CS416 - Project 2

User-level Thread Library and Scheduler

Nicholas Dundas

Liam O’Neill

Functions, Structures, Globals, Defines

Project Contains 4 main files for the user-level threads.

Only *thread-worker.h* and *mutex\_types.h* are exposed to the user while all others are hidden within the source files.

* *thread-worker\_types.h* – relevant thread metadata
  + struct TCB
    - stores thread id (worker\_t value exposed to user)
    - context for executing thread (for swapping and getting contexts)
    - a void\* which stores the address for the return value of the thread
    - size of the queue (only used if the current TCB is the head of a thread)
    - pointers to TCB structs for the next and previous TCB if in a queue
* *mutex\_types.h* – mutex and metadata
  + struct worker\_mutex\_t
    - stores whether the mutex is locked (atomic\_flag)
* *thread-worker.h* – main functions to be called by a user using the library
  + int worker\_create(worker\_t\*,pthread\_attr\_t,void\*(\*)(void\*),void\*)
    - creates a thread and store id in worker\_t\*
    - always returns 0 or calls exit(EXIT\_FAILURE) if it cannot run and outputs error
    - initializes scheduler and threading system upon first call
  + int worker\_yield()
    - resets scheduling timer and calls scheduler directly
    - always returns 0 or calls exit(EXIT\_FAILURE) if it cannot run and outputs error
  + void worker\_exit(void\*)
    - pushes running thread onto terminated queue and stores the address of the return value in the TCB structure
  + int worker\_mutex\_init(worker\_mutex\_t\*,pthread\_mutexattr\_t\*)
    - initializes a mutex (must be called first before any other mutex function)
    - always returns 0
  + int worker\_mutex\_lock(worker\_mutex\_t\*)
    - attempts to acquire a mutex and if locked places running thread into block queue and saves context in a locking loop
    - returns 1 when lock was unlocked and locks the lock
  + int worker\_mutex\_unlock(worker\_mutex\_t\*)
    - unlocks the locks and removes all threads from the blocking queue
    - calls the scheduler to run the next thread (prevent starvation) and returns 0
  + int worker\_mutex\_destroy(worker\_mutex\_t\*)
    - frees the lock by first attempting to lock in order to make sure it is not currently in use, then clears the set flag
* *thread-worker.c* – contains all function definitions and hidden helper functions/globals

Defines

* + STACK\_SIZE
    - defines size of the stack (set to 16\*1024)
  + QUANTUM
    - defines the length of time in microseconds allotted to each thread before timer interrupts (set to 10\*1000)

Globals

* + int initialized\_threads = 0
    - keeps track of whether worker\_create was called before and to call init\_workers()
  + struct sigaction sa
    - used to setup the signal handler for the timer interrupt
  + struct itimerval timer
    - used to interrupt the program to call the schedule() function every QUANTUM microseconds
  + tcb main\_thread
    - stores relevant metadata for the main executing thread
  + ucontext\_t schedule\_context
    - store main context for the scheduler function to be called once a thread exits
  + tcb\* q\_ready
    - ready queue that stores thread addresses ready to be run
  + tcb\* q\_block
    - block queue that stores thread addresses that are currently blocked
  + tcb\* q\_terminated
    - terminated queue that stores thread addresses that are finished executing and waiting to be joined
  + tcb\* running
    - stores the current running thread address

Functions

* Timer Related
  + static int timer\_stop()
    - returns the value of setitimer(ITIMER\_PROF,NULL,NULL)
  + static int timer\_start(struct itimerval\* timer)
    - returns the value of setitimer(ITIMER\_PROF,timer,NULL)
  + static int timer\_reset(struct itimerval\* timer)
    - calls timer\_stop() and sets the timer it\_values to the interval before calling the timer\_start() again, returns any errors it encounters
* Queue Related (can only be used on anything with q\_####)
  + static size\_t q\_size(const tcb\*)
    - returns size of queue structure
  + static void q\_init(tcb\*\*)
    - initializes a queue structure
  + static tcb\* q\_back(const tcb\*)
    - returns the last item of a queue
  + static tcb\* q\_pop\_front(tcb\*\* queue)
    - removes and returns the first element of a queue
      * returns NULL if no elements
  + static tcb\* q\_emplace\_back(tcb\*\*,tcb\*)
    - inserts second argument at the back of a queue and returns it
  + static tcb\* q\_find\_elem(tcb\* queue,worker\_t)
    - returns address of first tcb in queue that contains worker\_t value
  + static tcb\* q\_remove\_elem(tcb\*\*, tcb\*)
    - removes an element from a queue
      * second argument must be a tcb in the queue
      * if second argument is NULL then function is Nop
* Thread Helper Related
  + static tcb\* get\_thread(worker\_t)
    - return an address for whatever tcb holds the given worker\_t value, returns NULL if none
  + static worker\_t get\_unique\_id()
    - returns the lowest unique value for worker\_t that no other thread holds
  + static void worker\_run(void\*(\*)(void\*),void\*)
    - helper function for worker\_create context creation so it can call worker\_exit when it finishes executing
  + static void sig\_handle(int)
    - function for sa that is called via the timer interrupt, always calls scheduler
  + static void init\_workers()
    - sets up all globals and timers so the threading system begins automatically scheduling threads
  + static void schedule(bool emplaced)
    - calls sched\_rr(emplaced)
  + static void sched\_rr(bool emplaced)
    - the RR scheduler and emplaces the running thread at the back of the ready queue if emplaced is false

