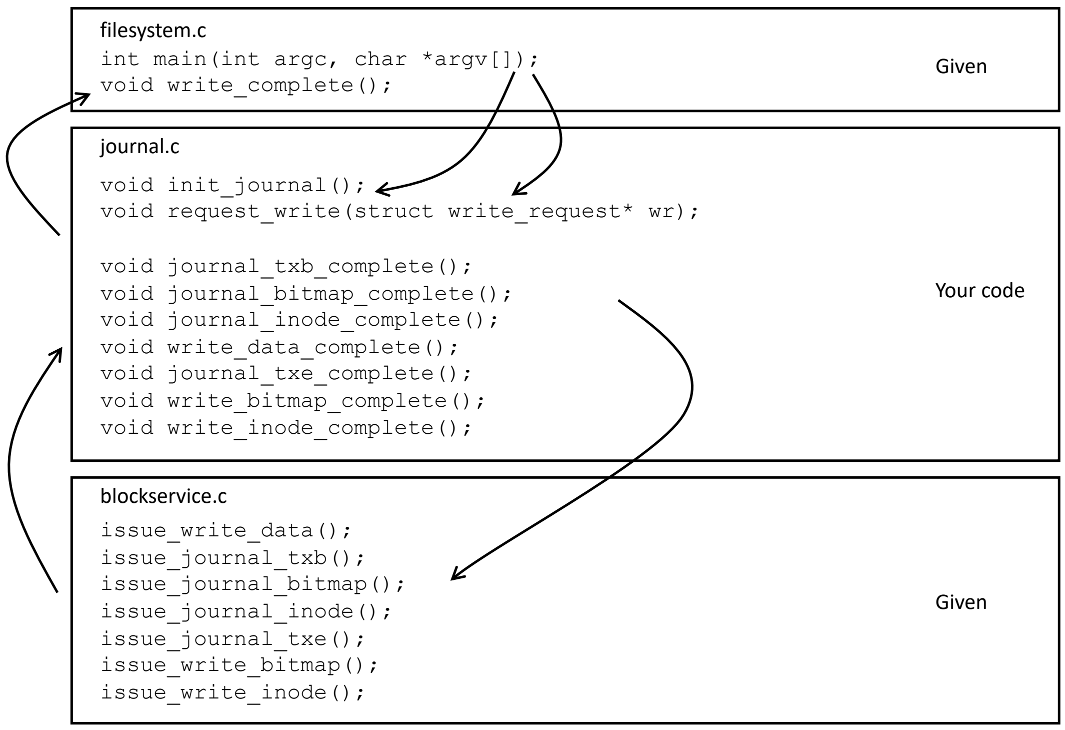
Project 2
COM S 362
Fall 2023

1. Introduction

For this project you are implementing an application in C. The code must work in Linux, specifically it must compile on either the Linux server `pyrite.cs.iastate.edu` or on the VirtualBox image provided with Project 1.

The purpose of this project is to implement a metadata journaling service layer that can be used by a file system to achieve crash consistent recovery. You are not implementing an entire file system, nor are you implementing the low-level code to write individual blocks to storage. Specifically, there are 9 functions to implement in `journal.c`.


filesystem.c
int main(int argc, char *argv[]);
void write_complete();
Given
journal.c
void init_journal();
void request_write(struct write_request* wr);
void journal_txb_complete();
void journal_bitmap_complete();
void journal_inode_complete();
void write_data_complete();
void journal_txe_complete();
void write_bitmap_complete();
void write_inode_complete();
Your code
blockservice.c
issue_write_data();
issue_journal_txb();
issue_journal_bitmap();
issue_journal_inode();
issue_journal_txe();
issue_write_bitmap();
issue_write_inode();
Given

The code you are implementing is used by a file system (`filesystem.c`) and it uses a low-level block layer service (`blockservice.c`) to write to the disk. The above figure shows the general flow of control. The file system initializes the journal and requests writes. The journal layer issues write requests in the block service layer. When an issued request is complete the block service layer calls a complete function in the journal layer. When the write is done complete is called in the filesystem layer.

The following is a fully working solution to the problem (if we don’t care about concurrency bugs).

```
/* This function can be used to initialize the buffers and threads.
 */
void init_journal() {
      // initialize buffers and threads here
}

/* This function is called by the file system to request writing data to
 * persistent storage.
 *
 * This is the non-thread-safe solution to the problem. It issues all writes
 * in the correct order, but it doesn't wait for each phase to complete
 * before beginning the next. As a result the journal can become inconsistent
 * and unrecoverable.
 */
void request_write(struct write_request* wr) {
        // write data and journal metadata
        issue_write_data(wr->data, wr->data_idx);
        issue_journal_txb();
        issue_journal_bitmap(wr->bitmap, wr->bitmap_idx);
        issue_journal_inode(wr->inode, wr->inode_idx);
        // commit transaction by writing txe
        issue_journal_txe();
        // checkpoint by writing metadata
        issue_write_bitmap(wr->bitmap, wr->bitmap_idx);
        issue_write_inode(wr->inode, wr->inode_idx);
        // tell the file system that the write is complete
        write_complete();
}

/* This function is called by the block service when writing the txb block
 * to persistent storage is complete (e.g., it is physically written to
 * disk).
 */
void journal_txb_complete() {
        printf("journal txb complete\n");
}

void journal_bitmap_complete() {
        printf("journal bitmap complete\n");
}

void journal_inode_complete() {
        printf("journal inode complete\n");
}

void write_data_complete() {
        printf("write data complete\n");
}

void journal_txe_complete() {
        printf("jounrnal txe complete\n");
}
```

