CSE 511 Project 2:

Distributed Threads with Distributed Shared Memory System

Prepared by:

Mahmudur Hera – mbr5797@psu.edu
A. Burak Gulhan – abg6029@psu.edu

Implementation Logic

Distributed Locking:

The code for the distributed locking is implemented in psu_lock.h. For distributed locking we used the following message types in mutex.proto

```
message mutex_request {
    int32 lockno = 1;
    int32 sequence_no = 2;
    int32 id = 3;
}

message mutex_reply {
    int32 acknowledgement = 1;
}

message mutex_deferred_reply {
    int32 sender = 1;
    int32 lockno = 2;
}
```

The algorithm is the same as the Ricart-Agrawal algorithm. The mutex_request message is sent to each node when a node wants to enter a critical section and this node receives a mutex_reply message in return. If a node decides to defer, it sends a null mutex_reply (since we need to send a reply in rpc calls), then it sends a mutex_deferred_reply message later.

Distributed Shared Memory:

The DSM is implemented in psu_dsm_system.h. There is also a dir.cc file which is run on the directory node, which is the last node in nodes.txt. This dir.cc program initializes and starts the directory node. We used the following message types in dsm.proto

```
message dsm_request {
    int32 type = 1;
    int64 vaddr = 2;
    int32 id = 3;
    bytes data = 4;
    string name = 5;
}

message dsm_reply {
    int32 ack = 1;
    int32 permission = 2;
    bytes data = 3;
}
```

The message dsm_request is sent whenever an rpc call is made and dsm_reply is what the client receives from the server. In dsm_request the parameters are:

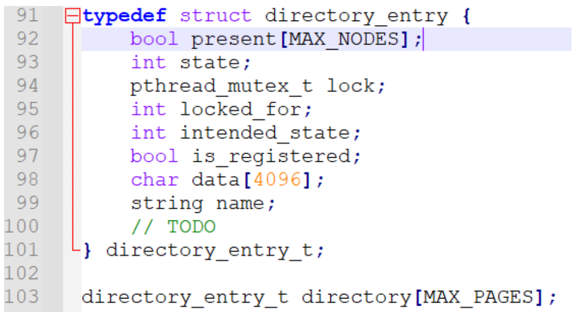
1) type: what kind of request is being made. This can be READ_REQ, WRITE_REQ, ACK, INVALIDATE, UNREGISTER, HEAP_REG
2) vaddr: this is either the virtual address, or is 0 if name is being used (in case of shared heap using psu_dsm_malloc())

3) id: This is the id of a node. Every node has a unique id.
4) data: This stores a page of size 4096 bytes. However, this is not used in all message types.
5) name: This is used to identify pages allocated using psu_dsm_malloc(). If ‘name’ is used then vaddr is set as 0.

For dsm_reply the parameters are:

1) ack: The type of acknowledgement sent. This can be NO_ACK, ACK_NO_DATA, ACK_W_DATA, ACK_W_PERM
2) permission: This stores the permission given. This can be INVALID, READ_ONLY, READ_WRITE, DIRECTORY
3) data: This stores a page table of size 4096 bytes.

The directory table is an array of directory_entry structs. This struct is shown below:



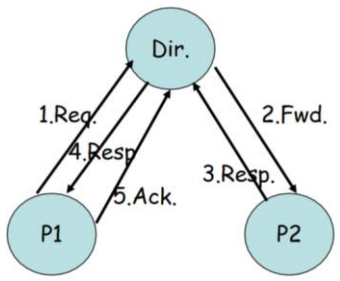
```
typedef struct directory_entry {
    bool present[MAX_NODES];
    int state;
    pthread_mutex_t lock;
    int locked_for;
    int intended_state;
    bool is_registered;
    char data[4096];
    string name;
    // TODO
} directory_entry_t;

directory_entry_t directory[MAX_PAGES];
```

The directory table is similar to what was shown in class. The only difference is that there is a “string name” member. This is used when shared memory is allocated in the heap using psu_dsm_malloc(). In which each node has their own name and corresponding virtual address and uses ‘name’ instead of ‘vaddr’ to communicate with the directory node.

