CS 518 - Project 2: User-level Thread Library and Scheduler

Submitted by Anmol Arora (aa2640) and Raunak Negi (rn444)

Data Structures :

- typedef uint **worker_t**:
  Unsigned Integer variable that is used to identify threads using their thread IDs.
- typedef struct **TCB**{}:
  t_id: thread ID is a unique identifier for a thread.
  t_context: stores the context of the thread which basically points to the line of execution.
  t_status: defines the status of the thread. A thread can be in any one of the following states as mentioned according to the write up:
  SCHEDULED -> the currently running thread
  READY -> ready for execution
  BLOCKED -> waiting for the mutex(lock) to be available in order to enter a critical section.
  t_priority: corresponding to queues in MLFQ
  elapsed: which indicates if the time quantum has expired since the time thread was scheduled or was the thread yielded before the expiration of the time quantum. Since we are not aware about how long the jobs will run we use elapsed time to measure how long the thread is yet to run.
  is_blocked: checks if the thread is blocked or not
  Is_yielded: checks if the thread yielded before time quantum expired.
- typedef struct **tcb_node**{}:
  thread_control_block: points to the tcb data structure of the current node
  next: points to the next node in the queue
- typedef struct **queue**{}:
  head: the top/first element in the queue
  tail: the last element of the queue
- typedef struct **worker_mutex_t**{}:
  flag: is used to indicate whether mutex is available or not
  thrd: points to the tcb of the thread holding the mutex

Part 1: Thread Library (50 Points)

1.1 Thread Creation

int worker_create(worker_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg)

This function serves as an alternative implementation of the pthread_create function, facilitating the creation of a new thread within the program. Upon invocation, this function initializes the thread control block (TCB), assigns a unique thread ID for identification purposes, creates a new context pointing to the specified function parameter, and subsequently enqueues the thread into the ready_queue.

During the initial call, essential data structures, including the TCB and run queues, are initialized. Additionally, the main thread, scheduler, timer interrupt, and the corresponding signal handler are set up. The function ensures proper initialization procedures by utilizing a global boolean variable to track its first-time invocation.

Upon initialization of the main_thread, it is placed into the run queue, and the context switches to the scheduler. The scheduler then resumes the execution of the main thread from the point where it was paused upon entering the run queue initially. Subsequently, the remaining code is executed. In the subsequent calls, the function creates a new thread, initializes its TCB, configures its context, and manually allocates dynamic memory for the stack. Finally, the newly created thread is added to the run queue in the ready state, ready for execution.

1.2 Thread Yield

int worker_yield();

This function handles the transition of the current thread from the SCHEDULED state to the READY state in two scenarios: when the thread completes its execution before the allocated time quantum or when it voluntarily allows another thread to use its remaining runtime.

The function swaps contexts with the next scheduled thread. Importantly, it also preserves the current thread's context, ensuring it can resume execution from the point where it left off when it is scheduled again in the future. It also increments the global tot_cntx_switches variable to keep track of all the context switches.

1.3 Thread Exit

void worker_exit(void *value_ptr)

The function is responsible for releasing the memory allocated for a thread's stack and Thread Control Block (TCB) once the thread has completed its execution and is no longer needed.

Noteworthy, the API also notes down the completion time of the thread and modifies global variables, including avg_turn_time and avg_resp_time, which are later utilized in the print_app_stats() function.

1.4 Thread Join

int worker_join(worker_t thread, void **value_ptr)

The function manages a situation where the calling thread needs to wait for another thread to finish its execution or terminate. If value_ptr is not NULL, the return value of the existing thread will be passed back.

1.5 Thread Synchronization

1.5.1 Thread Mutex Initialization

int worker_mutex_init(worker_mutex_t *mutex, const pthread_mutexattr_t *mutexattr){}

This function initializes a worker_mutex_t created by the calling thread. It sets the flag attribute of the **worker_mutex_t** to 0 indicating that the mutex is available.

1.5.2 Thread Mutex Lock and Unlock

int worker_mutex_lock(worker_mutex_t *mutex);

This function sets the lock for the given mutex and other threads attempting to access this mutex will not be able to run until the mutex is released.

int worker_mutex_unlock(worker_mutex_t *mutex)

The mutex structure is dismantled, resetting it to its initial state and releasing its held resources. Additionally, the threads that were previously blocked are now made available for execution. This is achieved by using the “free_jobs()” utility function, which handles the task of updating the status of blocked threads, marking them as ready, and placing them back into the appropriate run queues for future scheduling.

1.5.3 Thread Mutex Destroy

int worker_mutex_destroy(worker_mutex_t *mutex);

This function utilizes the built-in test-and-set atomic operation to determine whether a mutex can acquire the lock. The ‘flag’ variable, where 1 indicates that the mutex is already locked, is examined. If another thread has already acquired the mutex, the current thread is placed in the “blocked_threads” queue, and the context switches back to the scheduler. However, if the mutex

is successfully acquired, the mutex's tcb is set to the current thread’s tcb, and the flag variable is changed to 1 indicating that the mutex is unavailable and returns 0 to indicate successful acquisition.

Part 2: Scheduler (40 Points)

2.1 Pre-emptive SJF (PSJF)

In this scheduling policy called PSJF (Preemptive Shortest Job First), the code uses a unique approach. Instead of knowing exactly how long each thread will run, it estimates based on the number of time quantums a thread has used completely.

Incrementing Time Elapsed: When a thread runs, the scheduler keeps track of how many time quantums it has used completely by increasing its "elapsed" counter. If the thread has finished its execution before the time quantum expires or when it voluntarily allows another thread to use its remaining runtime, in such cases the elapsed time is not incremented.

Choosing the Next Thread: The scheduler checks all threads in the ready queue. The one with the smallest "elapsed" value (indicating it has used the least time) is chosen to run next. If there's a tie, the scheduler picks the first thread in the queue.

