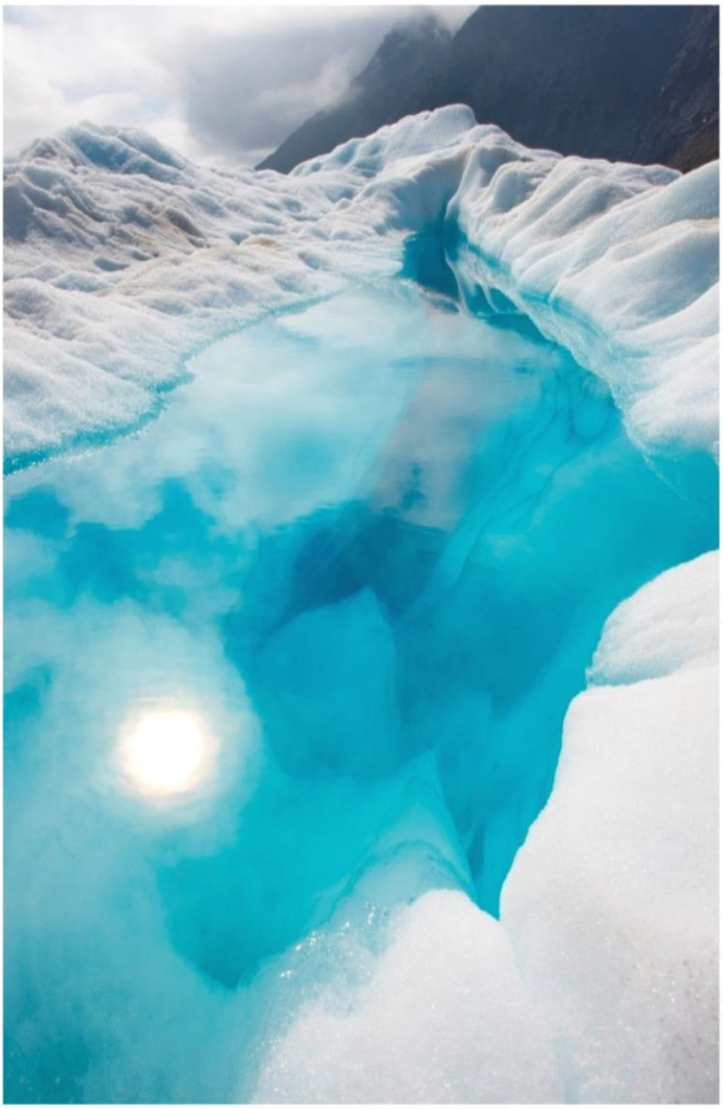


JFS Phase Two

CS536 ADVANCED OPERATING SYSTEM

Mengyuan Zhang | Dr. Sandeep Gupta | April 2, 2016

Overview

The objective of this project is to implement a journal file system using C programming language and to test this file system for the following properties:

1. Error Free environment
2. Error Free environment with multi-threading(before or after atomicity)
3. Error-prone system without multi-threading
4. Error-prone system with multi-threading

Now, we are in the phase two to realize an error free environment with multi-threading journal file system. In this phase, multi-thread is built based on the file architecture implemented in phase one. Moreover, phase two will demonstrate that each threads would run concurrently followed by before or after atomicity.

Multi-threads

The pthread library is used in the code, and for simplicity and convenience, three threads implementation will be showed in the demonstration. The partial code below just realizes this function:

```
int main(){
    pthread_t thread1;
    pthread_t thread2;
    int t0 = 0;
    int t1 = 1;
    int t2 = 2;
    sem_init(&mutex, 0, 1);
    sem_init(&pre_mutex, 0, 1);

    pthread_create(&thread1, NULL,(void *)&func, &t1);
    pthread_create(&thread2, NULL,(void *)&func, &t2);
    func(&t0);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    sem_destroy(&mutex);
    sem_destroy(&pre_mutex);
}
```

Before or After atomicity

To realize before or after atomicity, the lock is used to guarantee that a part of operation in one of the three threads could not be interrupted by other threads. Instead of using a pthread_mutex_lock, I choose to use the binary semaphore which functions similar as pthread_mutex_lock. The partial code below just realized this function.

```
sem_wait(&mutex);
pthread_mutex_lock(&lock);
printf("Thread %d: Do you want to initialize a new action? 1/0\n", thread_num);
scanf("%d", &in);
if(in == 1){
    ac_id = (ac_id+1)%16;
    int i = 0;
    JOURNAL_STORAGE[thread_num][ac_id] = (uchar *) calloc(1, JOURNAL_SIZE);
    printf("New Action is created!\n");
    printf("Thread %d: The action_id is %d\n", thread_num, ac_id);
}
sem_post(&mutex);
```

