Assignment 1

Assignment 1: `mycoroutine` API

```
int mycoroutine_init(co_t *main);
int mycoroutine_create(co_t *co, void(body)(void *), void *arg);
int mycoroutine_switchto(co_t *co);
int mycoroutine_destroy(co_t *co);
```

Assignment 1: Initializing and destroying coroutines

```
int mycoroutine_init(co_t *main) {
  initialize coroutine list
  add main to coroutine list

  set current coroutine to main
  set main coroutine to main
}
```

```
int mycoroutine_destroy(co_t *co) {
  if co is not in the coroutine list
    return error

  if co is the main coroutine
    destroy the coroutine list
  else
    remove the co from the list
    free related memory
}
```

Assignment 1: Initializing the main coroutine and global variables

```
static list_t *coroutine_list;
static co_t *current_context, *main_context;

int mycoroutine_init(co_t *main) {
  initialize coroutine list
  add main to coroutine list

  current_context = main
  main_context = main
}
```

Assignment 1: Creating and destroying coroutines

```
int mycoroutine_create(co_t *co, void (body)(void *), void *arg) {
  if co is already on the coroutine list
    return error
  add co to the coroutine list

  getcontext(co)
  allocate memory for the stack
  makecontext(co, body, arg)
}
```

```
int mycoroutine_destroy(co_t *co) {
  if co is not in the coroutine list
    return error

  if co is the main coroutine
    destroy the coroutine list
  else
    remove the co from the list
    free related memory
}
```

Assignment 1: Switching coroutines

```
int mycoroutine_switchto(co_t *co) {
  if co is not in the coroutine list
    return error

  previous_context = current_context
  current_context = co
  swapcontext(previous_context, current_context)
}
```

Assignment 1: Testing coroutines

- In order to test our coroutines implementation we modified our implementation of assignment 1.1 . We removed busy waiting and added manual context switching between the writing and reading thread each time the pipe is full or empty respectively.
- After the 2 threads are done copying the context is switched back to the main context which destroys the coroutines and exits.

Assignment 1: `pipe_write` function

```
int pipe_write(unsigned int p, char c, co_t *read_thread) {
    if pipe does not exist or is not open for writing
        return error

    if pipe's buffer is full
        mycoroutine_switchto(read_thread)

    add character to buffer
    update write pointer
    return 1
}
```

Assignment 1: `pipe_read` function

```
int pipe_read(unsigned int p, char *c, co_t *write_thread) {
    if pipe does not exist
        return error

    if the pipe is open for writing and empty
        mycoroutine_switchto(write_thread)

    if the pipe is closed for writing and empty
        pipe_remove(p)
        return 0

    set c to the next value of the buffer
    update the read pointer
    return 1
}
```

Assignment 2

Assignment 2: mythreads API

```
int mythreads_init();
int mythreads_create(mythr_t *thr, void(body)(void *), void *arg);
int mythreads_yield();
int mythreads_sleep(int secs);
int mythreads_join(mythr_t *thr);
int mythreads_destroy(mythr_t *thr);
int mythreads_sem_create(mysem_t *s, int val);
int mythreads_sem_down(mysem_t *s);
int mythreads_sem_up(mysem_t *s);
int mythreads_sem_destroy(mysem_t *s);
void mythreads_exit();
```

Assignment 2: mythreads API

```c
typedef struct thread_runnable {
    void (*body) (void *);
    void *arg;
} thread_runnable_t;

enum thread_state {
    READY,
    SLEEPING,
    BLOCKED,
    TERMINATED
};
```

```c
typedef struct mythread {
    thread_runnable_t runnable;
    co_t co;
    enum thread_state state;
    unsigned long long sleep_until;
    struct mythread *joining_on;
} mythr_t;
```

Assignment 2: `mythreads` internal functions

Scheduler related functions:

```
void run_scheduler(enum thread_state state);
void reload_states(enum thread_state state);
mythr_t *search_threads(bool running_thread_sleeping, mythr_t **earliest_thread);
void switch_thread(enum thread_state running_next_state, mythr_t *next_thread_node);
```

