User Mode Thread Scheduler with Dynamic Feedback

Team Members:

Kartik Sarda:	23114047
Nitin Agiwal:	23114074
Darsh Jain:	23114023
Shyam Agarwal:	23116089
Krishna Pahariya:	23113089

Project Repository:

github.com/theoriginalshyam/thread-scheduler

1 Introduction

Efficient CPU scheduling is critical for both operating systems and user-level runtimes, as it directly affects application throughput, latency, and overall fairness. In this project, we present a flexible *User–Mode Thread Scheduler with Dynamic Feedback* that not only implements a novel adaptive scheduling framework but also exposes eight classical scheduling policies as selectable options:

- **First–Come, First–Served (FCFS)**
- **Round Robin (RR)**
- **Priority Scheduling (with and without Aging)**
- **Shortest Job First (SJF)**
- **Multi–Level Queue (MLQ)**
- **Multi–Level Feedback Queue (MLFQ)**
- **Earliest Deadline First (EDF)**
- **Completely Fair Scheduler (CFS)**

Users of our library can *choose* at runtime which of these eight policies to apply to their user-level threads, allowing easy comparison and integration into diverse application scenarios.

Beyond offering these standard algorithms, our scheduler continuously monitors per-thread metrics—CPU utilization, latency to first run, and waiting times—and dynamically adjusts both time quanta and priorities during execution. This dynamic-feedback mechanism aims to:

- Balance fairness and avoid starvation or convoy effects via real-time aging and demotion/promotion of priorities,
- Minimize context-switch overhead by tailoring the quantum length to the observed behavior of each thread.

In addition, we have introduced two C++/POSIX-thread implementations of a simple two-resource deadlock scenario with built-in detection and recovery ("preemptor") using mutexes and semaphores. The rest of this report is divided into two parts:

1. **Survey & Comparison of Classical Algorithms.** For each policy, we present a description, Gantt chart visualization, and performance metrics under a heterogeneous workload.
2. **Design & Implementation of the Dynamic-Feedback ULT Scheduler.** We detail our user-level threading framework built on POSIX `ucontext`, describe synchronization primitives and data structures, walk through each scheduling policy's code, and showcase experimental results that demonstrate the benefits of our dynamic adjustment strategy.

2 Scheduling Algorithms

2.1 First–Come, First–Served (FCFS)

Description

FCFS schedules Taskes in the exact order of their arrival. Once a Task begins execution, it runs to completion without being preempted. This makes FCFS implementation straightforward, but it can suffer from the *convoy effect*, where short tasks wait behind long-running ones, increasing average waiting time.

Use Cases: Simple batch systems where turnaround time is less critical.

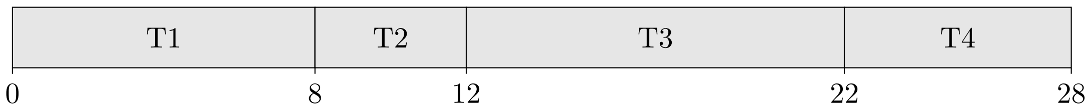
Advantages:

- Easy to implement with minimal overhead.
- Predictable: no context-switching once a task begins.

Disadvantages:

- Poor average waiting time if burst time distribution is skewed.
- No fairness guarantees for interactive or high-priority tasks.

Gantt Chart



Metrics

Task	Start	Completion	Turnaround	Waiting	Response
T1	0	8	8	0	0
T2	8	12	12	8	8
T3	12	22	22	12	12
T4	22	28	28	22	22
Avg.	—	—	17.5	10.5	10.5

2.2 Round Robin (RR), $q = 2$

Description

RR assigns each Task a fixed *time quantum* ($q = 2\,\mathrm{s}$ here). Taskes are placed in a circular queue; each runs for up to one quantum and, if unfinished, returns to the queue's tail. This preemptive approach ensures responsive sharing of CPU time among all tasks.

Use Cases: Time-sharing or interactive systems where fairness and responsiveness matter.

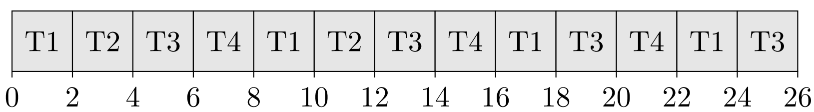
Advantages:

- Bounded response time for all tasks.
- Simple fairness across Taskes.

Disadvantages:

- Excessive context switching if quantum too small.
- Approaches FCFS behavior if quantum too large.

Gantt Chart



Metrics

Task	First Start	Completion	Turnaround	Waiting	Response
T1	0	24	24	16	0
T2	2	12	12	8	2
T3	4	26	26	16	4
T4	6	22	22	16	6
Avg.	—	—	21.0	14.0	3.0

2.3 Priority Scheduling (Non-Preemptive)

Description

Each Task is assigned a static priority; the scheduler always picks the highest-priority task (lowest number) ready to run. In this non-preemptive variant, once a task starts, it runs to completion even if a higher-priority Task arrives.

Use Cases: Systems requiring guaranteed service for critical tasks.

Advantages:

- Allows explicit differentiation between critical and non-critical jobs.

Disadvantages:

- Starvation risk for low-priority tasks.
- Aging or priority adjustments needed to ensure fairness.

