OS Project 1b Reports

Jin Han
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78250
Nov 30, 2015

Abstract
In the OS Project 1b, a dynamic library (libmythread.so) is built and it is used to isolate regular thread. Threads are implemented using separate processes with private and shared views of memory. We use this isolation to control the visibility of updates to shared memory, so each "thread" operates independently.

1. Introduction

Libmythread.so is finished by C++ language being compatible with C program. Original pthread_create function was replaced and thread_function would be executed in a child process which was created by clone system call. Child process id was recorded as original thread id for pthread_join function later. Malloc and realloc function was replaced to make heap area become sharable between processes. All synchronization initialize functions were replaced to make they would be shared by processes too.

2. Implementation

The whole implementation was finished by three steps. First, we would initialize global variable and heap area to make them become sharable between processes. Second, we replaced original thread and heap function to adapt with our goal. Finally, we handle the synchronization.

2.1. Initialize global variable and heap area

Before main function started, we need to finish all initialize steps. Otherwise, the result wont be accurate.
We will handle global variable area first. The initialization function is in hglobal class as global_initialize(). Since we could get the data start address by setting __data_start variable and end address by _end, we could locate the global variable area and calculate the size by these information.

Then we would use mmap function to locate a random memory block to store these initialized or uninitialized variables with their original value. After that, we would use mmap function to map the global variable area with a backup file.
The reason we are doing this is because clone system call would give child process another physical storage room. With mmap shared function, we would stop this physical storage mapping and all child process would be able to locate those variables in our mapped memory block.
In our example(test.c), we set global variable i, j. When their value were changed by process-as-thread, the change was captured successfully by other thread. In the end, the value was confirmed by the last visited thread. Thus, the initialization for global variable area is successful.

Next step, we would handle the heap area. The initialize function is in hglobal.h as heap_initialize(). At first, we would initialize a big user heap area by mmap function. The size was set by xdefines and could be changed according to current circumstances. We would use a global pointer to guide our replaced malloc function to obtain the memory block in our heap area. Since the whole heap area was shared between all processes, we could guarantee the change to a global area would be reflected correctly. Also, we need to maintain the heap list to record the heap id and corresponding size for future realloc and free.
In our example, if we use malloc function to obtain a memory block and link it to a shared variable, all thread would be able to operate the shared variable directly or through the address. According to our test result, the initialization for heap area is successful too.

2.2. Replace original function
Several original function were replaced and we will discuss them one by one.
First, we dealt with the pthread_create function. This is the start for everything. We used clone system call inside to make child function actually run in a child process. Also, we recorded the child process id as the thread id for future use.
Next, we replaced the pthread_join function. Here, we actually used waipid function to wait for the corresponding child process. Since we recorded all thread id and corresponding child process id in the creation step, it is easy to locate the child process we need to wait.

Next, we replaced the malloc and realloc function for heap. We want to guarantee the heap area we used would be available for all child process. We used a global pointer to make sure each memory block created by malloc function is in our initialized global heap area. After the assignment for each malloc function, the pointer would move forward corresponding size to next available address.
For realloc, actually we changed the function a little bit. We always assign new memory block for the realloc function. Since we recorded the heap object id and corresponding size, we would compare the old size and the new input size to get the minimum one. Then we use memcpy function to copy the right size content to the new memory block.

2.3 Handle synchronization
Fortunately, pthreads library could support the synchronizations among multiple processes now. That makes our job quite easy here.
We could use original pthread function to generate mutex, condition var objects. The only step we need to worry about is to set them to be shared by PTHREAD_PROCESS_SHARED attribute.
Thus, we replace all the original init function. If the original initialization was input with NULL attribute, we would generate a local shared attribute. Then we use this shared attribute to initialize the original object.

3. Test and Performance evaluation
We start our test with few simple programs.
In our test program one, we tested all those replaced functions to make sure they were executed correctly. Thus, we embed several printf function to track the executing steps.
As we could see, after the initialization of global variable and heap area, the original values were restored successfully for thread using.
After those thread changed the original value, those changes were captured correctly and the main thread print out the right value after waiting those executed child process.
Malloc and realloc function were performed correctly too and content was passed successfully after realloc function executed.

```
Malloc function is successful.
redefined malloc started.
Malloc function is successful.
Malloc random string: nwlrbbmqbh.
redefined realloc started.
oldsize : 10 new size : 7.
string after realloc: nwlrbbm.
Thread 36844 's child 36844.
Thread t1 : a address    0x602068.
Thread t1 : b address    0x60206c.
Thread t1 : 1 and 5.
clone() returned 36844
Thread 36845 's child 36845.
Thread 36844 is join with child.
child 36844 will join!
Thread 36845 is join with child.
child 36845 will join!
Thread t2 : a address    0x602068.
Thread t2 : b address    0x60206c.
Thread t2 : 1 and 3 .
Thread t2 Test : changed
clone() returned 36845
Main process : a address    0x602068.
Main process : b address    0x60206c.
Main process : a : 1, b: 3
Malloc string after pthread: changed.
redefined realloc started.
oldsize : 7 new size : 4.
string after realloc: chan.
dada@ubuntu:~/Desktop/test$
```

Next, we will compare our libmythread.so with original pthread library. We create a simple program which will create 16 thread and all of them would finish a 1000000000 times add function. As we could see from the result below

```
dada@ubuntu:~/Desktop/test$ ./compare
main start.
Finally, k= 103871069, m= 103847159, t= 207718228.
Thread processing cost total time: 5049938
```

With regular pthread library, the whole cost time is 5049938 microseconds. With our libmythread.so library, we finished whole calculation in 5438661 microseconds. Averagely, our libmythread.so library would increase about 10-50% overhead.

```
Thread 36965 is join with child.
child 36965 will join!
Finally, k= 107196645, m= 109736541, t= 216933186.
Thread processing cost total time: 5438661
```

Also, we executed our library with multithreadtests test. Most of them could be done with our library.

4. Problems and Discussion

Basically, we just started a huge framework and there are lots of jobs need to be done in future.

About the heap area, right now we set up the whole user heap area as shared, but that could be done more precisely to avoid data corruption and make the whole system safer.

Besides that, there are still few benchmark test failed with our library, we should check back our code more carefully to extend it availability.

5. Conclusion

So far, most of requirement of project 1b is full filled and whole project is completed successfully. Thread and process function are practiced and we got a deep understanding inside them. The whole project acts as a wonderful guidance to be familiar with the thread and processes. We kind just started a big framework and lots for work should be done in future.