Asst2: Scheduler + Memory
Daniel Schley (drs218)
Anthony Castronuovo (ajc320)
Thanassi Natsis (tmn61)
ilab machine used: grep.cs.rutgers.edu

Technical decisions (IMPORTANT):
Initially, when main called malloc before creating any threads, the data would still get put into pages in user space. However this lead to problems when main allocated data that needs to be referenced from other threads (i.e. my_pthread_t and arg passed to create) because the address lies in main's protected space, but when the thread is running and tries to access it, it seg faults because of the protection and swaps its own page into place. Rather than limiting the user to only being able to use locally created arguments to my_pthread_create(), any malloc calls in main that are made before the first my_pthread_create() is called is put into the library owned space of the memory block, this way threads have an unprotected reference to its args and my_pthread_t's. The tradeoff in this case means that if main does not create any threads, it is only limited to allocating within library space (roughly 200 pages worth of space) but once threads are created, all threads including main have full access to the user space of memory (roughly 1848 pages worth).

Additionally, there is a similar issue with exit and join. If a thread passes the pointer to malloc'ed data into my_pthread_exit(), the data belongs on the protected page and when it is passed to the thread that called join, it has no way to access this data. However everything works as intended when a local variable is passed through exit (i.e. int x = 5; my_pthread_exit(&x);).

malloc(size_t size)
This function automatically gets redefined to myallocate and if it is redefined this way, it indicates the the function that it was called by a thread. When a thread calls this, its memory gets allocated onto protected pages and can allocate up to 1848 pages worth of memory (200 pages are set aside for library saved data and other metadata).

free(void* address)
This function automatically gets redefined to mydeallocate which locates the memory on the page of the thread that called free.

Trimming the fat
Originally our page struct contained an int pageID (4 bytes), an int threadID (4 bytes), and a bool isFree (1 byte) for a total of 9 bytes. Now all of this information is stored in an unsigned int (4 bytes). pageID can range from 0 to 2000 so it needs 11 bits. isFree can be 0 or 1 so it needs 1 bit. Then we decided to give threadID 8 bits so you can make up to 256 threads which is more than what we can store or the user is likely to make. This totals 20 bits so the

smallest primitive that can hold this is an unsigned int. The first 2 bytes are used to store pageID, the 3rd byte stores threadID and the last byte stores isFree.

Our block struct contained a bool isFree (1 byte), an int size (4 bytes), and a struct Block* next (8 bytes) for a total of 13 bytes. Now all of this information is stored in an unsigned int (4 bytes). isFree can be 0 or 1 so it needs 1 bit. Size can be 0 to 8MB so it needs 23 bits. Block* next is not required, instead we used an isNext which is 0 or 1 so it needs 1 bit. To get to the next, you increment the address of the current location by its size if isNext is 1, otherwise isNext is 0 meaning it has no next so null is returned when you try to get the next block. This totals 25 bits so the smallest primitive that can hold this is an unsigned int. The first 3 bytes store size, the next 4 bits store isNext, and the last 4 bits store isFree.

Asst1: My_pthread project
Daniel Schley (drs218)
Anthony Castronuovo (ajc320)
Thanassi Natsis (tmn61)
ilab machine used: cd.cs.rutgers.edu

Pthreads:

**intmy_pthread_create(my_pthread_t * thread, my_pthread_attr_t * attr,
void *(*function)(void*), void * arg)**
This function takes a my_pthread_t pointer/address, an attr which is ignored, a void* function that takes a void* argument, and a void* argument. On the first call of this function, an initialization function is called that sets up the scheduler, its resources, saves the currently running context (main) which will be treated as another “thread”, and starts a timer for the scheduler to know when to swap contexts. After checking if this is the first call to pthread create, it makes a context for the given function to run, allocates its own stack, and enqueues it into the running queue. The function returns 0 on success and 1 on failure.

voidmy_pthread_yield()
This function is explicitly called by the thread that wishes to yield its current slice of time provided to it and enqueues it to the same priority level in the running queue.

voidmy_pthread_exit(void * value_ptr)
This function is explicitly called by the thread that wishes to exitand takes a void* argument which can be used to pass an argument to another thread that wishes to join on this thread.

intmy_pthread_join(my_pthread_t thread, void ** value_ptr)
This function is called by the thread that would like to wait until the thread specified by the first argument is done executing before proceeding to the rest of the code. The second argument is a void** that points to the void* argument provided by pthread exit (if it decides to provide one). Once join is called, it checks the list of threads to see if the thread it wishes to wait on exists, and if it is terminated or not. If it is already terminated it returns immediately, but if not, the calling thread is put to sleep and on a waiting list which is checked every time a thread terminates. Once the proper thread terminates, the joining thread gets taken off the waiting list and put back into the running queue. This function returns 0 on success and 1 on failure.