2.3.1 Metrics Comparison: Aging vs No Aging

We evaluated both variants on a heterogeneous workload—composed of both short and long jobs with varied priorities—where each task $\mathrm{T_i}(a, b, p)$ has arrival time a, CPU burst b, and static priority p:

$$\{\mathrm{T1}(0,4,5),\ \mathrm{T2}(1,10,1),\ \mathrm{T3}(2,6,4),\ \mathrm{T4}(3,3,3),\ \mathrm{T5}(5,2,2)\}.$$

Variant	**Avg. Waiting**	**Avg. Response**	**Avg. Turnaround**
No Aging	6.2	6.2	11.2
With Aging	7.8	3.8	12.8

Table 1: Non-Aging vs. Aging on a Heterogeneous Workload

Calculation Details

$$\begin{aligned}
\text{Avg. Waiting (No Aging)} &= \tfrac{0+14+2+7+8}{5} = 6.2,\\
\text{Avg. Response (No Aging)} &= \tfrac{0+14+2+7+8}{5} = 6.2,\\
\text{Avg. Turnaround (No Aging)} &= \tfrac{4+24+8+10+10}{5} = 11.2,\\
\text{Avg. Waiting (With Aging)} &= \tfrac{0+11+17+5+6}{5} = 7.8,\\
\text{Avg. Response (With Aging)} &= \tfrac{0+6+2+5+6}{5} = 3.8,\\
\text{Avg. Turnaround (With Aging)} &= \tfrac{4+21+23+8+8}{5} = 12.8.
\end{aligned}$$

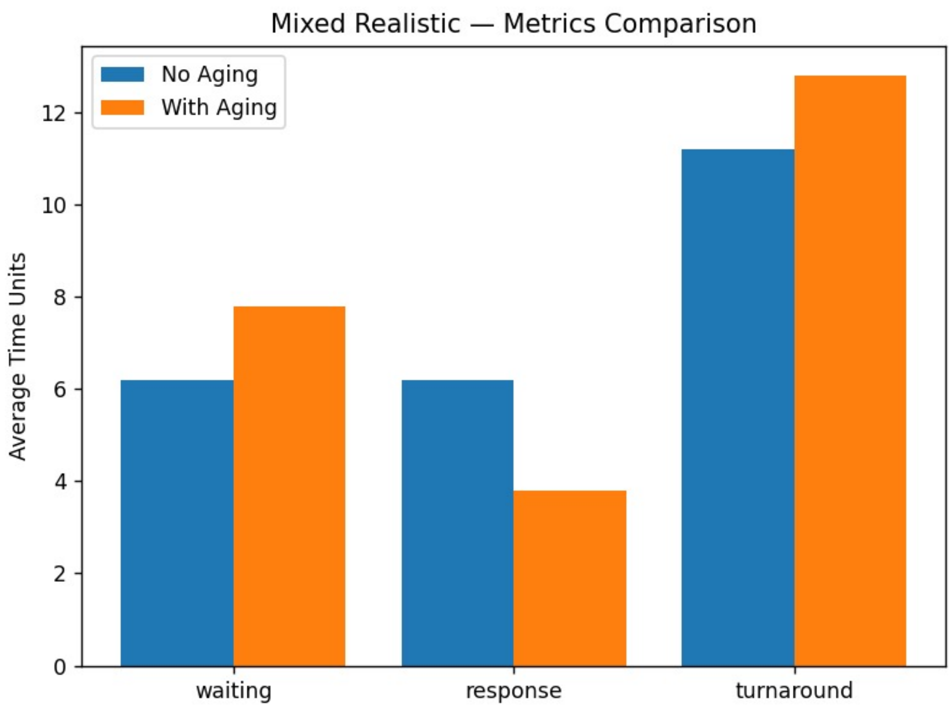


Figure 1: Heterogeneous Workload: Metrics Comparison Bar Chart

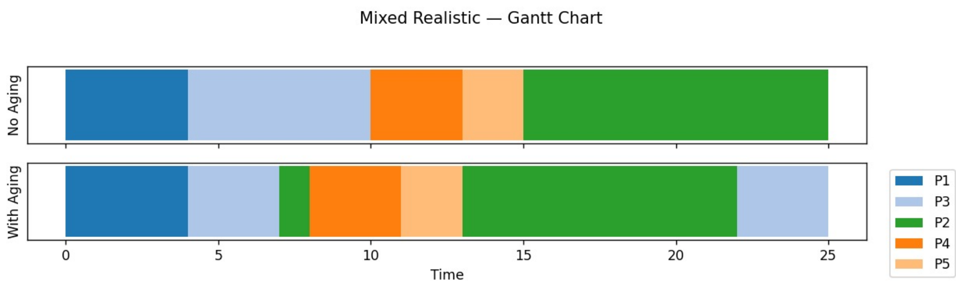


Figure 2: Heterogeneous Workload: Gantt Chart Comparison

2.4 Shortest Job First (SJF)

Description

SJF always selects the Task with the smallest estimated burst time next (non-preemptive). This minimizes average waiting time under the assumption of known burst lengths but may starve long tasks if short ones continuously arrive.

Use Cases: Batch scheduling where job lengths are known in advance.

Advantages:

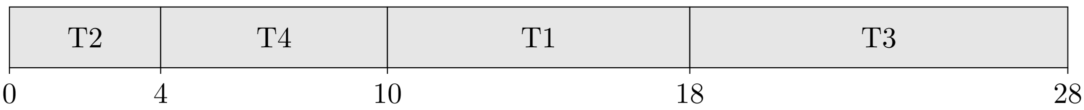
- Optimal for minimizing average waiting time.