It should be noted that any worker\_#### function stops the timer when called and only resumes timer immediately preceding the swapcontext() call at the end of the scheduler

Design

When designing the Threading we wanted to ensure that the minimum amount of system calls and instructions were desired when running for creating as little overhead as possible. To achieve this, we implemented doubly linked lists that were built into the thread metadata. Every thread aside from the main executing thread will use the data so the extra bytes from unused pointers is minimal. This allows us to only have to malloc for two of the functions, *get\_unique\_id()* and *worker\_create()* respectively. The former requiring it so we can utilize an array created on the heap that stores whether a given index has an associated TCB value with it. Such an array can be of any size determined by the number of threads. The latter malloc related function is required for creating the stack for which each thread uses. All other functions just require setting member values of the structures, so they do not require any expensive system allocation operations. The queues can be accessed via the respective functions outlined above. The timer functions were simplified due to their extensive reuse as we call timer\_stop() on any function that affects the queue so the timer does not interrupt in the middle of such a critical execution segment that would break queue structures. Due to the project specifications all attributes of mutexes and threads were ignored.

The *worker\_create()* function is the first function you might typically call and it will call *init\_workers()* once when first called. It uses the global variable *initialized\_threads* to keep track of this. *init\_workers()* also saves the state of the main thread as well setting up the timer interrupts for the scheduler. Assuming we continue *worker\_create()* we then push the new thread to *q\_ready* once all of its relevant metadata is setup with makecontext() and local stack creation. Worker yield lets the user directly call the scheduler to stop executing the current thread and reset the timer.

*worker\_exit*() can be called at the end of functions to set the address of the return value. It emplaces the running thread at the end of *q\_terminated* and calls the scheduler with emplaced set to true.

*worker\_join()* blocks a thread from execution until another thread is in *q\_terminated*. It does this by constantly checking whether a thread is in said queue and attempting to remove it. If it fails it yields the current thread otherwise it sets the second argument (if not NULL) to whatever the address of the return value is.

*worker\_mutex\_init()* simply clears the atomic flag of the mutex

*worker\_mutex\_lock()* attempts to lock the mutex with the test and set atomic function and if it is already locked it will enter a loop where it pushes the thread to *q\_block* and calls the scheduler with emplaced set to true. This effectively implements a better spinlock by only spinning once then blocking until another lock calls unlock.

*worker\_mutex\_unlock*() will clears the atomic flag of the mutex and push all blocked threads to the *q\_ready* allowing them to try to test each of their locks. It then calls the scheduler so another thread may execute preventing any starvation.

*worker\_mutex\_destroy*() attempts to lock the lock then clears the atomic flag of the mutex

Scheduler

The scheduler uses Round Robin Scheduling where after a specified quantum, we pop and save the front *q\_ready* with *q\_pop\_front().* The current executing thread is pushed to the back of a *q\_ready* with *q\_emplace\_back()* and the current executing thread is swapped via swapcontext() between the last thread executing and the recently popped TCB from the queue. Timer stop is called at the start to prevent any interrupts from ruining execution of critical parts where the queue structure is changed. All the thread states are stored implicitly by knowing which queue they are in and the interactions each of the functions have with the queues. The scheduler will only automatically schedule anything in *q\_ready* while worker\_join() only checks *q­­­\_terminated* for any threads that are done. *q\_block* is only interacted with via the mutexes so it exists as a place that is ignored by normal threading functions allowing threads to not take up any time in the scheduler like a typical spinlock would.

References and Collaboration

The main references consulted were the Linux Man pages and cppreferences. The main collaborators for this project were Nicholas Dundas and Liam O’Neill. These references were used to see what functions would return and if there we any error values that needed to be checked. We shared and referenced the same pages to ensure that we know the functions we were working with. Utilizing github and groupme we could effectively collaborate and share changes in real time with ease. This allowed us to work together efficiently and know who had done what.