Actual Execution: The chosen thread is then set up to run, involving tasks like setting timers and configuring the context for the thread. The scheduler dequeues a thread with the smallest elapsed time from the ready_queue. After dequeuing, the thread's status is set to SCHEDULED, and the is_blocked flag is marked as 0, indicating it's ready to execute. A timer is configured to interrupt the thread's execution after a specific time quantum, preventing any single thread from monopolizing the CPU. The current time is recorded for scheduling analysis, and the total context switches counter is incremented. Finally, the thread's context is set, allowing it to resume execution.

2.2 Multilevel Feedback Queue (MLFQ)

This function evaluates whether a thread has yielded to the scheduler. If not, it reduces the thread's priority in the Thread Control Block (TCB) structure . This decrease signifies that the specific thread (working_thread) is moved to a lower-priority queue for one complete quantum time slice. If the thread does yield, its priority remains unchanged. If the priority is lowered, the current thread is transferred to the lower-priority queue. If the priority remains the same, the thread is reinserted into the original run queue it was removed from during the scheduling process. Subsequently, threads in the highest-priority non-empty run queue are scheduled to run in a round-robin fashion. The actual execution, including setting timers and configuring the context for the selected threads in the highest non-empty priority queue, is managed by the sched_rr(queue *queue) function.

Part 3: Helper Functions

void enqueue(tcb *newThread, queue *queue);

The enqueue function is responsible for adding a new thread, represented by the tcb (Thread Control Block) structure, to a specified queue (ready_queue).
If the queue is not empty (indicated by a non-null tail), the new node is linked to the end of the queue: it becomes the next node after the current tail, and then the tail is updated to point to the new node, making it the new tail of the queue.
Additionally, the function records the arrival time of the newly added thread using the clock_gettime function with the CLOCK_REALTIME clock, storing this timestamp in an array (arrival_time) for future reference and analysis.

tcb_node* identifySmallestElapsedNode(tcb_node *head){}

The function identifySmallestElapsedNode takes a linked list of threads represented by tcb_node structures as input. It iterates through the list, comparing the elapsed time attribute (elapsed) of each thread's tcb structure. The function identifies and returns the node with the smallest elapsed value, indicating the thread that has utilized the least CPU time so far.

void adjustQueueForSmallestNode(tcb_node *head, tcb_node *smallestNode){}

The function operates on a linked list of threads represented by tcb_node structures. It takes the head of the list (head) and a specific node (smallestNode) as input. The function rearranges the list so that the smallestNode, representing the thread with the shortest execution time, becomes the new head of the list. It traverses the list to find the smallestNode, detaches it from its current position, and reattaches it at the beginning of the list. This adjustment ensures that the thread with the smallest execution time is positioned at the front of the list, ready for immediate execution.

void setQuantumTimer(struct itimerval timer, int quantumTime)

The function configures a timer represented by the struct itimerval to enforce a specific time quantum (quantumTime). The timer's initial value is set to the specified quantum time in microseconds, ensuring precise timing for thread execution. The timer is then set using the setitimer function with the ITIMER_PROF option, indicating that the timer decrements in real time.

void handleCurrentThread(queue *jobs_queue)

The function manages the current thread's status and handling within the scheduling system. If the current thread has not yielded voluntarily, indicating it has utilized its time quantum, the

function increments its elapsed time. However, if the thread has yielded, indicating it relinquished its remaining time, the yielded status is reset. Regardless of the yield status, the thread is marked as ready and re-enqueued into the jobs_queue

static void sched_rr(queue *queue)

The sched_rr function is a key component of the Round Robin scheduling algorithm. It selects the next thread to execute from the specified queue (queue). If the queue is not empty, the function dequeues a thread and adjusts the queue accordingly. The dequeued thread becomes the current thread (curr_thrd), and its state is set. The function then sets a timer to enforce the time quantum, records the scheduling time, increments the context switch counter, and switches the context to execute the selected thread.

Part 4: Results and Analysis

Benchmark results of Linux's **pthread** library with different configurations of pthread thread number

Thread Count	Linux pthread performance					
	vector_multiply		parallel_cal		external_cal	
	PSJF	MLFQ	PSJF	MLFQ	PSJF	MLFQ
4	283	323	690	821	2722	2037
6	303	327	490	656	2471	2487
8	428	431	530	401	2664	2617
16	404	383	353	293	3031	2820
32	616	591	265	267	3058	3014
64	698	721	192	178	3082	3018
128	799	797	145	176	2573	2820

Benchmark results of of our **worker_t** library with different configurations of worker thread number

Thread Count	worker_t performance (micro-seconds)					
	vector_multiply		parallel_cal		external_cal	
	PSJF	MLFQ	PSJF	MLFQ	PSJF	MLFQ
4	51	51	3931	5018	947	921
6	61	53	2955	2945	764	845
8	64	53	3156	3404	1003	795
16	59	58	6506	2808	1077	885
32	80	78	2701	3505	1098	718
64	94	119	2705	2675	1189	1074
128	75	77	3916	2683	1056	921

Analysis:

1) Vector_multiply: Our worker thread library outperforms the pthread library for both the scheduling algorithms (i.e PSJF and MLFQ).
2) Parallel_cal: the pthread library outperforms our worker thread library for both the scheduling algorithms (i.e PSJF and MLFQ).
3) External_cal: Our worker thread library performs better than pthread library for MLFQ scheduling algorithm.
4) For all the files in our worker thread library there is no significant difference in the run time depending on the scheduling algorithms.

Part 5: Collaboration and References

1. Recitation 4 and 5
2. Piazza Class discussion
3. Links in the write up
4. https://en.cppreference.com/w/c/chrono/timespec