Group 11-05: Daniel Schroeder, Brian Ozarowicz, and Luke Morrison

Assignment II: I/O Elevators

CS 444 Operating Systems II
Spring 2017
4 May 2017

Abstract

This document is a summary of Assignment 2 for CS 444 Operating Systems II at Oregon State University Spring 2017. This document includes a log of commands used to complete the kernel assignment, a description of the concurrency assignment solution from Group 11-05, and the responses to the concurrency assignment follow up questions.

I. Kernel Assignment

BLAH BLAH BLAH ABOUT KERNEL ASSIGNMENT

Listing 1. The new sstf_add_request() function in sstf-iosched.c

```c
// This is the original function from noop-iosched.c:
static void noop_add_request(struct request_queue *q, struct request *rq)
{
    struct noop_data *nd = q->elevator->elevator_data;

    list_add_tail(&rq->queuelist, &nd->queue);
}

//This is the new function in sstf-iosched.c:
static void sstf_add_request(struct request_queue *q, struct request *rq)
{
    struct sstf_data *nd = q->elevator->elevator_data;
    /* While sector of new request is larger than current sector check next request in queue */
    while (blk_rq_pos(rq) > blk_rq_pos(&nd->queue)) {
         nd = nd->queue.next;
         /* Stop when the next sector is larger than the new request */
         if (blk_rq_pos(&nd->queue) < blk_rq_pos(nd->queue.prev))
                 break;
    }
    /* Add new request after the last smaller sector */
    list_add_tail(&rq->queuelist, &nd->queue);
}
```

II. Concurrency Assignment

Write-up

Our concurrency solution implemented pthreads and multithreading to solve a producer/consumer style problem. The problem: with a buffer of size 32, have producers add items (structs with int value number and int value wait time) to the buffer. The producers must wait a random number of seconds (time to produce) and also wait if the buffer is full. The consumers take the items from the buffer, sleeping for the time given in the buffer item struct's wait value, and blocking if the buffer is empty. To do this we implemented mutexes, pthread conditions, and concurrent programming.

Our producers, once the thread is created, get a random number using either rdrand from x86 assembly instructions or the implemented mt19937 function and sleep for that amount of time (the time it takes to produce). When done sleeping the thread will lock the mutex which gives it exclusive access to the buffer. It then enters a loop to see if there are any available indexes in the buffer. If the buffer is full the producer thread waits for a consumer to throw the pthread condition for "full" meaning a consumer has opened up a space on the buffer. Once it is able to add its item to the buffer it throws a pthread condition "empty" to signal that the buffer is no longer empty, prints out the number, waits the time for the item it is adding, unlocks the mutex, and exits.

Similarly, our consumer threads lock the mutex and enter a loop checking for an item in the buffer to consume. If the buffer is empty it will wait for a producer thread to throw a pthread condition that the buffer is no longer empty. Once there is an available item in the buffer the consumer stores its number and wait time in local variables and sets both struct fields to -1 (implying it is now an empty space) and unlocks the mutex. The consumer then waits the given amount of time, prints a message that it has consumed the specified number, then exits.

Our main function mallocs enough size for the buffer and thread arrays, checks to see if rdrand is available to use, and initializes the mutex and pthread conditions. It then uses for loops to create and pthread_join all the threads and wait for them to exit. The pthread condition variables allow the threads to synchronize by locking and unlocking the mutex until the specified condition is satisfied.

The program accepts a command line parameter as the number of threads to produce. If the program is executed with `./concurrency1 100` it will create 100 producer and 100 consumer threads. On our own local machines this program works for N >1000 threads, but on os-class it segfaults after about 240 threads, which we assume to be due to limitations on server permissions to mediate the amount of threads a single user can produce.

What do you think the main point of this assignment is?

The main point of the assignment was to gain more experience with dealing with potential race conditions and how to allocate processing time to different threads. To deal with with race conditions, we were expected to write a semaphore implementation or to use mutexes to make sure critical variables are only being manipulated by a single thread at a time. As an extra, this assignment was also about giving unique information to different threads, as each had to be aware which fork was to the "left" and "right" of each thread.

How did you personally approach the problem?