Disadvantages:

- Starvation risk for longer jobs.
- Requires accurate burst-time predictions.

Gantt Chart

Order: T2, T4, T1, T3.



Metrics

Task	Start	Completion	Turnaround	Waiting	Response
T2	0	4	4	0	0
T4	4	10	10	4	4
T1	10	18	18	10	10
T3	18	28	28	18	18
Avg.	—	—	15.0	8.0	8.0

2.5 Multi–Level Queue (MLQ)

Description

MLQ partitions Taskes into fixed queues based on type or priority. Each queue uses its own scheduling algorithm, and queues are served in strict priority order. Lower queues may starve if upper queues are always busy.

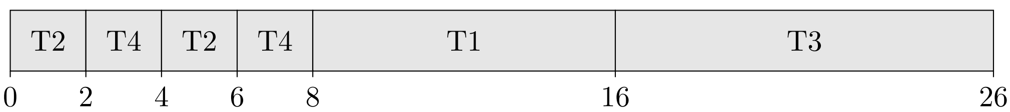
Use Cases: Systems with distinct job classes (e.g., interactive vs. batch).

Disadvantages:

- Static queue assignment can lead to starvation of lower classes.

Gantt Chart

Queue 1 (RR, $q = 2$): T2,T4; Queue 2 (FCFS): T1,T3.



Metrics

Task	Completion	Turnaround	Waiting	Response
T2	6	6	2	0
T4	8	8	2	2
T1	16	16	8	8
T3	26	26	16	16
Avg.	—	14.0	7.0	6.5

2.6 Multi–Level Feedback Queue (MLFQ)

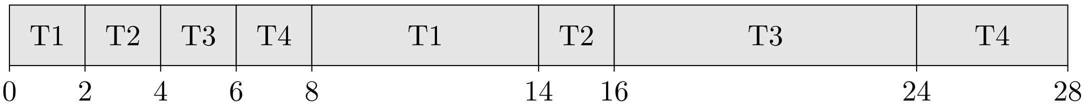
Description

MLFQ extends MLQ by allowing Taskes to move between queues. If a Task uses its quantum fully, it is demoted to a lower-priority queue. Long-waiting Taskes can be promoted to prevent starvation. Tuning queue parameters is essential.

Use Cases: Modern general-purpose OS schedulers seeking balance of responsiveness and fair throughput.

Gantt Chart

Queue 1 (RR, $q = 2$): T1,T2,T3,T4 demoted; then FCFS in Q2.



Data Structures

To implement MLFQ efficiently, we use:

- **Vector of deques** – one deque per priority level.
  - Each deque holds the ready Taskes in that level.
  - Fast push_back and pop_front allow O(1) enqueue/dequeue when demoting or running a job.
- **Array of iterators/indices** – maps each Task ID to its position in its current deque.
  - Enables O(1) removal when a Task is promoted or preempted mid-quantum.
- **Counters or timestamps** – track how long each Task has waited.
  - If wait time exceeds a threshold, the scheduler promotes the Task by moving it to a higher-priority deque.

Metrics

Task	Completion	Turnaround	Waiting	Response
T1	14	14	6	0
T2	16	16	12	2
T3	24	24	14	4
T4	28	28	22	6
Avg.	—	20.5	13.5	3.0

