Eric Fiore: EJF96
Daniel Parks: DAP291
Erik Knorr: ERK58

Project 1

Operating Systems (416)

ReadMe

Structs (How data is stored)

Queue

Queues for both mutexes and the scheduler will use this struct

a) top
   - a link to the top first node in the queue.

b) bottom
   - a link to the last node in the queue.

my_pthread_t

All data about the thread structure will be saved here. Including thread identification, thread priority, how long a thread is running for its particular execution block, thread priority, thread status, a return value for a thread, pointers to any threads that are waiting for this thread to exit (for joining), execution context of the thread. A more detailed description is below:

a) thread_id
   - An integer identification of the thread

b) priority
   - The priority of the current thread as an integer value. This can be a value of 0 – 5 with 5 being the lowest priority and 0 being the highest priority.

c) intervals_run
   - An integer value of the number of time slices run in its current execution context. This is not a running total of the time slices a thread has run.

d) status
   - This can be:
     - Running
     - Yield
     - Wait_thread
     - Wait_mutex
     - Unlock
     - Thread_exit
     - Embryo

e) ret
   - the return value of the thread

f) uc
   - the execution context of the thread

my_pthread_mutex_t

data for any mutexes will be stores here. This will include a reference to a queue that will keep track of the threads waiting for the mutex and reference to the thread that currently has the mutex. More detailed information below.

a) waiting
   - the queue that will keep track of threads waiting for this mutex

b) user
   - a reference to the thread that currently has the mutex.

Eric Fiore: EJF96
Daniel Parks: DAP291
Erik Knorr: ERK58

Threads:

General information for threads:

Threads can be in one of several states considered by the scheduler during runtime:

- active: "able to be run" as in not waiting on anything
- yield: yields the processor to another waiting thread
  - Action: place the calling thread at the end of its current priority queue
- wait_thread: thread has called my_pthread_join() on another thread
  - Action: search a table of terminated threads for the thread being joined, if the thread being joined is found (that is the thread that need to be joined is completed) clean up the resources taken by the joining thread and continue the function of the calling thread.
- wait_mutex: thread has attempted to call my_pthread_mutex_lock()
  - Action: check the queue associated with the specified mutex. If the thread it at the front of the mutex queue, mark is as running and give it the mutex. If the thread is not at the head of the mutex queue, check the rest of the queue, and add the thread to the end of the queue if it is not already there, and move the thread to the end of its priority queue.
- Exit: thread has called exit
  - Action: remove it from the queue and deallocate the thread's resources

my_pthread_create(my_pthread_t * thread, pthread_attr_t, void *(*function)(void*), void* arg)

This function will accept incoming requests (through a call to the library) and create a new thread. The operation is described below.

When the first thread is created with my_pthread_create(), the scheduler is initialized. (see scheduler_int( ) for more information). When subsequent threads are created a context is created and the thread is attached to that context. Each thread context is linked to main. The TCB is created giving the thread an ID (see get_id( ) for more information) and the thread is placed at the end of the level 0 queue and set to active status. (more information can be found under the scheduler section. During the execution of the my_pthread_create( ) the timer is paused and resumed at the end of the function.

my_pthread_yield()

If a thread wishes to stop execution temporarily a call to my_pthread_yeild( ) can be made. The operation is described below.

The status of the calling thread is changed to yield and the SIGVTALRM is raised which alerts the scheduler to run. (see scheduler_alarm_handler( ) for more information)

my_pthread_exit(void *value_ptr)

When a thread has finished executing and is ready to exit a call to my_pthread_exit( ) can be made. The operation is described below.

The timer is paused, and the return value of the calling thread is passed to the TCB of running_thread. The status is changed to thead_exit and a flag is set to notify the system that the thread has completed. SIGVTALRM is raised and the scheduler will handle the alarm. (see scheduler_alarm_handler( ) for more information)

My_pthread_join(my_pthread_t thread, void **value_ptr)

Any threads that wish to pause until the execution of a sibling thread has terminated can call my_pthread_join( ). Once this function is called execution of the calling thread is paused until the sibling thread terminates. The operation is described below.

A check is made on the done flag to see if the sibling thread (the thread that need to finish before the calling thread) has already finished.

If the sibling thread is not done the timer is paused, the status is set to wait_thread and a reference to the waiting thread is passed into an array that keeps track of any threads that are currently waiting. (waiting[ ]) The SIGVTALRM is raised and the scheduler will handle the alarm. (see scheduler_alarm_handler( ) for more information)

If the sibling thread is done the timer is paused, any return value is passed. Cleanup is then performed including removing the calling thread form the wait table, freeing any memory that was taking by the thread that was used by the sibling thread, remove the sibling thread from the thread table and free the id of the sibling thread. The timer is then resumed..

MUTEXes:

General information about MUTEXes

NOTE we are not responsible for user error. If a user attempts to lock, or unlock the same mutex twice we cannot predict the outcome. If a user locks and does not unlock we cannot predict the outcome. If a user does not use the locks appropriately (i.e. lock all critical sections of code) we are not responsible.

MUTEXes will be represented by queues associated with global variables. When a thread requests a lock, it will either be given the lock if there are no other waiting threads, or be placed at the end of a queue of waiting threads. The front of the queue represents the user that is currently next in line for the lock and not the user currently using the lock. The user that is currently using the lock is represented in the my_pthread_mutex_t struct as "user".

my_pthread_mutex_init(my_pthread_mutex_t, const pthread_mutexattr_t * mutexattr)

If a user wishes to lock a critical section of code a mutex needs to be generated first. By creating a lock the user can use the lock later on to prevent unintended results from happening. The operation is described below.