In the handler() function, most of the DSM logic is handled. Whenever a RPC call is made, the method send_dsm_request is run. Depending on the parameters in the message ‘dsm_request’, the proper changes are made in the DSM table and a reply with proper permissions and data is sent using the message ‘dsm_reply’.

For reads we used the following message flow:



For writes we also get data from valid node before sending invalidate messages. The reason for this will be explained below.

In the handler() function we find the corresponding node id using the exception address. Then we check the state of this node. If the state is READ_ONLY then we know that the exception was caused by writing and therefore a WRITE_REQ is sent to the DSM directory node. If the state is INVALID then the exception can be caused by both reading or writing. Using the context pointer, we can check if the exception was caused by a read or write. If the exception is caused by a read in the INVALID state, then a READ_REQ is sent to the server. If the exception was caused by a write, then we first send a READ_REQ so that the node gets the proper data, then we send a WRITE_REQ to invalidate all other nodes. Making two sequential calls in this state works most of the time, except when two nodes enter the handler at the same time for the same page. If such a case occurs then after one node finished their READ_REQ and before calling WRITE_REQ the other node can send a READ_REQ. If this happens, when one of these nodes enters WRITE_REQ, the other node inside the handler doesn't get invalidated properly, since the handler also gives temporary read/write permission to pages to modify them. Therefore, one page keeps the old data without updating. To fix this problem we also returned the latest page with the reply for WRITE_REQ.

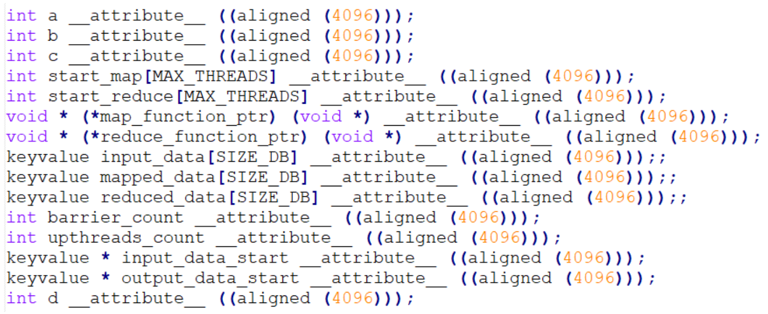
The function cleanup() is called whenever a nodes exits the program. Here an UNREGISTER is sent to the directory node along with the corresponding page data and virtual address/name. If this node is the only node with the most recent page data, then the directory node stores this page itself.

The functions psu_dsm_register_datasegment() and psu_dsm_malloc() are similar to each other. Depending on the size parameter they send either REGISTER or HEAP_REG messages to the directory node for each page of size 4096 bytes. If the size isn't divisible by 4096 then it is rounded up to a page.

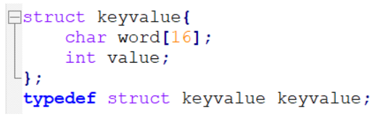
In psu_dsm_malloc() since we allocate space dynamically, the virtual addresses aren't the same across nodes. Therefore, we use a 'name' parameter to keep track of shared pages. Each node has its own vaddr_to_entry and entry_to_vaddr map for each name to keep track of their virtual addresses. If a size larger then a single page is allocated, then for each page, after the first page, a (i) is added after the name, where “i” is the page number. For example if we allocate 4096*3 for “test”, then we have three pages with names “test”, ”test(1)” and ”test(2)” sent to the directory node.