Mutexes:

my_pthread_mutex_init(my_pthread_mutex_t *mutex, constmy_pthread_mutexattr_t *mutexattr)

Init takes as arguments a pointer to a mutex created by the user to be initialized, and it takes in a my_pthread_mutexattr pointer. For the sake of the assignment, mutexattr will be ignored. This function initializes the mutex given by the user. A new my_pthread_mutex_t object is created which is mapped back to the mutex given in. The flag, isInit is set to 1, which says the mutex has been initialized. The new mutex is pushed into a list of my_pthread_mutex_t objects, which keeps track of which mutexes are currently initialized, and which ones are currently being used via the isLocked flag in the structure definition.

my_pthread_mutex_lock(my_pthread_mutex_t * mutex)
This function takes in the mutex to be locked. First we perform a check to see if the mutex has been properly initialized, and if it is currently in the list of mutexes being used. If not, the lock function will return with -1. If the mutex is in the list, and it has been properly initialized, it will go on to lock the given mutex. For this ,we used a spin lock by using an atomic test-and-set operation. (We were going to implement lock so that there is a waiting queue with sleeping threads waiting on the lock but the TA said it was much simpler to do spinlock and said we should do that instead for this project). Once the lock has been successful, the function will return 0, and and threads attempting to access the lock, will be waiting.

my_pthread_mutex_unlock(my_pthread_mutex_t *mutex)
This function takes in the locked mutex to be unlocked. Considering that nothing notable happens if isLocked is set to 0 from already being 0, we decided not to implement a check if the lock was already taken. In both this, and the lock functions, we utilize atomic functions, so that the isLocked flag is changed without being interrupted. We return 0 to show successful unlocking. We also return 0 if the mutex was already unlocked, seeing as nothing actually would have happened.

my_pthread_mutex_destroy(my_pthread_mutex_t *mutex)
Destroy takes in the mutex to be destroyed. First, we traverse the mutex list to see if the mutex has been initialized. If there is no mutex in the list, we return -1. If the mutex is in the list, first we set the isLocked to 0 to make sure any lock that was held by the mutex, is unlocked. Then we set all the attributes next to NULL, mutexID to -1, and isInit to 0. Finally we set the entire mutex to NULL as well as the mutex returned from the list.

Scheduler:

The scheduler struct itself has a pointer to the current context that is running and a multilevel (5) priority queue that runs whatever is next to run from the topmost level down. If there is nothing in the top level it checks the next and so on until it gets to the bottom level. Each level runs the threads for a time slice of 50ms * (priority level +1) (50ms at priority 0, … , 250ms at priority 4). After a thread runs for a full time slice it is demoted to the next level queue until it reaches the bottom level. In addition to the running queue there is also a list of threads that have been created in this process for bookkeeping on exits and joins. There is a join list for threads that

called join and are waiting on the thread they want to finish. There is also a number to keep track of the number of cycles the scheduler has gone through for scheduler maintenance purposes. For scheduler maintainence, every hundred time the scheduler has been run, a function is called from the scheduler to move up the priorities of the threads in the run queue. The lower priority queues update more threads and the threads selected to be updated are selected based on the amount of time since the threads creation. There is also a termination handler context that every new context made will link to so that if they terminate naturally, the handler will call the scheduler and swap out its context for the next one to run.

Benchmark:

We created a test case that created multiple threads. Those threads would iterate over a few million times each and we recorded the runtime of the program every 10 cycles. We took the last result to be the total runtime of the program. We varied the number of queues to be 1, 3, 5, and 10. We also varied the base queue time to be 1, 25, 50, and 100 milliseconds. Every queue after the base queue is given the next multiple of time.We were not able to use the provided testcases for benchmarking because they ran too quickly so there was no significant difference between the number of queues or allocated queue time.

		Number of Queues			
		1	3	5	10
Quanta Time	1	21.87	21.73	21.71	21.69
	25	21.73	21.3	21.3	20.41
	50	21.73	20.64	20.51	20.91
	100	21.14	21.54	20.81	20.54

Our results show that using a queue size of 5 with a base queue time of 50ms provided the fastest overall runtime for the program in seconds.