Create a queue to represent the mutex and set the current user to NULL. Any mutex attributes are ignored.

my_pthread_mutex_lock(my_pthread_mustex_t *mutex)

if the user wishes to protect a section of code from unintended consequences a lock that was previously created in my_pthread_mutex_init( ) can be put into effect. The user is responsible for using the lock function correctly. If a section of code is used without calling my_pthread_mutex_lock( ) that section of code can be entered and changed. Only the correct use of the function will result in intended results. The operation is described below.

When the function is called the timer is paused and the current holder of the lock is checked. If the user is the calling user then resume the timer and exit the function. If the user is claimed by another thread add the lock to the queue created in my_pthread_mutex_lock( ) and add a reference to the waiting thread in the queue entry. Set the waiting thread's status to wait_mutex, resume the timer and raise SIGVTALRM. The scheduler will detect the alarm and take appropriate actions. (see scheduler_alarm_handler( ) for more information)

If the lock is not claimed then claim the lock and resume the timer.

my_pthread_mutex_unlock(my_pthread_mutex_t *mutex)

when the user decided a critical section of code has been passed and wished to unlock the user can request the lock be given up in order that another thread can gain access. The operation of the function is described below.

Upon entering the function, the timer is paused and a check is run to make sure the calling thread is the owner of the lock. If the calling thread is not the owner exit the function with an error. If the calling thread is the owner of the lock pass the lock to the next thread in the mutex queue. The thread that has newly has gained access to the lock has its status to active and it is placed back on the priority scheduler at its previous priority.

my_pthread_mutex_destroy(my_thread_mutex_t *mutex)

if a lock will no longer be in use the user should request that the lock be destroyed by calling my_pthread_mutes_destroy( ). Any resources used by the lock are cleaned at this point. The operation is described below.

When the function is first entered the timer is paused. Any threads that are in the mutex waiting queue are removed from the queue and placed at their previous priority level. Memory that was taken by the mutes is freed and the timer is resumed.

<u>Scheduler:</u>

General information about the scheduler

The scheduler is made up of a multi-tiered system implemented with 5 levels. The highest level of priority will be at level 0, and the lowest will be at level 4.

Each thread is given a time to run that is a multiple of 1 time slice[1]. The priority level that a thread is run at will determine the amount of time slices it receives. Threads at higher priority levels receive fewer time slices. Threads at lower priority receive more time slices. For example, threads that are at priority level 0 run for one time slice, threads at priority level 1 run for two time slices, up through priority level 4 run which runs for five time slices. This can be simplified by saying that a thread runs for $(\mathrm{T}+1)\alpha$ time where T is the priority level and α is a time slice[2].

Higher levels of priority are allowed to cycle their queues more often as compared to lower priority levels. This results in higher priority queues allowing more threads to run during that priority level's execution cycle as compared to a lower priority level. For example, if level 0 allows five threads to run then level 1 will allow four threads to run up to level 4 which will allow one thread to run. This is simplified by saying if priority level 0 runs *n* threads through during its execution cycle priority level 1 runs *n – 1* threads, priority level 2 runs *n – 2* threads until the lowest priority level which will run *n-(n-1)* threads.

The two above rules have the result that higher priority levels are allowed to run more threads during its execution cycle but each thread it runs is allowed to run for a shorter period of time as compared to lower level priority queues. Lower priority levels run less threads during its execution but each thread it does run is allowed to run for a longer period of time.

scheduler_alarm_handler(int signum)

Upon entering the scheduler the timer is paused. A switch statement is set up to determine the correct action. Each case of the switch statement is a possible thread status. The possible thread statuses are described in the thread section of this document.

- active:

  if the number of time slices has not reached its threshold for its current priority level then the timer is started again, and execution of the thread resumes. If the number of time slices has been reached the thread is placed into the appropriate level. That is if the priority level is any except the lowest level it is placed one spot lower. If the thread is at the lowest level it is placed back at the highest level again.
- yield:

  Place the queue back at the last spot of the queue at the current priority level
- wait_mutex

  no action is taken
- thread_exit

[1] A time slice is an interval of time that is kept track by an internal timer. For the purposes of this project each time slice is 25 milliseconds. The run time of each thread will be a multiple of one-time slice. Exceptions are made for threads that terminate before the time slice has expired. In this scenario the timer does not reach the end of the allotted time

[2] Please not that α will always be set at 1

check to see if there is another thread waiting on the running thread (through my_pthread_mutex_join( )). If there is a thread waiting set its status to active and place it back in the priority queue at its previous priority level

If the current priority level has run its allotted amount of time slices then increment the priority level unless the level is at the bottom level. If the priority level is at the bottom reset the level to the top.

Select a new thread to run by advancing the queue and set the context of the next thread to be run.

The timer is reset.

scheduler_init( )

this function is called by my_pthread_create( ) the first time it is run only. All five priority queues are created for later use as well as a queue where threads that have terminated will be placed. The context for main is created and main is attached to this context. Three timers are set up. One to count the running time of currently running threads, one for use when a thread timer will need to be paused, and the final is for the current time. The timer is started at this time and the signal handler is installed at this point.