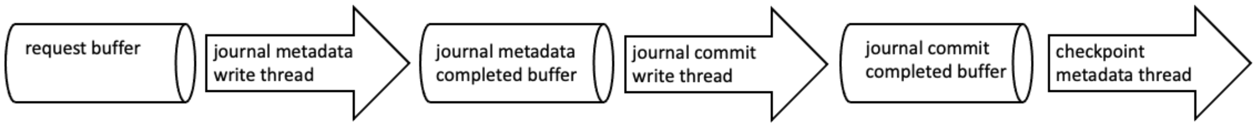
```
void write_bitmap_complete() {
        printf("write bitmap complete\n");
}

void write_inode_complete() {
        printf("write inode complete\n");
}
```

What is wrong with the above solution? The journaling layer is not waiting at the correct times for all data to be written to persistent storage (e.g., the HDD or SSD). This means a crash at the wrong time may result in the filesystem being unrecoverable. From the textbook reading, we have the following steps for a correct implementation of metadata journaling.

1. Data write: Write data to final location.
2. Journal metadata write: Write the begin block and metadata to the log; **wait** for data write (step 1) and journal metadata writes to complete.
3. Journal commit: Write the transaction commit block (containing TxE) to the log; **wait** for the write to complete; the transaction (including data) is now committed.
4. Checkpoint metadata: Write the contents of the metadata update to their final locations within the file system.

To solve the problem, you are required to use the following approach. In `journal.c`, you must implement 3 buffers and 3 threads as shown below.



request buffer
journal metadata write thread
journal metadata completed buffer
journal commit write thread
journal commit completed buffer
checkpoint metadata thread

When the file system calls `request_write()` the request (or the pointer to it) is enqueued into the request buffer. A journal-metadata-write-thread takes a request out of the request buffer (whenever the buffer is not empty), calls the functions to write the data and journal metadata (see example code above), waits for all issued writes to complete, and then enqueues the request in the next buffer. A journal-metadata-commit-write-thread takes a request out of the previous buffer, issues the journal `txe` (transaction end), waits for completion of writing the `txe` block, and then enqueues the request in the next buffer. The checkpoint-metadata-thread takes a request out of the previous buffer, issues writing the metadata, waits for completion of writing the metadata, and then calls `write_complete()`.

2. Requirements

Code requirement: You may not change `journal.h` in any way, this will impact grading. You can modify `journal.c` for testing purposes.

init_journal() (20 points)

The function `init_journal()` is used to create the threads and initialize the buffers. Assume the function is called by the file system (see `filesystem.c`) before calling `request_write()`.

Buffers (20 points)

The buffers are FIFO order. Do not use dynamic memory for the buffers. Each buffer may store a maximum of `BUFFER_SIZE` (defined in `journal.h`) entries. An entry can be either a struct `write_request` or simply a pointer to a `write_request`. A common data structure for this situation would be a *circular buffer*.

Threads (30 points)

For full points the following conditions must be satisfied.

1. Threads must not have any race condition bugs.
2. Threads must achieve maximum available concurrency. This means a thread must be working on a request whenever possible. The only times a thread is waiting is when its input buffer is empty, it is waiting for issued blocks to complete writing to storage, or its output buffer is full.
3. Synchronization must be done with pthread or semaphore constructs (i.e., your threads cannot depend on spinlocks). For more information, consult the man pages for the following.

- `pthread_create`
- `pthread_join`
- `sem_init`
- `sem_wait`
- `sem_post`
- `sem_destroy`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutex_destroy`
- `pthread_cond_init`
- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_destroy`

Testing (20 points)

Create the following test scenario by modifying `block_serivce.c`. Currently, writing blocks to storage completes immediately. Suppose for first time `txe` (transaction end) is written it takes a significant about of time to complete. Simulate this by creating a separate thread that sleeps for 1 second before calling `journal_txe_complete()`. This should cause the journal-metadata-completed-buffer to completely fill up and the journal-metatdata-write-thread to become stuck waiting. In your `journal.c` code print the message "thread stuck because of full buffer" to verify this is the case.

You may complete additional test cases, but at a minimum the above scenario should be in your submission.

Documentation (10 points)

Document `journal.c` with a clear description of the overall approach. How do your threads avoid race conditions and achieve maximum concurrency?

Create a **`README`** file containing the results of testing.

5. Submission

You will submit the project on Canvas. Your program must compile and run without errors on `pyrite.cs.iastate.edu` or on the VirtualBox image used for project 1.

Put all your source files (including the Makefile and the README file) in a folder. Then use command zip -r <your ISU Net-ID> <src_folder> to create a .zip file. For example, if your Net-ID is ksmith and project2 is the name of the folder that contains all your source files, then you will type zip -r ksmith project2 to create a file named ksmith.zip and then submit this file.