Project 2: User-Level Thread Library

Name: Kelvin Ihezue, Bryan Shangguan
iLab usernames: ki120, bys8
Environment: macOS (uses ucontext; warnings suppressed via -Wno-deprecated-declarations)
Build:
make clean && make SCHED=PSJF in project root,
Then cd benchmarks && make clean && make && ./genRecord.sh

PART 1

1) Implementation Details

For Part 1 I implemented a user-level threading library with:

- **a TCB (thread control block) and a FIFO ready queue,**
- **Core APIs: worker_create, worker_yield, worker_exit, worker_join,**
- **Preemption via SIGPROF + a scheduler context (effective preemptive round-robin over the FIFO queue).**
- **basic non-recursive mutexes (worker_mutex_init/lock/unlock/destroy),**
- **Stats: total context switches (+ placeholders for avg turnaround/response).**
- **A small test plus three benchmarks.**

Data structures

- **TCB: t_id, status (READY/RUNNING/BLOCKED/COMPLETED), ucontext_t context, stack/size, start_routine/start_arg, return_value, is_finished, join state (waiting_on, joiner_tid), timing fields, and next pointers for the ready queue/registry/mutex waiters.**
- **Ready queue: simple FIFO (head/tail). rq_enqueue/rq_dequeue are O(1).**
- **Global state: main_context, scheduler_context, curr, registry (all_threads), next_tid.**

1.2 API Logic

worker_create

- one-time init (capture main_context, set up scheduler_context, arm timer).
- For each thread, allocate TCB + stack, set uc_link=&scheduler_context, start via _trampoline() which calls the user function then worker_exit

worker_yield

- mark current as READY, enqueue, swapcontext to scheduler (or from main to scheduler if no current).

worker_exit

- store return value, mark COMPLETED, wake any joiner, switch to scheduler (doesn't return).

worker_join

- if target is already COMPLETED, return value and free its resources; otherwise caller blocks and the scheduler runs until the target exits.

1.3) Preemption + scheduler (RR)

- The timer handler (SIGPROF) fires every quantum and, if a thread is running, marks it READY, enqueues it, and swapcontext s into the scheduler.

- schedule() pops the next READY TCB, marks it RUNNING, bumps the context-switch counter, and setcontexts to it. If there's nobody to run, it returns to main_context.

- This is effectively a preemptive round-robin over a single ready queue.

1.4) Mutexes (Part 1.5)

- worker_mutex_init: sets owner=0, clears an internal wait queue, and marks init=1.

- lock:
  - If free, take it.

  - If owned by the same thread, return EDEADLK (non-recursive).

  - Otherwise, mark caller BLOCKED, queue it on the mutex wait list, and swapcontext to the scheduler. When it resumes, it should own the lock.

- **I also handle the case where "main" locks/unlocks (it can yield via the scheduler until the lock becomes free).**

- **Unlock:**
  - **Only the owner can unlock (EPERM otherwise).**
  - **If there's a waiter, transfer ownership to the next waiter and enqueue it READY; else set owner=0.**

- **destroy: EBUSY if a thread owns the lock or there are waiters; otherwise reset the fields.**

What I tested

benchmarks/test.c

- cooperative alternation inside a mutex.
- **Mutex edge cases:** re-lock same mutex returns error; destroy returns EBUSY if locked/waiters; destroy succeeds after clean up.
- **Preemption smoke:** two tight loops (no yields) both make progress → timer preemption works.

2) Benchmarks & Results

How I ran them (Included in the ReadMe)

```
cd project-2                    # build from the project root first.
make clean && make SCHED=PSJF

cd benchmarks
make clean && make          # builds external_cal, parallelCal, vector_multiply against
./genRecord.sh          # required for external_cal

./external_cal 6
./parallelCal 6
./vector_multiply 6

# also tested larger thread counts:

./external_cal 50
./external_cal 100
./parallelCal 50
./parallelCal 100
./vector_multiply 50
./vector_multiply 100

make test                #Optional: To run the test program under the benchmarks
```