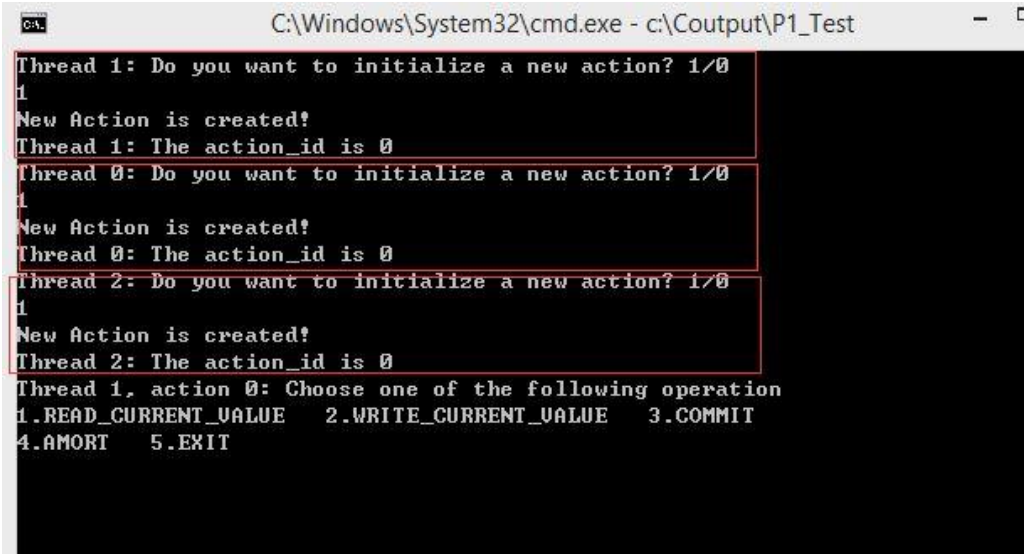
In this usage of semaphore, I just keep the BoA atomicity for the choice operation in each threads so that it will looks better in the demonstration

```
if(in == 1){
    flag = 1;
    sleep(1);
    sem_wait(&mutex);
    while(flag == 1){
        printf("Thread %d, action %d: Choose one of the following operation\n", thread_num, ac_id);
        printf("1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT\n4.AMORT   5.EXIT\n");
        scanf("%d", &input);
        switch(input){
            case 1:{
                uchar file_name[JOURNAL_FILE_NAME_LEN] ={0};
                uchar *file_data = NULL;
                printf("Thread %d: Enter the file name to be read: ", thread_num);
                scanf("%s", file_name);
                printf("\n");
                case 5:{
                    printf("Thread %d Action %d exits now!\n",thread_num,ac_id);
                    JOURNAL_STORAGE[thread_num][ac_id] = NULL;
                    flag = 0;
                }
                break;
                default:{
                    printf("Choose an option from 1 to 6\n");
                }
                break;
            }
        }
    }
    sem_post(&mutex);
    else{
        sleep(1);
    }
```

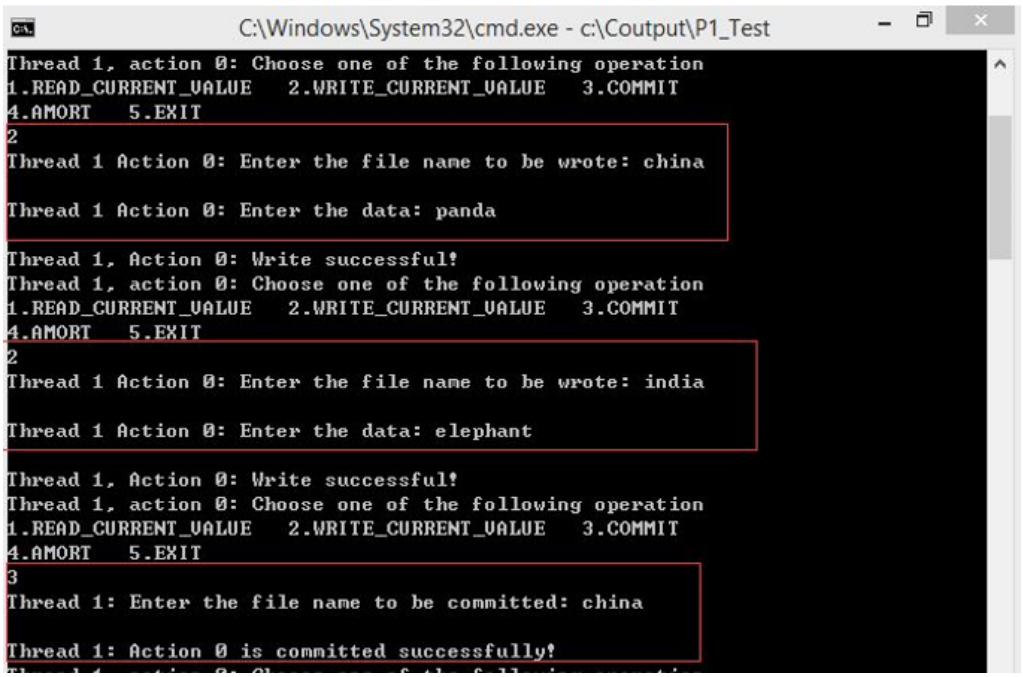
This code is a inner loop of a new action, here the semaphore is used to guarantee that every thread's action is atomic.