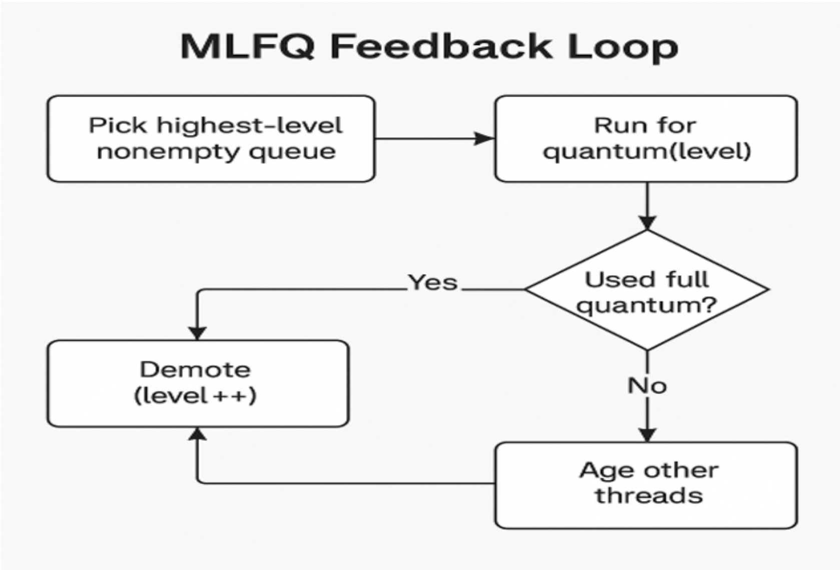


Figure 3: Flowchart explaining the working of MLFQ

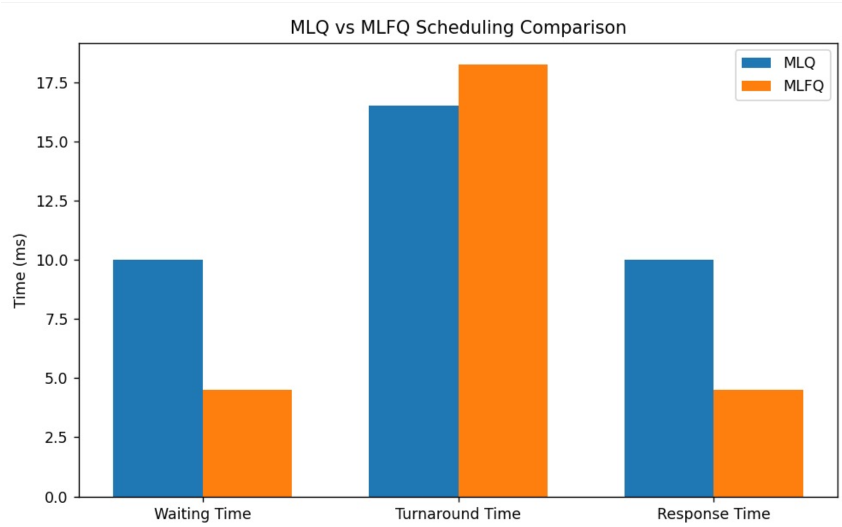


Figure 4: MLQ vs MLFQ Metrics Comparison

2.7 Earliest Deadline First (EDF)

Description

EDF is a preemptive real-time algorithm that always selects the ready task with the earliest deadline. It optimally schedules if CPU utilization 100% but can fail under overload.

Use Cases: Real-time embedded and multimedia systems.

Gantt Chart

Order: T2,T4,T1,T3.

T2	T4	T1	T3
0–4	4–10	10–18	18–28

Metrics

Task	Completion	Turnaround	Waiting	Response
T2	4	4	0	0
T4	10	10	4	4
T1	18	18	10	10
T3	28	28	18	18
Avg.	—	15.0	8.0	8.0

2.8 Completely Fair Scheduler (CFS, $q = 4$)

Description

CFS maintains a *red-black tree* sorted by each task's accumulated *virtual runtime*. On each scheduling decision, it runs the task with the smallest virtual runtime next, ensuring proportional fairness. A time slice of $q = 4$ s is used to approximate equal share among tasks.

Use Cases: General-purpose OS kernels favoring fairness and interactivity.

Advantages:

- Provides fairness without rigid time-slice rotation order.
- Dynamically adapts to Task behavior and priorities.

Disadvantages:

- More complex data structures and bookkeeping overhead.

Data Structures

CFS keeps two core structures:

- **Unordered Map of vruntimes:**
  - `std::unordered_map<Task*, double> vruntime;`
  - Tracks each task's accumulated virtual run time.
- **Red–Black Tree (multiset):**
  - `std::multiset<Task*, decltype(cmp)> rq(cmp);`
  - Tasks are ordered by the comparator 'cmp' which picks the smallest 'vruntime' (tie-breaking on task ID).
  - Insertion, removal, and finding the next task all cost $O(\log n)$.

Dynamic Adjustment of Priorities via vruntime: Each time a task runs for a slice of CPU time, we update:

$$\texttt{vruntime[tk]} \mathrel{+}= \frac{\text{slice}}{\texttt{tk->priority}}.$$

Because lower-priority tasks divide by a larger 'priority' value, they accumulate 'vruntime' more slowly and so get scheduled more often relative to higher-priority ones. On every scheduling decision the tree automatically reorders, ensuring the next task is always the one with the minimal updated 'vruntime'.

Metrics

Task	Completion	Turnaround	Waiting	Response
T1	20	20	12	0
T2	24	24	18	4
T3	28	28	18	8
T4	16	16	10	12
Avg.	—	22.0	14.5	6.0

3 Comparison of Averages

Algorithm	Avg. Turnaround	Avg. Waiting	Avg. Response
FCFS	17.5	10.5	10.5
RR ($q = 2$)	21.0	14.0	3.0
Priority	15.5	8.5	8.5
SJF	15.0	8.0	8.0
MLQ	14.0	7.0	6.5
MLFQ	20.5	13.5	3.0
EDF	15.0	8.0	8.0
CFS ($q = 4$)	22.0	14.5	6.0

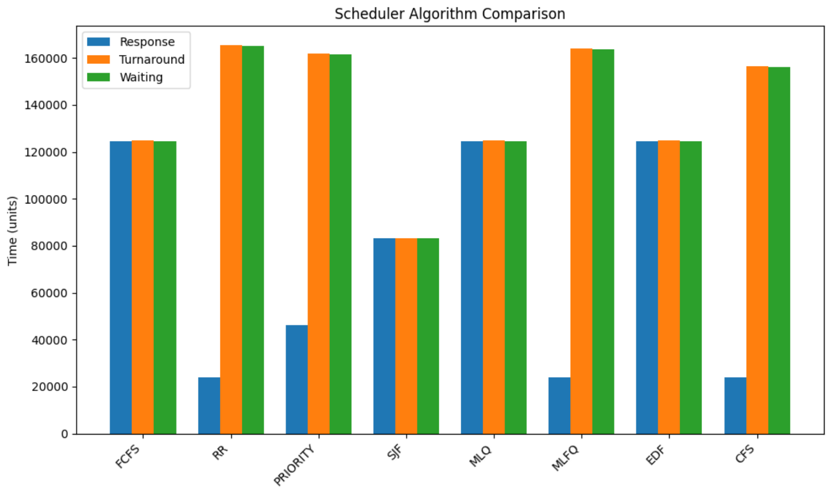


Figure 5: Comparison of different scheduling algorithms

4 User-Level Thread Scheduler Implementation

This section walks through our ULT-based scheduler in `threadedscheduler.cpp`. It covers how we set up multiple "threads" (contexts) in user space, synchronize them with a mutex and condition variable, and implement three classic scheduling policies (FCFS, RR, Priority). Paste each code snippet into the corresponding block.

