Asst1-416
OS Assignment 1 (My PThread and Scheduler)

AUTHORS: Bruno Lucarelli
Alex Marek
Joseph Gormley

LIBRARY: my_pthread_t.h

iLab machine: man.cs.rutgers.edu

my_pthread_create()
This function essentially creates a new ‘context’ in the calling process. This gives the illusion of code running concurrently. During the first call to my_pthread_create() the scheduler will be initialized to store the recently created thread in the MLPQ data structure. Any following threads created will be stored in the same manner.

my_pthread_yield()
The calling thread will surrender it’s quanta and wait for the scheduler’s cycle to be placed in MLPQ’s corresponding priority queue. To keep the scheduler informed, the thread’s status is set to THREAD_YIELDED() in its control block.

my_pthread_exit()
This is a termination of the calling thread. If the calling thread was joined previously, my_pthread_exit() will pass a value to the waiting thread’s control block’s valuePtr member (which is a void**, allowing us to dereference the user’s input in join() and set its value to the value passed into exit(). In the case this is called implicitly, the pointer is never passed and the context is switched back to the scheduler.

my_pthread_join()
The function calling join is set to the wait status of THREAD_WAIT, only once the target thread calls my_pthread_exit() can the joined thread finish running its context. In addition, the waiting thread sets the target thread’s waitingThread from the default id of NUM_PRIORITY_LEVELS + 2 to the waiting thread’s id. This allows my_pthread_exit() to reference the waiting threads passed pointer and edit it if explicitly called.

my_pthread_mutex_init()
Called explicitly by the user, my_pthread_mutex_init will take the parameter mutex and initialize the my_pthread_mutex_t struct values: status, waitQueue, ownerID, and attr. Status is set as UNLOCKED, waitQueue would be a list of threads waiting for this mutex to unlock therefore it is initialized NULL, ownerID is the ID of the thread that is using the mutex and is initially set as the manager thread.

my_pthread_mutex_lock()
This function will check if the requested mutex is currently LOCKED by another thread. If it is, then the calling thread will be added to the waitQueue and will signal BLOCKED until another thread unlocks this mutex. When the mutex is available, the calling thread will successfully acquire and lock the mutex.

my_pthread_mutex_unlock()
This function will set the status of the mutex parameter to status UNLOCKED and will alert the next thread waiting to acquire this mutex. Mutex returns 0 on success, -1 on failure. This function fails if a mutex is not initialized or is not owned by the calling thread.

my_pthread_mutex_destroy()
This function will clean the current state of the mutex unless it is uninitialized or if it is currently LOCKED by another thread. Returns 0 on success, -1 on failure.

==========

0. DESIGN / DATA STRUCTURES

==========

‘Pnodes’

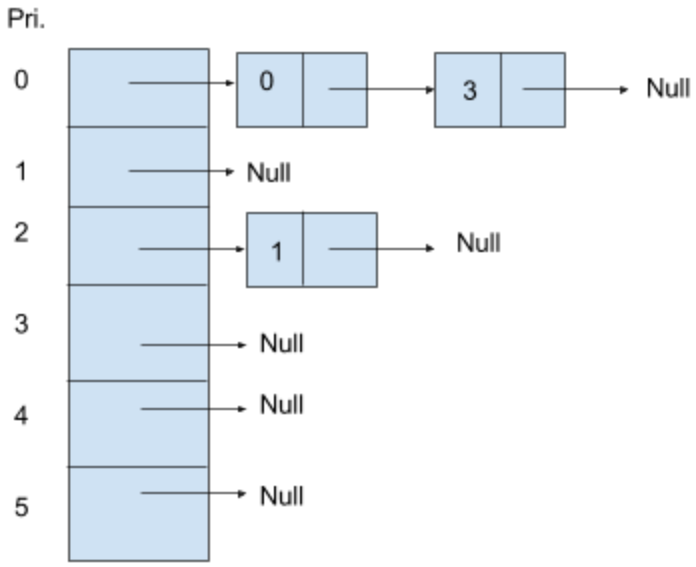
The pnode is the basic struct used as a unit for storing information about a thread in a queue, such as the **runQueue**, **MLPQ**, and a mutex’s **waitQueue**. The pnode only has two members: a my_pthread_t member called “tid”, and a pointer to another pnode called “next”. A pnode only stores the thread ID (TID) of its corresponding thread and the next node, because we can use the TID to access the global tcbList array and gain further attributes. Because we only store the TID and a pointer to the next node (the bare minimum information) and relegate any other information to a globally-accessible array of tcb’s called **tcbList**, redundancy is minimized and heap-data management is much cleaner.