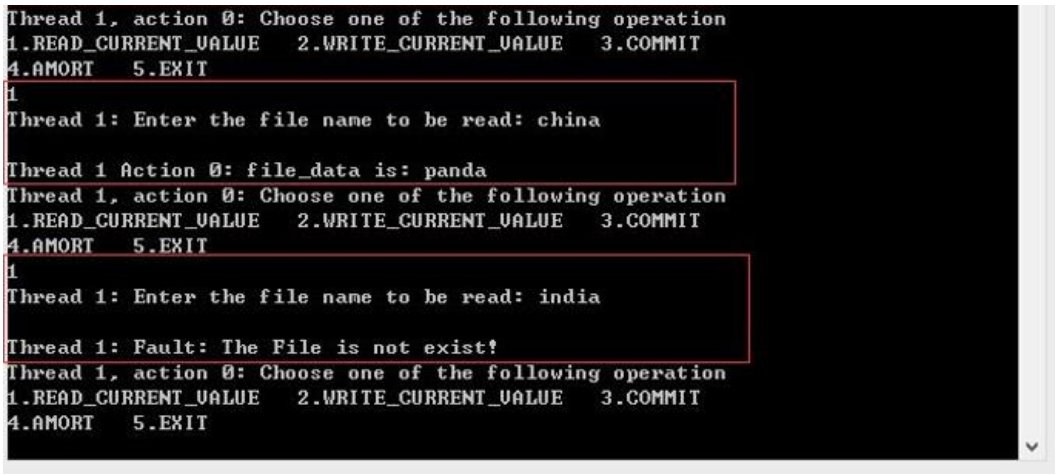
Demonstration

As there are three threads, at the beginning for every thread, the computer would ask you if you want to begin a new action:

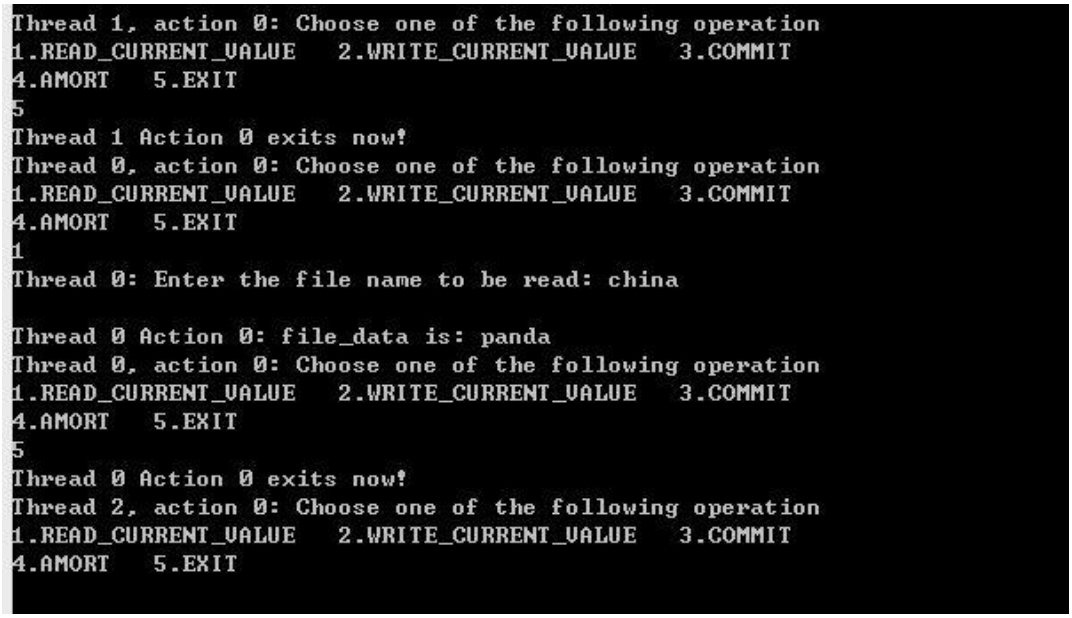


As you enter an action in any of the three threads, now it will atomically execute the command for this action till you command 5(exit). In the execution of action, you can write and read multiple times, and commit or abort every write you executed before by providing the file name you wrote. Every action inherits the AoN atomicity in phase one.





The action of one of the three threads in operation would execute forever and block other threads unless you exit the action. So other threads could execute concurrently without block:



In this demonstration, it shows that three threads are operating healthily and each of threads follows the Before or After atomicity.

Thank you!