CSCI 432 Project 1: Threads and Monitors

Erik Kessler
Williams College

Abstract

In this paper, we describe our exploration of concurrency through a multi-threaded disk scheduler and a user-level thread library implementation. We describe the architecture, implementation details, and design decisions of our programs. By the end of the paper, we also explore and evaluate the correctness of our implementation and conclude with a reflection on the project as a whole.

1 Introduction

The thread abstraction allows programmers to create powerful, concurrent programs. Multi-threaded programs have a variety benefits over single threaded programs including allowing for more responsive, more efficient, and simpler programs. However, writing effective programs requires careful consideration of synchronization, or controlling how the threads interleave. The first part of this project, the disk scheduler, served as an introduction to writing concurrent programs using locks and monitors to ensure synchronicity. The disk scheduler also serves as a good way to test our thread library implementation. At a high level, the disk scheduler takes I/O requests from multiple sources, determines an order to service them, and reports to the requester when the request is complete. Building this requires creating multiple threads to serve as the scheduler and requesters along with using locks and monitors to protect the shared data from race conditions.

The second part of the project involved writing our own implementation of the thread library. While the disk scheduler helped us understand how to use threads, locks, and monitors, the thread library forced us to understand, at a more fundamental level, how locks work. Additionally, the thread library forced us to consider things like ensuring atomicity and handling errors gracefully. Lastly, testing is an important practice in good software design, and this project helped develop this skill as we had to create a comprehensive test suite to test our implementation. The thread library we implement has a function for creating new threads along with an implementation of Mesa monitors which include functions such as lock, unlock, wait, signal, and broadcast.

2 Architectural Overview

We start this section with an overview of the disk scheduler architecture and go on to discuss the structure and design of our thread library implementation. The disk scheduler starts by reading in the input for the disk scheduler. This step includes reading a file, for each requester, that indicates which tracks it will request. We store this list of tracks as a vector and encapsulate it in a `Requester` struct. Once we have our list of Requester objects, we initialize the thread library and pass all the requesters. This parent thread first creates a `Scheduler` struct which hold information about the number of live requesters, max number of requests that can be on the queue, and position of the disk reader's head. We then create a new thread that executes the *startScheduler* function with the Scheduler passed as the argument. After creating

the scheduler's thread, the parent starts each Requester by starting a new thread running *startRequester* with the Requester passed as the argument. Figure 1 illustrates this setup.

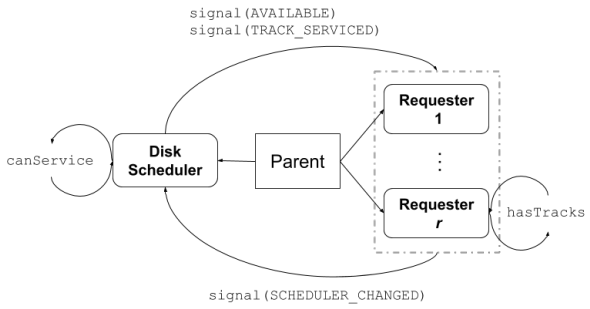

Figure 1: The high level structure of the disk scheduler

First we describe how requesters act. A requester iterates through each track in its list of tracks. For each track, the requester tries to create a request, but if the scheduler is full, it will *wait* for the `SCHEDULER_AVAILABLE` condition. Since the scheduler is shared, we use a lock, `SCHEDULER_LOCK`, to protect any access to the scheduler. When the scheduler is available, the Requester creates a `Request` struct which holds the track and requester's id. This request is then pushed to the queue and the Requester *signals* `SCHEDULER_CHANGED` which will wake the scheduler. The Requester then *waits* on the condition variable equal to the track it wanted serviced. This process is repeated for each track, and when there are no tracks left, the Requester is done so it decrements the number of live requesters, *signals* the change, *unlocks*, and finishes.

In an infinite loop, the Scheduler first checks if it can service a track meaning the queue is as full as it can possibly get. Again, since the scheduler is shared, we protect it with the `SCHEDULER_LOCK`. If it cannot service a track, the scheduler *waits* for a `SCHEDULER_CHANGED` signal which occurs when a new request is added or the number of live requesters changes. If the number of live requesters went to 0 the scheduler returns which exits the whole program. In the case where there a track can now be serviced, the scheduler selects the request that is closest to the current head position, prints that that request was serviced, *signals* which track was serviced to wake up the requester, and *signals* the the scheduler is available.

Now we discuss the structure of our thread library. First, we write a function *setupThread* that uses the *getcontext* and *makecontext* methods to setup a new ucontext with a new stack for the thread. We also wrap the function the user wants to run in our own private *start* function that allows us to control the lifecycle of the thread. We encapsulate threads with a `Thread` struct which holds the ucontext, stack pointer, and information about locks the thread is waiting for or holding and condition variables the thread may be waiting for. The setupThread function is used in *thread_libinit* to create the parent thread and we use *swapcontext* to transfer control to this newly setup thread. Similarly, *thread_create* uses *setupThread* but instead of immediately swapping in the new thread, the newly created thread is put on the ready queue.