Map Reduce:

Map reduce uses both psu_lock.h and psu_dsm_system.h. Our wordcount program implementation is in wc.cc For our implementation we used several global shared variables shown below.



```
int a __attribute__ ((aligned (4096)));
int b __attribute__ ((aligned (4096)));
int c __attribute__ ((aligned (4096)));
int start_map[MAX_THREADS] __attribute__ ((aligned (4096)));
int start_reduce[MAX_THREADS] __attribute__ ((aligned (4096)));
void * (*map_function_ptr) (void *) __attribute__ ((aligned (4096)));
void * (*reduce_function_ptr) (void *) __attribute__ ((aligned (4096)));
keyvalue input_data[SIZE_DB] __attribute__ ((aligned (4096)));;
keyvalue mapped_data[SIZE_DB] __attribute__ ((aligned (4096)));;
keyvalue reduced_data[SIZE_DB] __attribute__ ((aligned (4096)));;
int barrier_count __attribute__ ((aligned (4096)));
int upthreads_count __attribute__ ((aligned (4096)));
keyvalue * input_data_start __attribute__ ((aligned (4096)));
keyvalue * output_data_start __attribute__ ((aligned (4096)));
int d __attribute__ ((aligned (4096)));
```



```
struct keyvalue{
    char word[16];
    int value;
};
typedef struct keyvalue keyvalue;
```

The a,b and c variables are used for synchronization across nodes. The start_map and start_reduce are used to synchronize mappers and reducers. The pointers map_function_ptr and reduce_function_ptr are used to point to which functions mappers and reducers run. The arrays input_data, mapped_data and reduced_data are arrays where the input, result of mappers and result of reducers are stored. The values barrier_count and upthreads_count are also used for synchronizing mappers and reducers. The pointers input_data_start and output_data_start are used in mappers and reducers to determine where data is being read from and where to store data.

The structure is similar to the word count program given that uses pthreads. In our implementation, there is a main function that calls the master node function (mr_master()) or the worker nodes function (mr_reducer) depending on the IP address and their location in nodes.txt. The IP at position NUM_THREADS becomes the master node. The master node waits for all worker nodes to start, then it initializes the global variables, stores the input data in input_data[] and called the functions psu_mr_map() and psu_mr_reduce(). The psu_mr_map() and psu_mr_reduce() functions synchronize the mappers and reducers using some of the global values described above. These also update the map_function_ptr and reduce_function_ptr values.

The worker nodes wait until the master node has started. Then when the signal for mapping arrives (by checking start_map[worker_node_id]) the function at map_function_ptr is run. When the mapper is finished the value of the shared variable ‘barrier_count’ is increased atomically to signal the master node. The same thing is done for the reducers, except the function that is run is reduce_function_ptr.

The mapper and reducer functions are called toy_mc_mapper() and better_wc_reducer().

K-means

The kmeans implementation is very similar to the word count implementation (explained above). One difference is that the keyvalue structure stores coordinates as the value instead of a char array. Another difference is that a global array for the number of clusters and number of data points is stored is stored in num_clusters and num_data_points respectively.

One important thing to keep in mind about our implementation is that the number of mappers/reducers are equal to the number of clusters, determined in the input file. It is possible for the number of mappers/reducers to be different from the number of clusters, but keeping them equal was the simplest way to implement kmeans as a map reduce algorithm. Therefore the nodes.txt file should contain at least num_clusters + 2 IP addresses (+1 for the master node and +1 for the directory node). Also the last IP in nodes.txt is the directory node, which is initialized by running "./dir" The second last IP in nodes.txt is the master node. **Note that there shouldn't be any empty lines in the nodes.txt file.**

RPC Logging:

All RPC messages sent (but not messages received) are stored in rpc-log-file*.txt , where * is the node id (index in nodes.txt) of the message sender.

Difficulties Faced

One problem we faced in the DSM implementation was determining if a read or write caused a page fault. At first, we assumed that all page faults for nodes in an invalid state were a result of a read, however this didn’t work when an invalid node tried to write before reading. We then found a way to determine if the exception was caused by a read or write by using the context pointer parameter. For the case of invalid write, we first sent a read request then a write request. This seemed to work without problems, but during our map reduce implementation we found out that this causes pages to not invalidate correctly if two nodes in an invalid state tried to write to the same page at the same time (more details are given in the DSM implementation part). We fixed this problem by making write requests send the most updated data to the requester.