From here on out, it’s important to keep in mind that a pnode’s presence in any given queue indicates something about the thread’s status to the scheduler. If the pnode is in the MLPQ, then the thread sharing its TID is treated as though being in the MLPQ; if the pnode is in the runQueue, the thread itself is treated as though it is in the runtime queue. Etc. More details on this in upcoming sections.

MLPQ

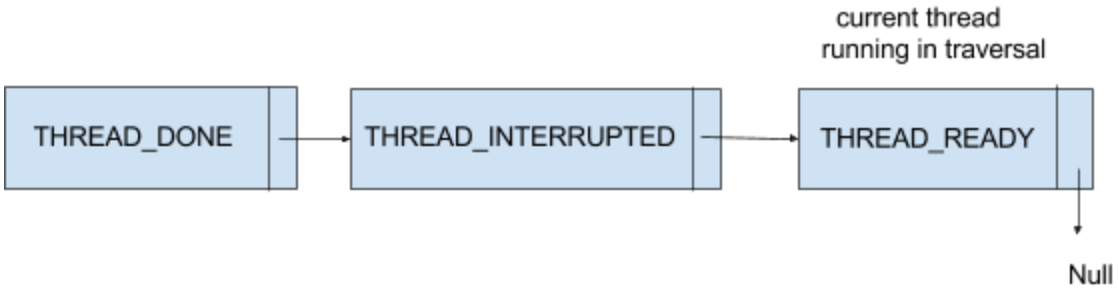
The Multi-Level Priority Queue (MLPQ for short) is a structure we use to store threads at multiple levels of “urgency”. It is an array of pnodes with the number of total cells equal to the number of priority levels. Each cell stores a pnode, which itself can point to other pnodes- thus letting the MLPQ store multiple linked lists. Any pnode in the MLPQ is not present in the runQueue, and vice-versa, as they populate from one another during the maintenance cycle; meaning that there is no redundant storage between the two. In addition, the maintenance cycle can use the TID stored in each pnode in the MLPQ to perform operations such as changing statuses, freeing a thread control block, and removing the pnode from the MLPQ to populate the runQueue.

As discussed in class, the “highest” level is Priority 0, where threads are allocated the fewest number of time slices, and the “lowest” level is Priority X, where threads are allocated the greatest number of time slices. Let X equal the number of total priority levels possible- our implementation is not hard-coded to have a certain maximum number of threads or a certain number of priority levels, and uses easily-modifiable tokens at the top of the my_pthread.c file to determine these files (MAX_NUM_THREADS and NUM_PRIORITY_LEVELS, respectively). We chose to make our maximum number of threads 64 and our number of priority levels 5, by default, due to the method in which we allocate time slices. More about this in the scheduler.



'Runtime Queue'

The runtime queue (or runQueue, colloquially) is itself a pointer to a pnode. When populated by the scheduler's maintenance cycle, it will point to one pnode, which may itself be the head of a linked list of pnodes. The runQueue is used by the scheduler to keep track of which threads need to run and which order they should be run in, along with keeping track of the current state of the run stage.

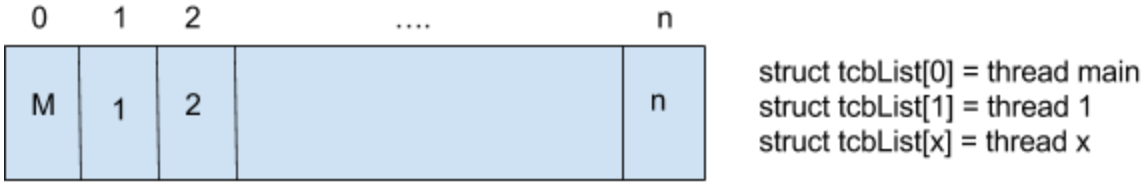


Thread Control Block

As given 'jobs' are assigned to different thread, details about each thread need to be stored in such a manner to be easily referenced with minimal redundancy. These details include such things as a status, stack pointer, thread id, current priority, slices of time allocated to it, the pointer to the value returned by the thread it has joined (if applicable), etc. We call a Thread Control Block a "tcb" colloquially.

