CS416 Project 2

Christopher Yong (cy287), Joshua Ross (jjr276)

Structs

Thread Control Block

In our program, the thread control block has 7 attributes.

- rpthread_t id
- rpthread_t joinTID
- int priority
- int status
- uint desiredMutex
- ucontext_t context
- void* exitValue

id is the identifer of the thread.
joinTID is used to indicate which thread ID, the current thread is waiting on.
priority is used to indicate the priority of the thread (used in MLFQ)
status is used to indicate the current state of the thread (READY, SCHEDULED, BLOCKED, WAITING)
desiredMutex is used to specify which mutex it wants and is waiting for
context is used as the context of the thread
exitValue is used to store the return value of rpthread_exit and set it in rpthread_join.

Mutex

In our program, the mutex structure has 4 attributes.

- uint id
- rpthread_t tid
- volatile int lock
- rpthread_t waitingThreadID

id is the identifier of the mutex, we use this to keep track of what threads want which mutex.
tid is used to keep track of which thread currently holds this mutex.
lock is used for the test_and_set, this is value that gets tested.
waitingThreadID is used to see which thread will get dequeued off the blocked list and given the chance to run and attempt to obtain the mutex when the mutex is unlocked.

Scheduler

In our program, the scheduler structure has 4 attributes.

- uint numberOfQueues
- Queue* priorityQueues
- char usedEntireTimeSlice
- uint timeSlices

numberOfQueues is the number of queues in the scheduler
priorityQueues is an array of queues that are the different priorities in MLFQ (in RR, it will just use the MAX PRIORITY queue)
usedEntireTimeSlice is used to indicate if the current thread used the entire time slice
timeSlices is used to record the number of times the scheduler function has been entered (used for MLFQ for priority boosting)

Queue

QueueNode

In our program, the QueueNode structure has 2 attributes.

- tcb* node
- QueueNode* next

node is the a thread's thread control block and it indicates a thread.
next is the next node in the queue.

Queue

In our program, the Queue structure has 3 attributes.

- QueueNode* head
- QueueNode* tail
- int size

head represents the front of the queue.
tail represents the back of the queue.
size represents the number of nodes in the queue.

Thread Library

rpthread_create

If this is the first call to rpthread_create, we perform an initialization of the library.
Initialization (First Call)

Step 1) Initialize our scheduler struct and the associated queues in the struct

Step 2) Initialize our join queue, blocked queue, and exit queue

Step 3) getcontext for our global scheduler context and malloc the stack and set the stack flags and size and link the scheduler context to schedule function via makecontext

Step 4) getcontext for our global exit context and malloc the stack and set the stack flags and size and link the exit context to the rpthread_exit via makecontext

Step 5) Initialize the main TCB context (mallocing the stack and setting the associated flags and stack size) and use getcontext to initialize the maintcb context and uc_link to the exit context and also set the global current running thread variable to mainTCB

Step 6) Initialize the sigaction handler and initialize the timer and start the timer

After initialization (First and following calls)

Step 1) Pause the timer

Step 2) Create a new TCB (mallocing the stack and setting the stack size and flags) and uc_link the new TCB context to the exit context and lastly link the function pointer passed in to rpthread_create to this new TCB context via makecontext

Step 3) Set the passed in rphread_t* thread to the thread ID of the newly created TCB

Step 4) Enqueue the new TCB into the scheduler queues

Step 5) Resume the timer

Step 6) return 0

For all the following rpthread_create calls, the program will perform only the **After initialization** section.
Note: every new thread including the mainTCB priority will initially be set at max priority and also when it tries to set the threadID to the global threadID, it must retrieve a mutex and if the mutex is locked, spinlock with yield

rpthread_yield

Step 1) Disable timer

Step 2) save the current context to the TCB and load the scheduler context via swapcontext

Step 3) when the thread returns after the swap (meaning it got scheduled to run again), return 0

rpthread_exit

Step 1) Disable timer

Step 2) Save the value_ptr to the exitValue of the current running thread's TCB

Step 3) Free the context stack

Step 4) Search the join queue for a thread that is waiting for the current thread to terminate
Case 1: Finds a thread:

- 4a) Dequeue the found thread off the join queue
- 4b) set the necessary fields (Status back to Ready, joinTID to 0, and set the found thread's exitValue = current's exitValue)
- 4c) Enqueue the found thread back into the priority queues in scheduler struct so it can scheduled again
- 4d) enqueue the current thread into the terminatedAndJoined queue

Case 2: Does not find a thread:

- 4a) enqueue the current thread into the exit queue

Step 5) set the current running thread to NULL

Step 6) setcontext to scheduler context

rpthread_join

Step 1) Check if the terminated thread id is greater than or equal to the global generating threadID variable and if it is, return -1

Step 2) check if the terminated thread id is 0 or the current running thread id, return 0 if so (we check terminated thread id is 0 because for our program, 0 represents -1 since we start generating the thread ids at 1.)