4.1 File Header & Includes (Lines 1–7)

At the top, we bring in all required libraries:

```
#include "threadedscheduler.h"  // Scheduler class and task definitions
#include "ult_sync.h"          // Our ULTMutex and ULTCondVar
#include <ucontext.h>           // getcontext(), makecontext(), swapcontext()
#include <vector>               // std::vector for context/task lists
#include <algorithm>            // std::sort, std::min
#include <iostream>             // std::cout for logging
#include <time.h>               // nanosleep for simulating work
```

We include all necessary headers so our ULT scheduler has:

- **threadedscheduler.h**: Declares the `ThreadedScheduler` class, task structures, and algorithm enums.
- **ult_sync.h**: Defines `ULTMutex` and `ULTCondVar`, our user-level locking primitives.
- **ucontext.h**: Provides `getcontext`, `makecontext`, and `swapcontext` for manual context switching.
- **vector**: Used to hold lists of tasks and ULT contexts.
- **algorithm**: Supplies `std::sort` and `std::min` for scheduling logic.
- **iostream**: Allows printing log messages to `stdout`.
- **time.h**: Gives `nanosleep` to simulate work delays.

These imports let us entirely manage "threads" in user space, build dynamic arrays, sort for priority, and simulate execution without touching kernel APIs.

4.2 Synchronization Primitives (Lines 8–13)

We declare global sync objects and shared data:

```
static ULTMutex shared_mtx;     // ensures one ULT at a time in critical section
static ULTCondVar shared_cv;    // lets ULTs wait for/data_ready event
static bool data_ready = false; // signaled by ULT 0 to wake others
static int  shared_counter = 0;// updated inside protected critical section
```

We declare globals to coordinate ULTs:

- `shared_mtx`: A `ULTMutex` ensuring that only one ULT can modify `shared_counter` at a time.
- `shared_cv`: A `ULTCondVar` that ULTs can `wait()` on or be `broadcast()` to when `data_ready` changes.
- `data_ready`: A simple boolean flag; ULT 0 sets this to true and wakes waiting ULTs.
- `shared_counter`: A simulated shared resource, incremented inside a protected critical section.

Use `lock()/unlock()` around any access to `shared_counter`, and `wait()/broadcast()` on `shared_cv` to implement event-based synchronization fully in user space.

4.3 ULT Context Structure (Lines 14–19)

Each ULT has its own stack and a flag:

```
struct ULTContext {
    ucontext_t ctx;               // CPU state + stack pointer
    char       stack[64*1024];    // private 64KiB stack space
    bool       finished = false;  // becomes true when thread completes
};
```

Each user-level thread (ULT) needs:

- `ucontext_t ctx`: Stores CPU registers, stack pointer, and execution state.
- `char stack[64*1024]`: A private 64 KiB stack for that ULT's function calls.
- `bool finished`: Marks when the ULT has completed all its work, so the scheduler can stop rescheduling it.

By assigning `ctx.uc_stack.ss_sp` and `ss_size`, each ULT has its own call stack, and the `finished` flag tells the trampoline when to exit.

4.4 Scheduler State (Lines 20–25)

Global variables for managing contexts:

```
static ucontext_t               sched_ctx;     // context for scheduler itself
static ThreadedScheduler*       g_sched_ptr;   // pointer to scheduler object
static std::vector<ULTContext>  g_contexts;    // array of all ULT contexts
static size_t                   g_current_idx; // index of the running ULT
```

We maintain global variables to manage context switching:

- `sched_ctx`: A `ucontext_t` capturing the scheduler's own stack and registers – where ULTs return when yielding.
- `g_sched_ptr`: A pointer to the active `ThreadedScheduler` instance, so `task_trampoline` can access the `tasks` vector.
- `g_contexts`: A `std::vector<ULTContext>` holding each ULT's context and stack buffer.
- `g_current_idx`: The index of the ULT that is currently executing, used by synchronization primitives to enqueue/wake the correct context.

These globals glue together the scheduler logic, the per-ULT contexts, and the sync mechanisms.