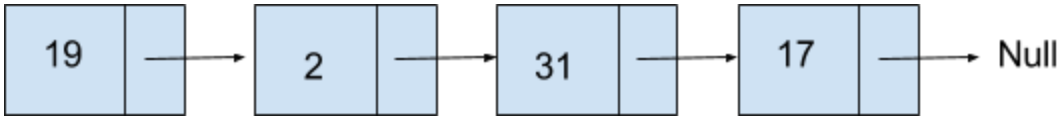
We chose to make tcb's globally accessible in the library by storing a global array of tcb's called tcbList. tcbList has a total number of cells equal to the maximum number of threads allotted to the user… with one reserved for their main thread. The current token in our library, MAX_NUM_THREADS, is set to 64; meaning that a user can create 63 of their own threads while one thread, with TID 0 and cell tcbList[0], is reserved for their main function. Our design

allows this token to be changed flexibly, and for the purposes of well-rounded testing, we've set it to 64 by default. More about this number in the scheduler.



Thread Recycle

Thread IDs are distributed all the way until the highest tid is assigned, once this occurs, we recycle previously assigned thread ids providing the respective thread has finished its task. We assure this happens by inserting such tids into a linked list as the thread's job has been completed. In addition, a thread's cell in tcbList is set to NULL when it has finished (and the corresponding tcb is freed); so a new tcb can be added to that spot and the thread is effectively "recycled". All of these operations besides actually adding a finished TID to the recycledQueue are done in constant time- a TID to be recycled is picked from the front of the list, an array is accessed instantly, etc. The addition of the finished TID takes O(n) time for a maximum of n possible recycled TID's.



My_pthread_mutex_t (or mutex, for short)

The my_pthread_mutex_t struct contains all necessary information for a mutex in our system- a mutex status (LOCKED or UNLOCKED), a queue of pnodes indicating the order of threads currently waiting on this mutex, the TID of the owner of the lock, and a member "attr" storing the mutex attributes passed in by the user and required for POSIX implementation.

Mutexes are a unique part of our system as they are the only entity other than the scheduler that can change thread attributes. A thread that has released its mutex will set the first thread waiting on the lock to have a status of "THREAD_READY", and a thread that is waiting on a mutex will have its status changed to "THREAD_BLOCKED". The design of mutexes thus takes advantage of the scheduler's implementation to maintain consistency and minimize redundancy.

==========

1. SCHEDULER DETAILS

==========

The scheduler works, generally speaking, in the exact same fashion as described in class for a MLPQ implementation. It is initialized when the user calls my_pthread_create() for the first time, at which point a thread is created for their calling code and then another “first user thread” is created for them. The scheduler runs the maintenance cycle once, getting back to the main function, at which point it can carry on until some form of interruption occurs (a join() or yield() call, a timeout preemption, etc). There are six essential thread statuses to keep in mind going forward, that will be referenced:

- THREAD_READY- indicates that a thread is ready to run.
- THREAD_DONE- indicates that a thread is finished running successfully.
- THREAD_INTERRUPTED- indicates that a thread was interrupted due to running out of time.
- THREAD_WAITING- indicates that a thread is joined into another thread.
- THREAD_BLOCKED- indicates that a thread is waiting on a lock held by another thread.
- THREAD_YIELDING- indicates that a thread will yield to all other threads in its current level of the MLPQ.

There are two essential parts to the scheduler:

1) runQueueHelper()
    a) This function is used by the scheduler to perform operations related to running threads in the runtime queue, or runQueue.
    b) Global variables current_thread and current_status store information obvious to their names- the former stores the TID of the current thread, and the latter stores the status of that thread. In addition, the status of whether the currently-running thread has manually called exit is stored in a variable called current_exited, in order to determine whether to perform an implicit exit on a thread that did not call pthread_exit() before finishing execution. The runQueue does not use these as much as auxiliary functions use the current thread ID and its status to make determinations about their operations on tcb’s.
    c) Notably, a thread that runs over its allocated time will be interrupted by SIGVTALRMhandler(). This signal handler is set at the beginning of each runQueueHelper() call, as an itimer is set directly before each thread is run. This function sets the thread’s status to THREAD_INTERRUPTED and swaps contexts with the Manager, allowing it to resume running threads while the scheduler “remembers” that the thread had been interrupted.
    d) runQueueHelper() does not itself free or allocate space. It also does not set thread statuses in the tcb. It does, however, facilitate the context swaps and preemptions that may occur and change those statuses. maintenanceHelper() actually makes use of the state changes that occur during thread execution.