Step 3) Pause timer

Step 4) Search the exit queue for the terminated thread
Case 1: Finds the terminated thread

- 2a) sets the value_ptr to the terminated thread's tcb's exitValue
- 2b) enqueue the terminated thread into the terminatedAndJoined queue

Case 2: Does not find the terminated thread

- 2a) Checks the terminatedAndJoined queue to see if the terminated thread has already exited and joined by another thread or if in the join queue, there is another thread already waiting for the terminated thread to terminate, if either of these conditions are true then return -1, otherwise continue to next step (since we cannot have two joins on the same terminated thread)
- 2b) Set the current thread's status to WAITING, set the current's joinTID to the thread ID it is waiting on
- 2c) Enqueue the current thread into the joinQueue
- 2d) Save the current thread's context to current thread tcb and load the scheduler context via swapcontext
- 2e) When the thread resumes, pause the timer
Note: when this thread resumes, we know that the only way it could have resumed is if the thread it was waiting on, terminated and put this thread back on the run queue (and the terminated exit value is also stored in the current's exit value)
- 2f) Set the value_ptr to the current thread's exitValue

Step 5) Resume timer

Step 6) Return 0

Thread Synchronization

rpthread_mutex_init

Step 1) Check if mutex is NULL, return -1 if NULL

Step 2) Attempt to get the mutexIDMutex (if it does not get it, it spinslock but yields instead the while loop instead of just regular spinlock)

Step 3) Assign the mutex ID to the global mutex ID and increment the global mutex ID

Step 4) Release the mutexIDMutex

Step 5) Set the tid of the mutex to 0

Step 6) Set the lock of the mutex to 0

Step 7) Set the waiting thread ID to 0

Step 8) Return 0

rpthread_mutex_lock

Step 1) Check if mutex is null, if so return -1

Step 2) Attempt to get the blockedQueueMutex (spinwait with yield until it gets it)

Step 3) Test and set the mutex's lock attribute (header of the while loop)
Case 1: Does not get the mutex

- 3a) Disable timer
- 3b) Set the current status to BLOCKING and the desiredMutex field to the mutex's id
- 3c) Enqueue the current thread into blockedQueue
- 3d) Release the blockedQueueMutex
- 3e) Save current context to current's TCB and load the scheduler context via swapcontext
- 3f) When this thread resumes, it has the **CHANCE** to get the mutex again, so we first: attempt to get the blockedQueue mutex (spinwait with yield until it gets it)
- 3g) Set mutex waitingThreadID to 0
- 3h) Repeat the beginning of step 3 (so attempt to get the mutex's lock again)

Case 2: Gets the mutex

- 3a) Continue to step 4

Step 4) Set the mutex's tid field to the current thread's id

Step 5) Release the blockedQueueMutex

Step 6) Return 0

rpthread_mutex_unlock

Step 1) Check if the mutex is null or if the mutex is not locked by the current thread, or the mutex is not even locked, for any of these cases, return -1

Step 2) Set mutex tid field to 0 and sync lock release the mutex's lock attribute

Step 3) Attempt to get the blockqueueMutex (spin wait with yield until it gets it)

Step 4) Check if the mutex's waitingThreadID is 0 (meaning there is no thread waiting on this mutex) and if it is not 0, release the blockedQueueMutex and continue to step 5 otherwise proceed to step 4a

  4a) Search the blocked queue and dequeue the first thread it finds waiting on this mutex id

  Case 1: Does not find a thread waiting for this mutex

  i. Release the blockedQueueMutex

  Case 2: Finds a thread waiting for this mutex

  i. Set the mutex waitingThreadID to the found thread's ID
  ii. Release the blockedQueueMutex
  iii. Set the found thread status to READY
  iv. Set the found thread desiredMutex attribute to 0
  v. Enqueue the found thread back into the scheduler queues

Step 5) Return 0

rpthread_mutex_destroy

Step 1) Pause timer

Step 2) Check if the mutex is NULL, if NULL, resume timer and return -1

Step 3) Check if any threads are waiting for this mutex, or if the mutex is currently locked and if so, resume timer and return -1

Step 4) Set the mutex fields all to 0

Step 5) Resume timer

Step 6) Return 0

Scheduler

Setup

Before entering the RR or MLFQ functions, if the current thread's status is WAITING or BLOCKED, set current to NULL, so it does not get enqueued into the scheduler runqueues.
Note we keep a global thread control block pointer called current thread to indicate the current running thread's information.