4.5 Task Trampoline (Lines 26–68)

All ULTs start execution here via `makecontext`:

```
template<typename Func>
static void task_trampoline(uintptr_t idx) {
    ULTContext& c = g_contexts[idx];
    auto& tk    = g_sched_ptr->tasks[idx];
    // 1) Yield immediately to scheduler before doing anything
    swapcontext(&c.ctx, &sched_ctx);

    // 2) Condition-variable demo:
```

```
    if (idx == 0) {
        // ULT 0 signals that data is ready
        shared_mtx.lock();
        data_ready = true;
        shared_cv.broadcast();
        shared_mtx.unlock();
    } else {
        // Others wait until data_ready becomes true
        shared_mtx.lock();
        while (!data_ready) {
            shared_cv.wait(shared_mtx);
        }
        shared_mtx.unlock();
    }

    // 3) Main work loop, runs until 'finished' is set
    while (!c.finished) {
        std::cout << "[Thread " << tk.id
                  << "] running slice on task " << tk.id << "\n";

        // Critical section: update shared_counter
        shared_mtx.lock();
        shared_counter += 1;
        shared_mtx.unlock();

        // Simulate work by sleeping 30ms
        struct timespec ts{0, 30 * 1000000};
        nanosleep(&ts, nullptr);

        std::cout << "[Thread " << tk.id << "] done slice\n";
        // Yield back to scheduler for next slice
        swapcontext(&c.ctx, &sched_ctx);
    }

    // 4) Final cleanup before exiting
    std::cout << "[Thread " << tk.id << "] exiting\n";
    swapcontext(&c.ctx, &sched_ctx);
}
```

Every ULT begins execution in `task_trampoline(idx)`, which:

1. Immediately yields to the scheduler (`swapcontext`), so the scheduler can record timeline entries or adjust priorities before any work.
2. Demonstrates a condition-variable handshake:
   - ULT 0 locks the mutex, sets `data_ready = true`, broadcasts to all waiting contexts, then unlocks.
   - Other ULTs lock, then spin-wait on `data_ready` via `shared_cv.wait()`, automatically unlocking while parked and re-locking when resumed.
3. Enters a `while(!finished)` loop where it:
   - Logs that it is running.

- Locks shared_mtx, increments shared_counter, unlocks.
- Sleeps for 30 ms to simulate actual work.
- Logs completion of that slice.
- Yields back to the scheduler via swapcontext.

4. Upon finished == true, prints an exit message and returns to the scheduler one last time.

This trampoline neatly encapsulates initialization, synchronization, work, and termination all in one place, illustrating cooperative multitasking entirely in user space.

4.6 Context Setup (Lines 69–84)

Before running any ULTs, we must initialize their contexts:

```
static void setup_contexts(ThreadedScheduler* sched) {
    g_sched_ptr = sched;
    size_t n    = sched->tasks.size();
    g_contexts.clear();
    g_contexts.resize(n);
    for (size_t i = 0; i < n; ++i) {
        ULTContext& c = g_contexts[i];
        c.finished  = false;
        getcontext(&c.ctx);                       // capture current state
        c.ctx.uc_stack.ss_sp   = c.stack;
        c.ctx.uc_stack.ss_size = sizeof(c.stack);
        c.ctx.uc_link         = &sched_ctx;      // return here when done
        makecontext(&c.ctx, (void(*)())task_trampoline<void>, 1, (uintptr_t)i);
    }
}
```

Before any ULT runs, setup_contexts(sched) does:

- Stores the scheduler pointer in g_sched_ptr for later access.
- Resizes g_contexts to match the number of tasks.
- For each index i:
  1. Calls getcontext(&c.ctx) to capture the current Task state into that new context.
  2. Sets c.ctx.uc_stack.ss_sp to c.stack and ss_size to the stack's size.
  3. Assigns uc_link = &sched_ctx so that when the ULT function returns, control goes back to the scheduler.
  4. Uses makecontext to arrange for task_trampoline(i) to run when this context is activated.

This establishes each ULT's private execution context and ties its return path back to the scheduler.

4.7 Running a ULT Slice (Lines 85–89)

An inline helper switches into a chosen ULT:

```
inline void run_ult_slice(size_t idx, int /*run*/) {
    g_current_idx = idx;                          // for sync primitives
    swapcontext(&sched_ctx, &g_contexts[idx].ctx);
}
```

The helper `run_ult_slice(idx, run)` performs a context switch:

- Sets `g_current_idx = idx` so the sync primitives know which ULT is active.
- Calls `swapcontext(&sched_ctx, &g_contexts[idx].ctx)`:
  - Saves the scheduler's registers/stack into `sched_ctx`.
  - Restores the ULT's saved registers/stack from `g_contexts[idx].ctx`, resuming its execution at the last yield point.

When the ULT later does its own `swapcontext` back to `sched_ctx`, execution resumes right after this call, enabling a seamless cooperative scheduling loop.

4.8 Scheduler Entry & Dispatch (Lines 90–99)

The `ThreadedScheduler::run()` method sets up contexts and calls the chosen policy:

```
void ThreadedScheduler::run() {
    setup_contexts(this);    // Prepare all ULT contexts
    getcontext(&sched_ctx);  // Capture scheduler state
    switch (algorithm) {
      case T_FCFS:    runFCFS();    break;
      case T_RR:      runRR();      break;
      case T_PRIORITY:runPriority();break;
    }
}
```

When the user invokes `ThreadedScheduler::run()`, three things happen in order:

1. **Context Initialization:** The call to `setup_contexts(this)` allocates a separate `ucontext_t` structure and stack buffer for each "thread" (ULT) and stashes them in the global `g_contexts` vector.

2. **Scheduler Snapshot:** By executing `getcontext(&sched_ctx)`, the scheduler captures its own registers, stack pointer, and instruction pointer. Every ULT can later *yield* back to this exact saved state.

3. **Policy Selection:** A simple `switch` on the `algorithm` field transfers control into one of three loops (`runFCFS`, `runRR`, or `runPriority`). Each of those loops will call `run_ult_slice()` to actually swap into a particular ULT's context for its allotted time slice.

