Team: Benjamin De Brasi, Pranav Katkamwar, Swapneel Chalageri
Computer tested: Strategy

The goal of this project was to build a virtual memory library which would manage all malloc and free calls with stack allocated memalign()ed memory to emulate paged memory and the layer of abstraction that the OS provides between hardware memory and virtual memory. Our virtual memory implementation depends on our scheduler so the scheduler will be explained first followed by virtual memory.

Our scheduler uses a multilevel priority queue consisting of 3 levels: low, normal and high corresponding to threads which have the lowest priority to the highest. Each level has it's own quanta for any thread in each corresponding to 50000, 100000 and 150000 microseconds respectively. Each queue takes in my_pthread_node structs which serve as nodes for our scheduler multilevel priority queue. They contain fields necessary for the scheduling queue and for the user such as the total time that thread has been running, an utcx struct for swapping and making contexts, a retval for pthread join and exit, a flag for detecting priority inversion and more. These nodes are also what populate other queue structs contained in our scheduler struct such as one we have reserved for storing dead threads for use in pthread join and exit and a waiting queue for use with mutex locks. Our scheduler also has a pointer to the main context, a pointer to the running context and a count for the total amount of threads currently being scheduled. Finally, complimentary to the pthread nodes are the pthread_t struct that the user can declare which contains a pointer to it's corresponding node, its id and its retval. The last methods for the scheduler are related to the mutex locks implementations. Mutex_init, mutex_lock, mutex_unlock and mutex_destroy are the four functions involved with mutex locks.

The basic flow of these parts in our scheduler is as follows: The entry point to our scheduler can either happen through pthread_create or mutex_init depending on whether the user decides to initialize threads or mutex locks first. Any other function depends on the existence of either a mutex, a thread and, by extension, our scheduler and thus is invalid if one of those 2 functions were not called before. When either one of those functions are called the first piece of code checks an ints value which serves as a flag for the whether the scheduler has been initialized. If it hasn't then an init_scheduler() function is called which malloc()s all appropriate fields for the scheduler. With our implementation of the virtual memory all data here is stored in the OS region which consists of the first 4 megabytes of our 8 megabyte physical memory array with the exception of the first page which is used as a buffer page for swapping pages. Once the fields have been malloc()ed then the flag is flipped so that if either function gets called the scheduler will not be reinitialized. Afterward, either mutex_init or pthread_create will be continue it's functioning. In the case of mutex_init, a counter for the amount of locks will be incremented, the lock will be added to the to a linked list in the scheduler which stores all the locks. If pthread_create was called then all utcx structs are initialized and the levels are given to the node (always high upon creation), and the context is made. Afterward, there is another flag to check if this is first time pthread_create() was run. If so, the main context is set up which consists of setting the data for the main_context field in the scheduler, placed in the running queue and then returning so that context will immediately return to the user's main function when that thread node is set as the running context.

The next flow step after the threads and mutexes have been created is actually running the threads. The way this is done is by having all threads run through a specific function that allows our program to easily collect the retvals of each thread and place each thread to enter the dead thread queue once it's finished. The function of the dead thread queue is to allow us to see the retvals for pthread join even after the thread has been freed. This function takes in the function and the arguments the user wants to run in that thread as well. This function will continue to run until the timer signal handler gets run or until pthread yield, join and exit gets called. Each of these functions do a variation of a similar thing: deschedule a thread. The details, conditions and follow differ with each however.

If the signal is activated to reschedule a thread then the following chain of events occurs. 1. The scheduler counter is incremented. If the scheduler reaches a value defined in a macro for how often to call the maintenance cycle then it passes the if statement and the maintenance cycle is run. The maintenance cycle works by shuffling all the threads in the running queue between all the levels based on run and wait time thus sorting the order of all the threads. Then the maintenance counter is reset to 0 and the if statement ends. Then, the thread is checked for it's status. These are determined via an enum which correspond to things such as a wait, ready or dead. Depending on the current status the handler calls yield and reassigns the thread appropriately. Yield works by taking the running context and checking its status. If it's ready then we know we need to swap contexts, if not then we can set it since there is no context to save or put back into the scheduler. Before each of these there is a call to a mprotect_setter() which takes all the threads which belong to the previous running context and mprotect()s with PROT_NONE flag and mprotect() with PROT_READ | PROT_WRITE for the context that will be the next running context. After this the context is either set or swapped and the program resumes it's functionality from that thread. Almost all the same things happen if yield is called directly and if the signal is called with one exception. If the signal handler is called that means the thread ran for it's entire quanta so the former running context gets demoted and knocked down one level down in the multilevel queue. If yield is called and the timer has not been ended then the thread gets rescheduled to the same level of the multilevel queue.

The remaining pthread functions not covered are pthread_exit and pthread_join. Pthread_exit works by immediately taking the thread out of the running context and out of any other queues it is in and placing it into the dead thread queue. The retval is changed according to the retval the user gives. The reason the threads are stored in the dead queue is because it may be the case the user frees their thread, but we the retval is still needed for a join call and so we can free the scheduler node. By saving the tid and the retval in a queue then we can still check any threads exit status at any time if pthread_join is called. Pthread_join simply keeps whatever thread called it in a loop which calls pthread_yield until the retval of the thread that is supplied is marked ENDED. At that point the thread that was waiting can get rescheduled.

The last set of functions for the pthread library is the mutex_lock, mutex_unlock and mutex_destroy functions. Mutex_lock sets a flag that marks a thread as having a lock and a pointer in the thread struct is pointing to the specific lock it has. With that pointer we can easily assign all threads to the mutex locks waiting queue and also check when that lock has finally be released and then release the waiting threads for that lock and place them into the ready queue. In lock, we also have a check for priority inversion. The way we do this is by making sure that a node which holds a lock a node that is waiting on that lock are all scheduled with preference. That way waiting queues will not be tied down and can finish their functionality quickly. The mutex_unlock simply changes the flag for a lock and reschedules a thread. Mutex_destroy first checks if a mutex exists. If it does then it continues if not return NULL. If the mutex to be destroyed is currently locked then we return 1 for failure as freeing a locked and used mutex can cause deadlock. Finally, if these two conditions are met we can now free the lock.

All of the above described the scheduler part of our program and the parts of the virtual memory (hereafter VM) assignment which were directly related to the scheduler. The remaining is the virtual memory part of the program. Our implementation for VM is as follows. We have a memalign()ed array which we request 8 MB divided by 4096 bytes which represent the page size on the ilab machines. A global page table is used for bookkeeping. This is used to identify and traverse through pages when trying to locate them to swap or free.

The first time a user calls malloc() is initialize all memory that the process will have. The very first page in the 8 MB will be allocated as a buffer page. The buffer page will only for transferring the data of pages in case of a thread attempts to access data it should not have access to and to give the user the illusion of contiguous memory. After this the user is able to get multiple pages for a specific thread.

Phases A and B are complete, and the scheduler is fixed – all test cases on Sakai for Assignment 1 pass. The code was compiled and tested on the strategy.cs.rutgers.edu iLab machine.