Preemptive Round Robin

Note for RR, all threads are stored in the topmost priority runqueue in the scheduler struct.

1) Dequeue the topmost priority runqueue in the scheduler struct
   Case 1: Found a thread
   (a) If the current running thread is not null, enqueue back into the topmost priority runqueue in the scheduler struct
   (b) set the global current thread variable to the found thread

   Case 2: Did not find a thread (meaning runqueue is empty)
   (a) continue to next step (meaning it is leaving the current thread as the next thread to run)
2) start the timer
3) setcontext to the global current thread's context

Multilevel Feedback Queue

Step 1) Increase the time slice attribute in scheduler struct (indicates the number of times scheduler has been entered)

Step 2) Check if time slice attribute in scheduler struct, equals the threshold for boosting the priorities of all threads in the runqueues (if so, boost all the threads) and reset the time slice attribute back to 0

Step 3) Check if the global current running thread used the entire time slice, if it did, lower the priority

Step 4) Find the next thread to run by searching the topmost priority queue and if there are no threads to dequeue, go to next queue.
Case 1: Finds a thread
3a) Enqueue the global current thread into the scheduler run queues
3b) Set the found thread as the global current thread

Case 2: Does not find a thread

3a) Leave the current thread as the global current thread

Step 5) start the timer

Step 6) setcontext to the global current thread's context

Benchmarks

RPTHREAD library: vector_multiply

Round Robin

Results from running vector_multiply with the Round Robin Scheduler

- 1 thread: 79 ms
- 5 threads: 81 ms
- 10 threads: 82 ms
- 20 threads: 90 ms
- 50 threads: 103 ms
- 100 threads: 98 ms
- 500 threads: 91 ms

MLFQ

Results from running vector_multiply with the MLFQ Scheduler

- 1 thread: 79 ms
- 5 threads: 81 ms
- 10 threads: 82 ms
- 20 threads: 85 ms
- 50 threads: 102 ms
- 100 threads: 93 ms
- 500 threads: 92 ms

PTHREAD library: vector_multiply

Results from running vector_multiply with the pthread library

- 1 thread: 58 m
- 5 threads: 215 m
- 10 threads: 296 m
- 20 threads: 284 m
- 50 threads:324 m
- 100 threads: 347 m
- 500 threads: 449 m

RPTHREAD library: parralel_calc

Round Robin

Results from running parralel_calc with the Round Robin Scheduler

- 1 thread: 2565 ms
- 5 threads: 2569 ms
- 10 threads: 2568 ms
- 20 threads: 2566 ms
- 50 threads: 2566 ms
- 100 threads: 2565 ms
- 500 threads: 2569 ms

MLFQ

Results from running parralel_calc with the MLFQ Scheduler

- 1 thread: 2574 ms
- 5 threads: 2565 ms
- 10 threads: 2567 ms
- 20 threads: 2565 ms
- 50 threads: 2587 ms
- 100 threads: 2570 ms
- 500 threads: 2571 ms

PTHREAD library: parralel_calc

Results from running parralel_calc with the pthread library

- 1 thread: 2583 ms
- 5 threads: 898 ms
- 10 threads: 645 ms
- 20 threads: 597 ms
- 50 threads: 586 ms
- 100 threads: 583 ms
- 500 threads: 621 ms

RPTHREAD library: external_cal

Round Robin

Results from running external_cal with the Round Robin Scheduler

- 1 thread: 7029 ms
- 5 threads: 7021 ms
- 10 threads: 7026 ms
- 20 threads: 7052 ms
- 50 threads: 7004 ms
- 100 threads: 7027 ms
- 500 threads: 7020 ms

MLFQ

Results from running external_cal with the MLFQ Scheduler

- 1 thread: 7056 ms
- 5 threads: 7032 ms
- 10 threads: 7026 ms
- 20 threads: 7032 ms
- 50 threads: 7047 ms
- 100 threads: 7041 ms
- 500 threads: 7030 ms

PTHREAD library: external_cal

Results from running external_cal with the pthread library

- 1 thread: 7493 ms
- 5 threads: 3056 ms
- 10 threads: 2515 ms
- 20 threads: 2528 ms
- 50 threads: 2483 ms
- 100 threads: 2489 ms
- 500 threads: 2493 ms