As previously mentioned, we wrap the functions the user wants to run in our own start function. This start function first checks if there is a thread that just completed that should be deleted. When threads finish we swap them from the *runningThread* variable to the *deleteThread* variable with marks them for deletion. After checking for deletion, the start function runs the user's function with the specified arg. When that function completes, a function, *runNextThread*, pops the next thread off the ready queue to become the running thread while the current thread is sent to be the delete thread. If we ever go to run the next thread on the ready queue and the queue is empty, we exit the thread library.

Yielding simply requires pushing the current running thread to the back of the ready queue and running the next thread on the ready queue.

Lastly, we describe our implementation of Mesa monitors. To start, Figure 2 shows how the different thread library calls move threads around. We have already discussed the running thread and ready queue, but we have two additional data structures for storing which threads are waiting (a result of calling *wait*) and which threads are locked (a result of calling *lock*). First we describe how locking and unlocking work. Globally, we store a list of all locks currently held. When a thread requests

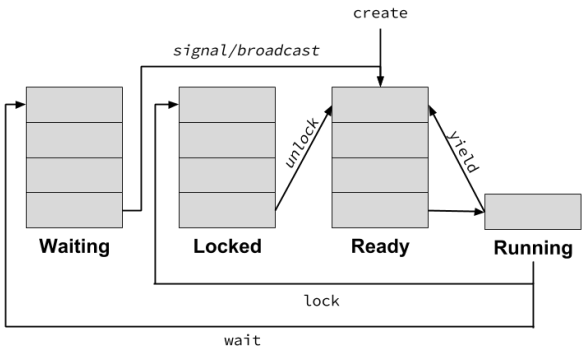

Figure 2: The high level structure of the thread library state transitions

a lock, we check that global list to see if the lock is already held. If not, we can immediately add the lock both the global list and the thread's internal list of locks held. If not, the thread is put on the locked queue and a flag is set indicating the lock the thread is waiting for and the next thread is run. Calling unlock removes the lock from the list of globally held locks and the thread's list of held locks. We then iterate through the locked queue looking for the first thread that is waiting on that lock. If such a thread is found, the thread is given the lock and put on the ready queue. Note, although we could have used a separate queue for each lock we use a single list of locked threads. We get the same behavior FIFO behavior without having to manage a queue for each lock.

Next, wait works by using a wait list. When a thread call wait, we put the thread on the back of the wait list and run the next thread. We also store a `Monitor` struct with the thread indicating what condition variable, lock pair it it waiting on. Again, instead of having a queue for each condition variable, we just have a single wait list that we will iterate through.

Signal and *broadcast* are what wake a thread and put back on the ready queue. Signal and broadcast both search the wait list for all threads with a monitor matching the monitor specified by the signal/broadcast call. In the case of signal, only the first match is woken up and moved the the ready queue. For broadcast, all matches are moved to the ready queue. When waiting threads get their chance to run again, they will try to acquire the lock in the same process as described above.

Lastly, we discuss some final considerations. To ensure automaticity, we disable interrupts whenever the thread library code is running and enable them before we return to the user's execution. There are also many places where errors can arise, so our code contains checks for user errors like misuse of locks and monitors along with other errors such as low memory or bad function calls.

3 Evaluation

Testing the correctness of our programs played an important role as bugs or unexpected behavior in these core pieces of software could be disastrous if part of an actual operating system. Thus we created a suite of tests to rigorously verify the correctness of both the disk scheduler and thread library.

Testing the scheduler involved creating different sets of tracks to request and checking that the requests got serviced as expected. We then added tests with preemption turned on which tried to uncover bugs by creating different interleavings of threads that might expose race conditions.

To test our thread library, we had a set of 17 small tests that each tested a specific area of the thread library. We also used our disk scheduler as more end-to-end test that worked the system as a whole. Our small test cases included testing the basic functionality of locking, unlocking, waiting and signaling. We tested that it handles errors such as calling init twice or not calling init first or trying to run a null function. Some tests verified everything was in FIFO order. Some tests verified interrupts were reenabled correctly. One of trickier tests verified that thread deletion was working properly by monitoring how much memory was being used by the program using *getrusage*. Another test verified bad allocations were handled by restricting the memory then creating a bunch of threads.

4 Conclusion

This project helped me come to have a deeper understanding of threads and monitors as well as a greater appreciation for the complexity that is involved in a thread library.

The scheduler showed how threads can be useful when you want to have multiple agents in your program doing things simultaneously. However, the scheduler also showed the subtle bugs that can arise in concurrent programs, but I feel like I learned some good practices around locks and monitors that help reduce such bugs.

The thread library demonstrated the benefit of encapsulating complexity and exposing a simple, consistent interface to the user. It also showed the importance of having a library that is well tested and works as expected so the user can be sure it is an issue with their code and not an underlying problem with the library.

Overall, I enjoyed the project and found it interesting to actually implement threads. Tracking down bugs was difficult at times but rewarding in the end as they led to a very good understanding of how things work.