./test

Results

```
● kelony@Kelvins-MacBook-Pro OS-Design % cd project-2
● kelony@Kelvins-MacBook-Pro project-2 % make clean && make SCHED=PSJF
  rm -rf testfile *.o *.a
  rm -rf testfile *.o *.a
  gcc -pthread -g -c -Wall -Wextra -O2 -I. -DUSE_WORKERS=1 -Wno-deprecated-declarations  -DPSJF thread-worker.c -o thread-worker.o
  thread-worker.c:539:13: warning: unused function 'sched_mlfq' [-Wunused-function]
    539 | static void sched_mlfq() {
        |             ^~~~~~~~~~
  thread-worker.c:555:13: warning: unused function 'sched_cfs' [-Wunused-function]
    555 | static void sched_cfs(){
        |             ^~~~~~~~~
  2 warnings generated.
  ar -rc libthread-worker.a thread-worker.o
  ranlib libthread-worker.a
● kelony@Kelvins-MacBook-Pro project-2 % cd benchmarks
● kelony@Kelvins-MacBook-Pro benchmarks % make clean && make
  rm -f external_cal parallelCal vector_multiply test
  rm -rf record *.dSYM
  cc -O2 -I.. -DUSE_WORKERS=1 external_cal.c -L.. -lthread-worker -o external_cal -pthread
  cc -O2 -I.. -DUSE_WORKERS=1 parallel_cal.c -L.. -lthread-worker -o parallelCal -pthread
  cc -O2 -I.. -DUSE_WORKERS=1 vector_multiply.c -L.. -lthread-worker -o vector_multiply -pthread
● kelony@Kelvins-MacBook-Pro benchmarks % ./genRecord.sh
● kelony@Kelvins-MacBook-Pro benchmarks % ./external_cal 6
  ***************************
  Total run time: 451770 micro-seconds
  Total sum is: 149345857856
  Verified sum is: 149345857856
  Total context switches 43
  Average turnaround time 0.000000
  Average response time  0.000000
  ***************************
```

```
***************************
● kelony@Kelvins-MacBook-Pro benchmarks % ./parallelCal 6
***************************
Total run time: 2018 micro-seconds
Verified sum is: 83842816
Total sum is: 83842816
Total context switches 124
Average turnaround time 0.000000
Average response time  0.000000
***************************
● kelony@Kelvins-MacBook-Pro benchmarks % ./vector_multiply 6
***************************
Total run time: 4842 micro-seconds
Verified sum is: 8999995500000500000
Total sum is: 8999995500000500000
Total context switches 6
Average turnaround time 0.000000
Average response time  0.000000
***************************
● kelony@Kelvins-MacBook-Pro benchmarks % ./external_cal 100
***************************
Total run time: 450245 micro-seconds
Total sum is: 149345857856
Verified sum is: 149345857856
Total context switches 138
Average turnaround time 0.000000
Average response time  0.000000
***************************
⊗ kelony@Kelvins-MacBook-Pro benchmarks % .vector_multiply 70
zsh: command not found: .vector_multiply
● kelony@Kelvins-MacBook-Pro benchmarks % ./vector_multiply 70
***************************
Total run time: 23881 micro-seconds
Verified sum is: 8999995500000500000
Total sum is: 8999995500000500000
Total context switches 70
Average turnaround time 0.000000
Average response time  0.000000
***************************
```

3) Analysis & pthread Comparison

3.1) What the numbers say

- **Correctness:** For all three benchmarks, the "total sum" always matched the "verified sum" for 1, 6, 50, and 100 threads. So the library is producing the right answers.
- **parallelCal:** Pure CPU math and pretty simple. You see small, steady improvements as you add threads, but it's still one kernel thread underneath, so the user-level threads are just time-slicing. No true multicore speedup here.

- **external_cal:** This one is basically I/O-bound (tons of tiny file reads). Adding more threads mostly adds overhead (more context switches, more contention on reads). Runtime is similar or slightly worse at very high counts.
- **vector_multiply:** Very CPU-heavy. If you crank up the thread count too much, the extra context switching eats into any gains. Best times tend to show up at moderate thread counts.

3.2) pthread vs. my user-level threads

How I switched to pthreads:

1. **In thread-worker.h, commented out #define USE_WORKERS 1.**
2. **Rebuild everything:**
   - **At project root: make clean && make**
   - **Then in benchmarks/: make clean && make && ./genRecord.sh**
3. **Run the same benchmarks again (./external_cal N,./parallelCal N,./vector_multiply N).**

What I noticed

- **I/O-heavy stuff (external_cal):** My user threads don't help much. When one thread is waiting on a file read, the whole process is basically stuck on that single kernel thread. With pthreads, the OS can run another kernel thread while one is blocked, so they usually do as well or better here.

- **Tiny, cooperative tasks:** User-level threads feel snappy. Switching between them is cheap (no kernel context switch), so quick loops that yield or short critical sections run smoothly.

- **CPU-bound on a multicore machine:** Pthreads tend to win. They're real OS threads, so they can run on multiple cores at the same time. My user-level threads all share one kernel thread, so they're just time-slicing, no true parallel speedup.

PART 2

We updated the library to support three distinct, pre-emptive scheduling policies, PSJF, MLFQ, and CFS. The active scheduler is chosen at compile time via the SCHED macro.

