that the requested lock exists and that the releasing thread owns the lock. If these conditions are met, this helper will "wake up" the next thread in the lock queue and allow it to acquire the lock--if there is no thread waiting, the lock simply becomes free. As with other functions, returns 0 on success and -1 on failure.

thread_wait(): This function takes a lock and a condition variable (CV) and is called by a thread that wishes to "wait" on that CV. It uses the find_cv() helper to check whether the requested cv/lock pair exists, initializing a new CV and associated queue if not. The calling thread is put on the CV's waiting queue, from which it will be awoken when thread_signal() or thread_broadcast are called. The thread also releases the lock using the unlock_interrupts_disabled() helper described above. These actions constitutes "putting the thread to sleep," so we also need to swap context back to the main thread so the program can continue running. After the program is

reawoken, it must reacquire the lock, which it does as part of the return statement of this function.

Thread_Signal(): This function is called when a thread wants to “wake up” threads which are waiting on a certain CV. It uses the find_cv() helper to locate the CV (if it exists--under Mesa semantics it is not an error to signal a non-existant CV), and it then takes the first thread off the CV’s queue and puts it on the global readyQueue (i.e. “wakes it up”). As with other functions, returns 0 on success and -1 on failure.

Thread_Broadcast(): This function operates very similarly to thread_signal(), except that it “wakes up” every thread waiting on a CV instead of just one. Threads are put on the readyQueue in the order in which they appeared in the CV’s queue.

Evaluation:

We used a total of 20 test programs in order to test our own thread.cc implementations, as well as to expose the 13 faulty libraries we were tasked with. These test programs were meant to utilize the provided thread functionality such that a faulty library would produce different output than a correct library. Some of our test cases were simply implementations of classic synchronization problems (toomuchmilk.cc, producerconsumer.cc) which were meant to tackle a broad range of basic implementation errors. Particularly productive in this respect was our adaptation of the disk.cc file from Project 2, which had to be adapted so as to not require inputs. That complex program, with several threads dealing with a shared data structure and very specific output orders, exposed an impressive 10/13 buggy libraries. This rendered many of our other test cases, which were simpler and targeted specific suspected behavioral bugs (signalnohold.cc,

signalnowait.cc, thirdthread.cc, diffthreadrelease.cc, etc.) unimportant as far as exposing the buggy libraries was concerned, but these targeted programs were useful for debugging our own thread library. As for the rest of the buggy libraries, deadlock.cc clearly exposed library K by intentionally creating a deadlock scenario and noinit.cc exposed library M by attempting to use the thread library without calling thread_libinit(). Disksignal.cc exposed library C simply by changing calls to thread_broadcast in our original disk.cc file to calls to thread_signal() (we suspect the bug had to do with thread ordering with the CV queue).

Conclusion:

This project required us to perfectly reconstruct the API of the thread.h library which we had used for synchronization in Project 2. This undertaking required a detailed understanding of the exact specifications of the given functions, as well as a firm knowledge of the Linux infrastructure which supports user-level thread implementations. To support proper synchronization, we also needed to ensure atomicity by disabling and re-enabling interrupts around critical sections of code. In order to test our library (as well as expose several provided buggy libraries), we also wrote a series of test programs that attempted to expose certain bugs in the libraries' handling of threads, locks, and condition variables. We come away from this project with a deep appreciation for the gritty details of implementing user-level synchronization and an understanding of the thorough testing required in complex and high-stakes OS programming.