To keep track of which forks were being used, we simply made a global array that would hold a boolean value to indicate whether the fork was in use or not. We also created a struct for each thread that would contain the index of the left fork, the index of the right fork and the name (ID) of the thread. This struct was passed into the thread starting point when calling the pthread_create() function.
When the thread starts, it generates a random amount of thinking time and sleeps for that amount of time. Then, it generates a random eating time and checks if the forks adjacent to it are available to start eating. If not, it waits until they are. The thread "eats" for the amount of time specified and when done, it releases the forks for other threads to use.
Each time the forks array is about to experience a read or write, a global mutex is locked to prevent any race conditions. If the mutex is already locked, then the thread will wait until available according to POSIX implementation.

How did you ensure your solution was correct?

We ensured that our implementation was correct by printing all the actions of each thread directly to standard output. That way, we can clearly see who is thinking or eating and for how long. Furthermore, we print out the state of each fork (available or in use). Finally, we also print thinking times and eating times.

What did you learn?

During this assignment, I learned how to properly pass variables when creating a thread. Although only a single variable of type (void *) can be passed, a structure can be created and typecasted into a void pointer which lets us pass in more arguments. Furthermore, the argument and return type of the function need to be explicitly defined as void pointer.
Another thing I learned is that the memset() function throws out the 'volatile' tag if being used on a volatile variable. I kept using memset() but made sure the compiler optimizations were disabled (just in case) since we are working with a few global variables.
Finally, I learned that it is a good habit to also lock mutexes when reading a shared variable, otherwise you might be reading some wrong information. It's hard to predict when the data will change, and it's safer to freeze the state of critical variables when reading multiple of them so that they are in the same "time state".

III. Version Control Log

Detail	Author	Description
6fc0cad	DSchroederOSU	Initial commit
237aed0	DSchroederOSU	Added assignment 1 folder and starter file
2c475d7	DSchroederOSU	added concurrency 1 file
4126204	DSchroederOSU	working on concurrency1.c. made a skeleton of the project
d862ede	DSchroederOSU	progress, got rdrand to work
a134ada	DSchroederOSU	finished concurrency logic with two threads
7096e2f	DSchroederOSU	DONE I THINK
29c5440	DSchroederOSU	This works for n threads, there was a bug that would not clear the buffer space once consumed
c6ed2db	ozarowib	recitation program
ce1d4cd	DSchroederOSU	added second random generator
870a6cf	DSchroederOSU	Merge branch 'master' of https://github.com/DSchroederOSU/CS444-Group_11_05
17fa4d7	Daniel Schroeder	works on os-class
8efe181	Daniel Schroeder	test
f860768	DSchroederOSU	fixed argc
63f1ea3	Daniel Schroeder	tex progress
e3916b0	Daniel Schroeder	idk
bafbb73	Daniel Schroeder	ahaha
d8ae437	ozarowib	conformed to class style guide
c7bf34a	ozarowib	IEEEtran working, updated Makefile

IV. Work Log

- Brian and Daniel created group directory on os-class at recitation April 11th at 11:00am
- Brian ran script for setting folder permissions on Friday April 14th
- Daniel and Brian met in OSU Library at 1:50 pm on Monday April 17th. Worked until 3:30pm on initial code and write-up document
- Throughout Wednesday April 19th Daniel developed the initial version of the homework concurrency program while Brian wrote the delayed concurrency program from Tuesday's recitation; Brian reviewed Daniel's code Wednesday evening
- Brian and Daniel met afternoon of Thursday April 20th to review and complete concurrency1.c code; found compile errors and rdrand error when run on os-class
- Daniel and Brian implemented mt19937 function into concurrency1.c evening of April 20th so random numbers could be generated on os-class and did final debugging

- Throughout Friday April 21st Daniel and Brian worked on the LaTeX Makefile and formatting
- Brian and Daniel finished the assignment write-up in a Google Doc
- Brian revised the concurrency1.c code to follow the class style guidelines
- Brian transferred the content from the Google Doc write-up to the LaTeX file
- Brian added IEEEtran.cls file to directory so proper formatting could be applied
- Daniel and Brian pushed final changes to GitHub and prepared tarball for submission on TEACH