Utility functions:

```
void run_thread(void *arg);
void alarm_op(enum alarm_op operation, struct itimerval *previous);
void sleep_until(unsigned long long timestamp);
void thread_timeout_handler(int signum);
```

Assignment 2: Avoiding race conditions

When certain functions of the library are called by different threads race conditions may occur (i.e. the scheduler function). In this case race conditions exist only because of the automatic switching that is controlled by an alarm. To prevent this, all functions that can cause a race condition run without consuming time from the alarm. This resembles the OS behavior when running atomic operations. After the operation has been completed the alarm is reactivated with the user's remaining time.

When switching context between different threads an interruption by the alarm could cause unexpected behavior and that's why before swapping the context the alarm is disabled. The new context has the responsibility to reenable the alarm and this is achieved by the functions described later.

Assignment 2: Sleeping mechanism

Our sleep mechanism relies on timestamps that denote when a thread is in the SLEEPING state. Threads marked as SLEEPING are excluded from CPU scheduling until their state changes. The scheduler is responsible for updating the states of these threads.

If all active threads are in the SLEEPING state, the scheduler invokes a system sleep call. This sleep call lasts for the minimum amount needed for the next thread to wake up.

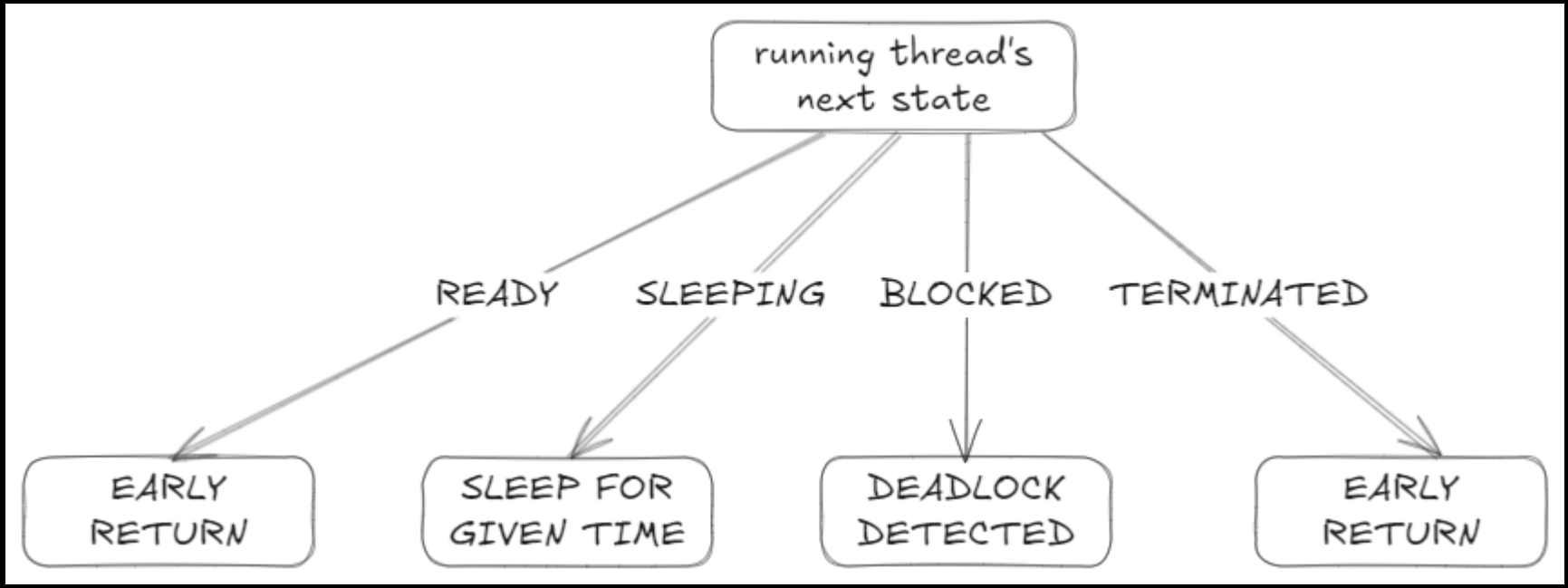
Assignment 2: The scheduler

The scheduler’s job is to determine if switching to another thread is needed based on the state of the previously running thread and finding which thread needs to run after. The logic of the scheduler is broken into 2 parts:

1. Only one thread currently exists
2. More than one threads currently exist

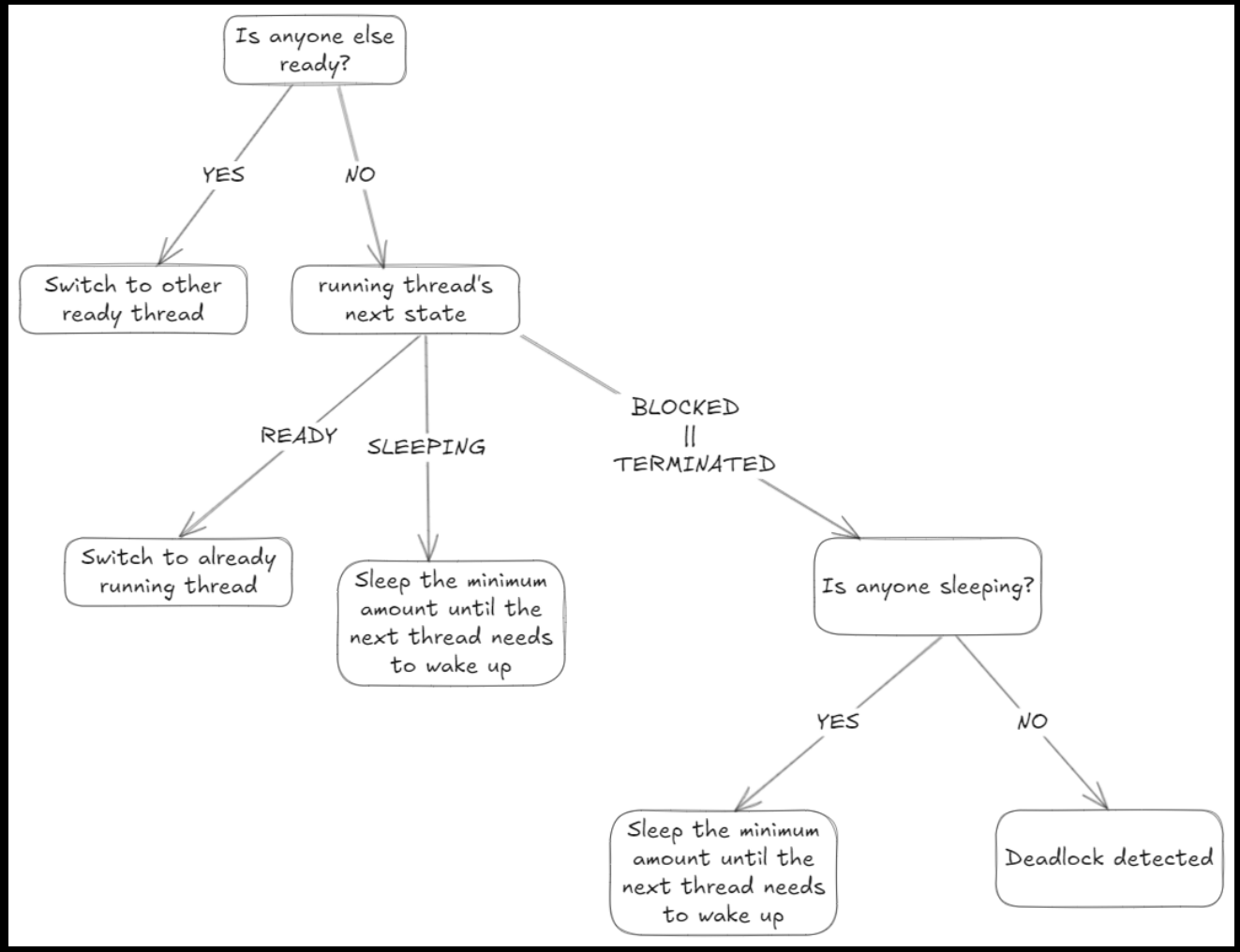
Assignment 2: The scheduler

On the first case when only one thread exists the logic of the scheduler is shown below