2) maintenanceHelper()
    a) This function is the “meat” of the scheduler as far as organization of threads, priority allocation, time slice allocation, deallocation, and population of the runQueue goes. It has three main parts:
        i) Part 1: runQueue clearing and housekeeping.
            (1) The runQueue is cleared, and threads are either inserted back into the MLPQ at their current priority (if the thread is blocked, waiting, or yielding), deallocated (if the thread finished running successfully with a status of THREAD_DONE), or inserts the thread back into the MLPQ at a lower priority level while decreasing its priority (if the thread has status THREAD_INTERRUPTED). In the case that a thread is yielding or was interrupted, its status is now set to THREAD_READY. Note that a thread that has finished its join() call is set to THREAD_YIELDED, so that it can hopefully run in a near future cycle.
            (2) Pnodes are moved from the runQueue to the MLPQ, saving malloc() and free() calls and thus minimizing issues with memory leaks.
            (3) This cycle ultimately acts to set the state of the MLPQ after each run cycle. This is the true “maintenance” portion of the cycle, while Part 2 acts to support runtime.
        ii) Part 2: runQueue population and timeSlice allocation.
            (1) The runQueue is populated only with threads that are marked as THREAD_READY. This can include threads that were interrupted or yielded last cycle, or which were unblocked/which finished their join() calls. The runQueue is populated in descending order of priority, until timeSlices run out. By convention discussed in lecture, we allow 20 time slices at most (25ms per quanta times 20 is half a second per run cycle).
            (2) Time slices are allocated to threads depending on their current priority level. The function used is 2^(priority), meaning that a thread at P0 will receive 1 time slice, at P1 will receive 2, at P2 will receive 4, at P3 will receive 8, and at P4 will receive 16. By the assignment’s convention, each time slice is worth 25ms. We set our MAX_NUM_THREADS token to 64 and our NUM_PRIORITY_LEVELS token to 5 so that, in the worst case, we can have one bottom-priority thread that gets 16*25ms = 400ms per cycle! This is more than adequate even for threads that perform multiple file I/O accesses, and there are still 4 time slices that can be used by easier, higher-priority threads.
                (a) An example of an allocation might be 4 P0 threads and 8 P1 threads, 12 P0 threads and 1 P3 thread, etc.

iii) Part 3: Check if runQueue and MLPQ are both empty. If so, we set a global manager_active key to 0. During the next maintenanceHelper() call, the MLPQ and tcbList pointers will be deallocated.

3) My_pthread_manager calls maintenanceHelper() and then runQueueHelper() each cycle. However, I listed them in the reverse order above so that you can better-understand how the maintenanceHelper() draws from the results of runQueueHelper. :-)

The auxiliary functions createTcb(), createPnode(), and init_manager_thread serve to simply perform all of the functions they describe. But the design of our tcb's and the implementation of our core functions all work with a design where the Scheduler's maintenance cycle intelligently handles threads depending on their statuses and priority levels and does most of the heavy-duty work.

It should also be mentioned, again, that the user's own Main code which makes the first my_pthread_create() call will be treated as thread #0 by the scheduler. So if you want to max out the number of threads our code runs, do MAX_NUM_THREADS - 1 (if you set the token to 100 in our library file, use a parameter of 99 when executing). Our tests, which will be described ahead, used NUM_PRIORITY_LEVELS of 5 and MAX_NUM_THREADS of 64, meaning that we ran all of the provided benchmarks using a parameter of 63.

==========
2. TESTING RESULTS
==========

We tested our library using four different test files. The first of them is our own creation, and the other three are the benchmarking files generously provided by one of our TA's. After looking at what each of them did, we felt they were sufficient to test our and strain our functionality, as they each focused on different aspects of our library's capabilities. All of the benchmarking functions provided to us (not the one we wrote) are run with 63 threads, giving a total of 64 threads running in tandem if we include the Main thread #0. We would like the grader to keep in mind that while we kept speed/space efficiency in mind during design, we primarily designed this library for robustness and flexibility (as we're not expert NPTL developers). :-0

For each testing file/benchmark, we set a clock at the very beginning of the main() function, and get the time/print it out right before it returns. We do this because this allows us to obtain as much information as possible about the functioning of all "child threads" including the main() function. We used a boilerplate example from Rutgers CS416 for this specific purpose: https://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/gettime.html

Do note for that for all of the included benchmarking files and our own test file, the calculated sum/result matches the correct/verified one. Meaning that our implementation "works" and is functionally-correct as far as we can tell. So these are figures for the correct scenarios. :-) We also check to make sure the correct library is being used, as our implementation will print out "using my_pthread implementation" upon initialization.

Basicpthreads.c

Overview: This is a basic test we put together to make sure that my_pthread_join() and my_pthread_exit() are able to work as intended in the case that the user decides to pass a value_ptr into join(). We pass a pnode pointer into join(), and the thread calling exit() sets that pointer to point to an allocated pnode struct containing TID 9001 and and linking to NULL as the next node. It does not test speed, but rather the success of these (constant-time) operations.