Why this matters: Capturing the scheduler's context lets ULTs cooperatively volunteer when they want to pause (via `swapcontext`), and the policy switch cleanly separates "which algorithm are we running?" from "how do we execute one slice?"

4.9 Scheduling Policies

Below are the three policies. Each uses `run_ult_slice` to give tasks slices of CPU time and records a timeline entry. Detailed descriptions follow each snippet.

4.9.1 First-Come, First-Served (Lines 100–115)

```
void ThreadedScheduler::runFCFS() {
    log("[T-FCFS] Starting");
    int t = 0;
    for (size_t i = 0; i < tasks.size(); ++i) {
        auto& tk = tasks[i];
        int s = t;
        int e = s + tk->remaining_time;
        _timeline.push_back({tk->id, s, e});
        log("[T-FCFS] Task " + std::to_string(tk->id)
            + " " + std::to_string(s) + "->" + std::to_string(e));
        run_ult_slice(i, tk->remaining_time);
        t = e;
        g_contexts[i].finished = true;
        swapcontext(&sched_ctx, &g_contexts[i].ctx);
    }
    log("[T-FCFS] Done");
}
```

In FCFS scheduling:

- We keep a single "time" variable `t` that starts at zero.
- We iterate over *each* ULT in the order they were created.
- For ULT `i`, we:
    1. Record its *start time* `s = t` and its *end time* `e = s + remaining_time`.
    2. Append the tuple {`id, s, e`} to the timeline and log it, so we can visualize when the ULT ran.
    3. Call `run_ult_slice(i, remaining_time)`, which switches into that ULT and lets it run until it finishes all its work.
    4. Update the clock `t = e`, mark that ULT's context as `finished=true`, and swap back to the scheduler.
- Once every ULT has run, we log "Done" and exit.

4.9.2 Round-Robin (Lines 116–140)

```
void ThreadedScheduler::runRR() {
    log("[T-RR] Starting");
    int t = 0;
    bool work_left = true;
    while (work_left) {
        work_left = false;
        for (size_t i = 0; i < tasks.size(); ++i) {
            auto& tk = tasks[i];
            if (tk->remaining_time > 0) {
```

```
                work_left = true;
                int run = std::min(tk->remaining_time, time_quantum);
                int s = t;
                int e = s + run;
                _timeline.push_back({tk->id, s, e});
                log("[T-RR] Task " + std::to_string(tk->id)
                    + " " + std::to_string(s) + "->" + std::to_string(e));
                run_ult_slice(i, run);
                tk->remaining_time -= run;
                t = e;
                if (tk->remaining_time <= 0) {
                    g_contexts[i].finished = true;
                    swapcontext(&sched_ctx, &g_contexts[i].ctx);
                }
            }
        }
    }
    log("[T-RR] Done");
}
```

Round-Robin divides CPU time into fixed quanta to share fairly:

- Initialize `t = 0` and a boolean `work_left = true`.
- Enter an outer *while* loop that repeats as long as any ULT has `remaining_time > 0`.
- Inside, we scan every ULT in a *for* loop:
  1. If its `remaining_time` is positive, we compute `run = min(remaining_time, time_quantum)`.
  2. We record `s = t` and `e = s + run`, push {`id, s, e`} to the timeline, and log it.
  3. We call `run_ult_slice(i, run)`, which resumes that ULT for exactly `run` units.
  4. After returning, we subtract `run` from its remaining time and set `t = e`.
  5. If the ULT finishes (`remaining_time <= 0`), we mark it `finished` and immediately swap back.
- The loops continue until *all* ULTs report no work left.
- Finally, log “Done” and return.

4.9.3 Priority Scheduling (Lines 141–175)

```
void ThreadedScheduler::runPriority() {
    log("[T-PRIORITY] Starting");
    int t = 0;
    const int FF = 50, AG = 1; // Feedback factor and aging increment
    size_t done = 0;
    while (done < tasks.size()) {
        std::vector<size_t> order;
        for (size_t i = 0; i < tasks.size(); ++i)
            if (!g_contexts[i].finished) order.push_back(i);
        // Sort by current priority (higher value = higher priority)
        std::sort(order.begin(), order.end(), [&](size_t a, size_t b) {
            return tasks[a]->priority > tasks[b]->priority;
```

```
        });
        size_t idx = order.front();
        auto& tk = tasks[idx];
        int run = std::min(tk->remaining_time, time_quantum);
        int s = t;
        int e = s + run;
        _timeline.push_back({tk->id, s, e});
        log("[T-PRIORITY] Task " + std::to_string(tk->id)
            + " pr=" + std::to_string(tk->priority));
        run_ult_slice(idx, run);
        tk->remaining_time -= run;
        t = e;
        // Adjust priorities: demote current, age others
        tk->priority = std::max(1, tk->priority - run/FF);
        for (auto j : order) if (j != idx) tasks[j]->priority += AG;
        if (tk->remaining_time <= 0) {
            g_contexts[idx].finished = true;
            swapcontext(&sched_ctx, &g_contexts[idx].ctx);
            ++done;
        }
    }
    log("[T-PRIORITY] Done");
}
```

Priority scheduling dynamically orders ULTs each round:

- We track `t = 0`, a `done` counter, and constants `FF` (feedback divisor) and `AG` (aging increment).
- Repeat while `done < total_ULTs`:
  1. Gather indices of all ULTs not yet marked `finished`.
  2. Sort them by their current `priority` (highest first).
  3. Pick the top index `idx`, compute its slice length `run = min(remaining_time, time_quantum)`, record `s` and `e`, push to the timeline, and log the priority.
  4. Switch into that ULT via `run_ult_slice(idx, run)`; then subtract `run` from its `remaining_time` and set `t = e`.
  5. Adjust priorities:
     - *Demote* the just–run ULT by dividing its run length by `FF`.
     - *Age* all other ready ULTs by adding `AG` to each.
  6. If this ULT finishes, mark it `finished` and swap back immediately; increment `done`.
- Log "Done" when every ULT has completed.

article amsmath graphicx enumitem booktabs

5 Deadlock Detection and Recovery: Mutex vs Semaphore Approach

5.1 Common Structure

Both versions share these components:

- Two resources (`mutex1+mutex2` or `sem1+sem2`) that threads T1 and T2 try to acquire in opposite orders, leading to classic deadlock risk.
- A deadlock detector thread that maintains a wait-for graph (protected by `graphMutex`) and signals when a cycle is found.
- A preemptor thread that, upon detecting deadlock (simulated by a timed sleep), forces one resource to be released so the other thread can proceed, then resumes the paused thread.
- **Thread1** acquires resource A, then loops trying to acquire resource B—but will pause itself and let the preemptor free resource A.
- **Thread2** simply acquires B then A, works, and releases both.

5.2 Deadlock Detection

Identical in both versions:

- Data structures:
  - `waiting_for[thread]` = resource when a thread blocks.
  - `owner_of[resource]` = thread when a thread acquires the resource.
- Every 2 seconds, build a directed graph from waiting → owner edges and search for cycles via DFS (Depth-First Search).

5.3 Preemption and Recovery

Aspect	Mutex Version	Se
Pause signaling	T1 checks `paused_thread1` under `controlMutex` conditionally	Sa
Releasing resource A	Preemptor must call `unlock_mutex(&mutex1)` externally (undefined behavior)	Pr
Ownership bookkeeping	`mutex_owner.erase(&mutex1)` then `pthread_mutex_unlock(&mutex1)`	se
Resuming T1	`pthread_cond_signal(&cv1)`	se

Key Difference:

- **Mutexes** enforce ownership: unlocking a mutex by a thread that did not lock it is undefined behavior (UB) and may crash or deadlock the program.
- **Semaphores** do not: `sem_post()` can be invoked by any thread or process, thus allowing an external "watchdog" to inject tokens and break deadlocks safely.

5.4 Code-Level Comparison

5.4.1 Lock/Unlock Wrappers

Mutex:

```
void lock_mutex(pthread_mutex_t* m) {
    // record in waiting_for, pthread_mutex_lock(m),
    // record in owner, erase waiting
}
```

```
void unlock_mutex(...) {
    erase owner;
    pthread_mutex_unlock(m);
}
```

Semaphore:

```
void lock_sem(sem_t* s) {
    // record waiting_for, sem_wait(s),
    // record owner, erase waiting
}
bool trylock_sem(sem_t* s) {
    sem_trywait + bookkeeping
}
void unlock_sem(...) {
    erase owner;
    sem_post(s);
}
```

5.5 Preemptor Behavior

- **Mutex:** forcibly unlocks both `mutex1` and `mutex2` via `unlock_mutex()`. This is inherently unsafe with pthreads.
- **Semaphore:** forcibly "unlocks" `sem1` via `sem_post(&sem1)` without prior `sem_wait()`, demonstrating how semaphores allow non-owner releases.

5.6 Pros and Cons

Criterion	Mutex + CondVar	Semaphore
Safety	Preemptor bypasses ownership rules → UB	`sem_post` from any thread is safe
Simplicity	Two primitives (mutex + condvar)	Single primitive + atomic flag
Inter-process	Mutexes usually limited to threads	Named semaphores can span processes
Predictability	UB when unlocking from wrong thread	Well-defined incrementing counter

5.7 Conclusion

Both implementations successfully detect and break the simulated deadlock. However:

- The semaphore version is **safer** for external preemption: semaphores allow any thread or process to `post()`, making watchdog/preemptor patterns reliable.
- The mutex version's forced unlock is **undefined behavior** in POSIX threads and should be avoided.

Recommendation: For scenarios where an external agent (thread or process) must forcibly release a resource, use `sem_t` (POSIX semaphores) rather than `pthread_mutex_t`.

This document details the `analyzeAlgorithms()` function in `analysis.cpp`, which benchmarks response time, turnaround time, and waiting time across several scheduling policies. We use a consistent task set and the `Scheduler` class.

6 Key Data Structures

originalTasks
: std::vector<Task> of size 1000 holding all generated tasks.

Metrics
: struct Metrics { std::string name; double resp, tat, wait; } stores per-algorithm metrics.

firstStart, completion
: std::map<int,int> mapping task ID to timestamps.

all
: std::vector<Metrics> accumulates results for CSV output and plotting.

7 Conclusion

This modular design cleanly separates each step—generation, scheduling, measurement, and reporting—allowing straightforward extension to new algorithms or metrics. The snippets above highlight key operations without overwhelming detail.