Assignment 2: The scheduler

When multiple threads exist the logic of the scheduler is the following



Assignment 2: The scheduler

This is the first case when only one thread exists

```
void run_scheduler(enum thread_state state) {
    disable alarm

    if only one thread exists
        switch (state)
            case TERMINATED:
            case READY:
                return
            case BLOCKED:
                exit because deadlock has been detected
            case SLEEPING:
                sleep until the given timestamp
        return
```

Assignment 2: The scheduler

This is the general case when multiple threads exist

```
    reload states
    find any ready thread
    find the thread that needs to wake up the earliest

    if there is a ready thread
        switch to ready thread
        return

    switch (state)
        case READY:
            reset alarm
            return
        case SLEEPING:
            sleep until the next thread needs to wake up
            switch to next thread
            return
        case TERMINATED:
        case BLOCKED:
            if no thread is sleeping
                exit because deadlock has been detected

            sleep until the next thread needs to wake up
            switch to next thread
}
```

Assignment 2: Storing threads

In the initial iteration of the library design, all threads were stored in an array. This array stored each thread as an entry, which the scheduler iterated through to identify the next thread to assign to the CPU.

This caused a problem when removing threads where we needed to move the last element to the index of the removed thread. But each time a thread is removed we essentially change the order (and priority) of all the threads. This could lead to many unexpected behaviours and can even be exploited by users in order to give certain threads better priority than other. To solve this we switched from using arrays to linked lists.

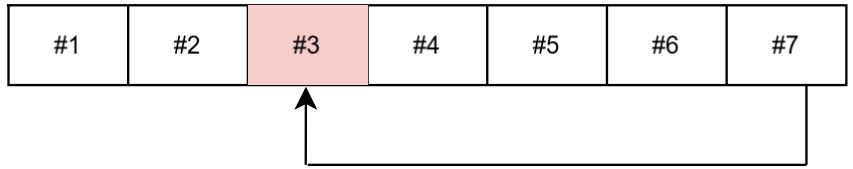
Assignment 2: Storing threads

Step #1



Malicious user creates a thread that will execute a small snippet of code and create a new thread with the same principle as this one, then it exits before the timeout

Step #2



Scheduler replaces the terminated thread with the last in the array, making the newly created thread the Running thread