All schedulers are driven by the SIGPROF timer. The timer's handler, t_handler, marks the currently running thread as T_PREEMPTED and swaps to the scheduler_context. The main schedule() function then calls the appropriate scheduler-specific function (sched_psjf, sched_mlfq, or sched_cfs) to handle the logic.

We also implemented a global scheduler_enqueue(tcb *t) function, which acts as a wrapper. Based on the active SCHED macro, it routes the TCB to the appropriate data structure: the FIFO run_queue for PSJF, the correct priority queue for MLFQ, or the min-heap for CFS.

2.1: PSJF

This policy implements the shortest time to completion first. Since the total job time is unknown, the scheduler uses the actual time run so far as a proxy. The thread with the minimum run_time_us is always chosen to run next.

Implementation:

1. Accounting: When sched_psjf() is called (either by preemption or yield), it first updates the run_time_us of the curr thread by calculating the time elapsed since last_start_time. If the thread is still ready, it's added back to the single FIFO run_queue
2. Selection: The scheduler iterates through the entire run_queue to find the thread with the minimum run_time_us. Tie-breaking is performed using the thread's t_id to ensure fairness
3. Dispatch: This "shortest" thread is manually removed from the linked list, set as curr, and tot_cntx_switches is incremented
4. Timer: A one-shot timer is set for the global QUANTUM which is defined as 10ms, after which the t_handler will preempt the thread and re-invoke the scheduler

Metrics:

1. Response Time: The first time a thread is scheduled, its first_run_time is recorded and used to update the global avg_resp_time
2. Turnaround Time: When a thread calls worker_exit(), its total turnaround time from creation_time to completion_time is calculated and used to update avg_turn_time

2.2: MLFQ

This policy implements the MLFQ rules. It uses NUM_LEVELS = 4 priority queues, with higher-priority queues having shorter time slices. The time slices are defined as (10, 20, 40, 80) milliseconds.

Implementation:

1. Rule 1 & 2 (Selection): sched_mlfq() always scans the queues from the highest priority (level 0) to the lowest (level 3). It dequeues and runs the first thread it finds from the highest-priority non-empty queue
2. Rule 3 (New Threads): When worker_create() is called, the new TCB is initialized with priority = 0 (due to calloc). scheduler_enqueue() then calls mlfq_enqueue(), placing the new thread in the highest-priority queue (level 0)
3. Rule 4 (Demotion & Yielding):

- Demotion: If a thread is T_PREEMPTED (meaning it used its full time slice), the scheduler increments its priority, demoting it to the next lower queue
- Yielding: If a thread calls worker_yield(), its status is T_READY. It is re-enqueued at its current priority level, not demoted

4. Rule 5 (Priority Boost): A global last_boost_time is maintained. Every PRIORITY_BOOST_S seconds (2 seconds), the scheduler iterates through all lower queues (levels 1-3), dequeues every thread, resets their priority to 0, and re-enqueues them in the highest-priority queue

The timer is set dynamically based on the priority of the thread being scheduled

2.3: CFS

This policy aims to ensure fairness by giving each thread an equal portion of the CPU. It does this by tracking vruntime_us (virtual runtime), which in this project is simply the thread's total accumulated runtime.

Implementation:

1. Data Structure: The runqueue is implemented as a min-heap. This structure uses cfs_heap_insert (sift-up) and cfs_heap_pop_min (sift-down) to efficiently manage TCBs based on their vruntime_us
2. Core Logic:
   - Accounting: When the scheduler runs, it calculates the elapsed_us the curr thread just ran and adds this value directly to curr->vruntime_us. The thread is then re-inserted into the min-heap
   - Selection: sched_cfs() simply calls cfs_heap_pop_min() to select the thread with the smallest vruntime_us from the heap
3. Dynamic Time Slice: The time slice is not fixed. It is calculated dynamically based on two macros:
   - time_slice_us = TARGET_LATENCY (20ms) / num_runnable_threads.
   - The num_runnable_threads is tracked by a global counter, cfs_total_threads, which is incremented in worker_create and decremented in worker_exit
   - This makes sure that as more threads become active, each one gets a smaller slice of the target latency period
4. Minimum Granularity: The calculated time_slice_us is checked against MIN_SCHED_GRN (1ms). If the slice is smaller than this minimum, it is set to MIN_SCHED_GRN to avoid excessive context switching overhead
5. Timer: The preemption timer is set to this dynamically calculated time_slice_us
6. New Threads: New threads are initialized with vruntime_us = 0 (from calloc) and inserted into the min-heap, ensuring they are scheduled promptly