Please note that for some reason, trying to make this original test file and compile it causes re-declaration errors in the header file unless I add/remove <pthreads.h> library include statements from either the header or the test file. We've included basicpthreads.c in our submission file, but due to time constraints cannot figure out how to get to compile consistently with the default vs. our implementations without adding/removing some include statements. This should not have any effect on performance, however.

Parameters: N/A, only runs with one thread.

Our library results (in nanoseconds):
Trial 1: 150563
Trial 2: 74060
Trial 3: 97140
Trial 4: 98157
Trial 5: 103591
Average: 104702.2

Default pthreads results (in nanoseconds):
Trial 1: 214468
Trial 2: 221884
Trial 3: 208170
Trial 4: 474438
Trial 5: 244667
Average: 272725.4

Our library vs. default library: avg_our/avg_default * 100 = 38.91%.

Our library took ~38.9% the time that the default library took, to perform a one-thread operation and fetch a pointer sent into the child thread. :-)

vectorMultiply.c

Overview: This TA-provided testing benchmark file has a specified number of threads performing operations on a vector of a modifiable size, in order to calculate an overall result. Many threads make use of the same multiplication function.

Parameters: 63 (for 63 child threads performing calculations and the main thread)

Our library results (in nanoseconds):
Trial 1: 232138
Trial 2: 234288
Trial 3: 228208
Trial 4: 243056
Trial 5: 239542
Average: 235446.4

Default pthreads results (in nanoseconds):
Trial 1: 806171
Trial 2: 878494
Trial 3: 780719
Trial 4: 847116
Trial 5: 842011
Average: 830902.2

Our library vs. default library: avg_our/avg_default * 100 = 28.34%

Our library took ~28.3% the time that the default library took, to perform vector operations in a multithreaded environment. :-)

externalCal.c

Overview: This TA-provided testing benchmark file performs read/write operations in the process of using a specified number of threads to access that file and contribute to an overall sum. It puts strains on the user's implementation as far as handling I/O operations goes, as these operations are less likely to be fulfilled in the lower limits of time quanta allocated. Many threads make use of the same I/O accessing calculation function.

Parameters: 63 (for 63 child threads performing calculations and the main thread)

Our library results (in nanoseconds):
Trial 1: 11421076
Trial 2: 7092970
Trial 3: 7177633
Trial 4: 8635131
Trial 5: 10613478
Average: 8988057.6

Default pthreads results (in nanoseconds):
Trial 1: 9090465
Trial 2: 5837006
Trial 3: 7448302
Trial 4: 6029065
Trial 5: 6306978
Average: 6942363.2

Our library vs. default library: avg_our/avg_default * 100 = 129.5%.

Our library took ~129.5% the time that the default library took, to perform a series of multithreaded operations paired with lots of I/O operations.

parallelCal.c

Overview: This TA-provided testing benchmark file performs a similar function to externalCal, except for the results being different and for a lack of access to I/O. This puts strain on the user's implementation as far as handling quicker, but not instant, operations go, and interrupts are almost certain to occur in most implementations. In addition, it puts strain on the user's implementation in causing far more context switches and maintenance cycles to occur, looking at how the user's overhead for context-switching and maintenance cycles scale up to more extreme scenario.

Parameters: 63 (for 63 child threads performing calculations and the main thread)

Our library results (in nanoseconds):
Trial 1: 722153326
Trial 2: 722739499
Trial 3: 722549853
Trial 4: 718654345
Trial 5: 719821736
Average: 721183752

Default pthreads results (in nanoseconds):
Trial 1: 567642019
Trial 2: 572890856
Trial 3: 573014108
Trial 4: 580041442
Trial 5: 568566701
Average: 572431025

Our library vs. default library: avg_our/avg_default * 100 = 126.0%.

Our library took ~126% the time that the default library took, to perform very intensive multithreaded calculations with many threads running and likely being interrupted.

Final Thoughts on the Project

We're quite happy with the robustness of our pthreads implementation, but are also very pleasantly surprised to see that the performance is only slightly worse than the default library's, and in some cases, significantly better! There may be some edge cases our testing didn't address out of our forgetfulness, but considering that our library can be used to handle 63-thread operations while correctly calculating results, this is a very pleasing outcome. Nothing is perfect, but we hope that helpful feedback from the TA's can provide us with insight that we can use to improve our system for future projects.

You can reach group member Bruno Lucarelli (bjl145) at brunojameslucarelli@gmail.com with any questions, comments, or inquiries. For any questions about our pacing, design problems, and originality, we would be glad to provide you with our Git commit history. :-) Have a great day!