Another problem we faced in the psu_dsm_malloc() function was how to keep track of different dynamically allocated pages. The DSM table normally uses virtual addresses to keep track of pages, but since dynamically allocated pages have different virtual addresses, we could not use virtual addresses as an identifier. We decided to add another column to the directory table that holds names in order to keep track of these pages. Either a page has a non-zero virtual address or it has a name. We added virtual address-to-name and name-to-virtual address global mappings to each node. So, when a page fault occurs, a node can use these maps to figure out what type of page the exception occurred in.

In map reduce, one problem we faced was how to synchronize mappers and reducers. We decided to use a globally shared array where each index determines whether a mapper/reducer should continue. We updated this array atomically using the distributed locking mechanism that we implemented.

Special Cases

We have made some hard coded limits in our program. These are:

For psu_lock.h:

- A maximum of 10 locks

For psu_dsm_system.h:

- A maximum of 100 pages
- A maximum of 10 nodes

For map reduce and kmeans:

- A maximum of 10 mappers/reducers
- A maximum shared database of 100 elements (410 elements for kmeans)

For the kmeans implementation the number of mappers/reducers is equal to the number of clusters. So the nodes.txt file should have enough IP addresses.

We also assume that the directory (dir.cc) will be run on the directory node (last ip in nodes.txt) before running any programs, that there will be a nodes.txt file and that these programs will be run on the W135 Lab machines.

While these limits and assumptions are met, our program should work for all cases. These limits can also be changed in the program before compiling, if they are insufficient.

Note: When compiling app1.cc in dist_sorting, there were two compilation errors. One error was that the global array partition_num and the function partition_num had the same name. To fix this we changed the function's name. The second problem was that in the initialize function, there is an undefined parameter &t. We replaced this parameter with null, since it's just used for generating a random seed.

Extra Work

In our DSM implementation, the directory node saves pages when a node with the only updated copy is exiting. This is done in the cleanup() function where the exiting node sends an UNREGISTER message to the directory node.

Our DSM supports multiple page allocations for both psu_dsm_register_datasegment() and psu_dsm_malloc() functions. How psu_dsm_malloc() does this was explained above under implementation.

Our kmeans implementation (kmeans.cc) works with any number of worker nodes equal to the number of clusters, which is read from the input file. So the amount of worker nodes aren’t hardcoded. (Note that there is an upper bound on the amount of workers, defined as MAX_THREADS = 10 this can be increased before compilation)

We also made two test files for psu_dsm_malloc() since we weren’t provided any.

1) app1malloc.cc is the first test case. This is the same implementation of dist_sort (app1.cc) which was given to us, but this test case uses psu_dsm_malloc() to allocate shared variables.
2) heap_test.cc is other test case. This allows allocating, reading and writing an arbitrary number of heaps of any size among any number of nodes. This program dynamically allocates character arrays of any size. Below is a brief explanation of how to use this program:
    a) First run heap_test on each node, and ./dir on the directory node.
    b) type: “malloc <name of variable> <size allocated>” to allocate memory. You can also type enter instead of spaces. Note that allocations larger than 1 page (4096 bytes) are supported
    c) type: “set <name of variable> <a string to set variable as>” to write to a variable. Writes can be larger than 1 page. (Note that in the W135 lab machines a single command can be at most 4096 characters. So, in order to allocate more than 1 page, you can modify the code to double the input string received as input or somehow increase this limit)
    d) type: “print” to print all shared variables initialized in this machine.
    e) type: “var <name of variable> <index>” to print a single character at the specified index.
    f) type: “0” to exit the program. Note that using ctrl+C to exit doesn’t allow the exiting node to send an unregister message to the manager node.

Work Distribution

Mahmudur Hera: Distributed locking, DSM, map reduce, kmeans

A. Burak Gulhan: DSM, kmeans, report