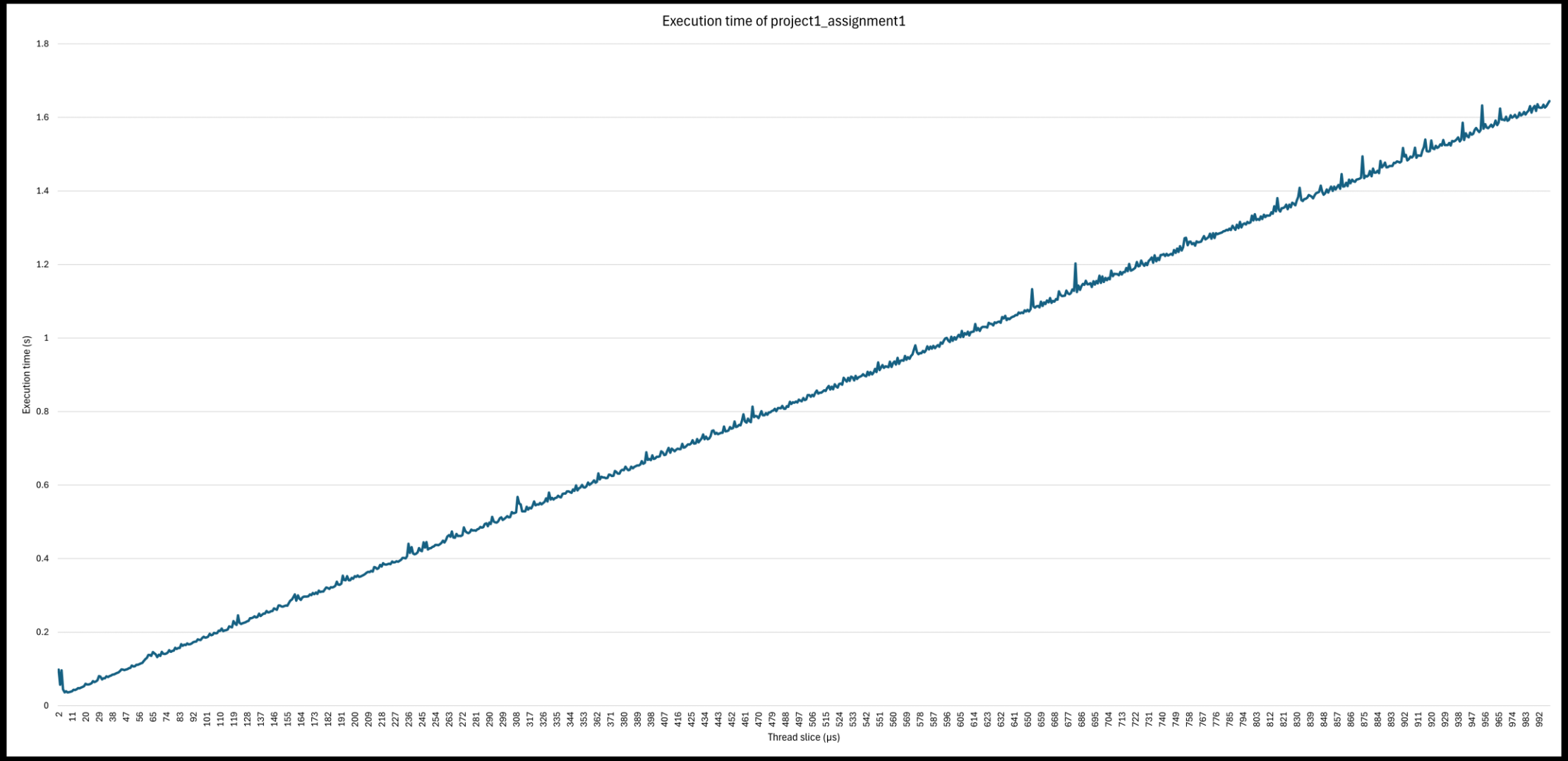
Step #3

Repeat

Assignment 2: Thread slice

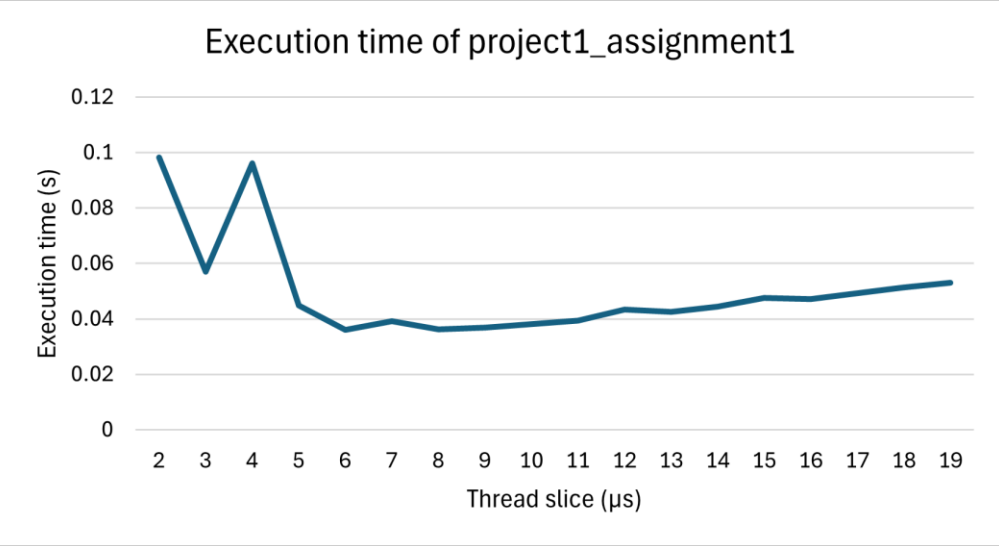
After we got everything working we wanted to find out what the best time slice is. We used the assignment 1 of project 1 and we ran some tests with a range of slices in order to determine the best option. The runtime of this project with an input of some kilobytes turns out to be a linear function of the time slice. This is visible in the following graph:

Assignment 2: Thread slice



Assignment 2: Thread slice

An interesting part of this graph is at the start of the axis where the slice becomes really small. While we would expect a lower time slice to give us better execution time instead the execution rises really quickly. This is because in such small time slices even the quickest atomic system calls may not have enough time to run so many threads cannot make progress in every slice they are given.



When the slice is 1µs the execution time becomes ~70 seconds which is equivalent to the performance we would get if we set the time slice to 40.000µs (40ms)!

Assignment 2: Scheduler related functions

```
void reload_states(enum thread_state_t running_next_state) {
    node_t *running_node = list_find(thrs, running_thr, NULL);

    for all threads starting from the currently running thread
        sleep_expired = thread is sleeping
                        and sleep timestamp has expired
        joining_thread_terminated = thread is blocked
                                    and the thread that it is joining on has terminated

        if sleep_expired or joining_thread_terminated
            set thread state to READY
}
```

This function is used to refresh the states of all the threads so that they accurately describe the thread. It updates threads that were sleeping but their sleep time has expired and threads that were joining on another thread and that thread has been terminated.

Assignment 2: Scheduler related functions

```
mythr_t *search_threads(bool running_thread_sleeping, mythr_t **earliest_thread) {
    mythr_t *next_ready = NULL
    mythr_t *earliest = running_thread_sleeping ? running_thr : NULL

    for all threads starting from the currently running thread
        if thread is READY
            next_ready = thread
            break
        else if thread is SLEEPING and is the earliest thread to wake up
            earliest = thread

    *earliest_thread = earliest
    return next_ready
}
```

This function finds the first ready thread after the currently running thread (if there is any)
and finds the thread that is sleeping and needs to wake up the earliest of any other thread.
It is called by the scheduler to find out which thread should be the next to run.

Assignment 2: Scheduler related functions

```
void switch_thread(enum thread_state_t running_next_state, mythr_t *next_thread) {
    set currently running thread state to running_next_state
    set currently running thread to next thread

    set next_thread state to READY
    mycoroutine_switchto(&next_thread->co)
    alarm_op(RESET, NULL)
}
```

This function performs the context switching between the threads. At first it might seem like the line after the coroutine switch will never run but as explained previously this is done so that there is no race condition when switching threads. This line along with the run_thread function ensures that any thread that is resumed resets the alarm for itself.

Assignment 2: Utility functions

```
void run_thread(void *arg) {
    void (*body) (void *) = get body of thread from arg
    void *body_arg = get arg of thread from arg

    reset alarm
    body(body_arg)
    run_scheduler(TERMINATED)
}
```

This function is a wrapper of the function that is run by a thread. This is used in order to reset the alarm every time a new thread starts and to return to the scheduler once it needs to terminate.

Assignment 2: Utility functions

```
void alarm_op(enum alarm_op_t operation, struct itimerval *previous) {
    switch (operation) {
        case DISABLE:
            set signal handler to ignore SIGALRM
            disable timer and store the previous
            break;
        case ENABLE_PREVIOUS:
            if previous has run out
                run_scheduler(READY);
            else
                set signal handler to normal handling
                enable timer and restore previous alarm
            break;
        case RESET:
            set signal handler to normal handling
            enable timer and reset alarm
            break;
    }
}
```

This function performs an operation on the timer and the signal handling.

Assignment 2: Utility functions

```
void sleep_until(unsigned long long timestamp) {
    calculate sleep amount based on timestamp

    while sleep amount is not 0 {
        usleep(sleep amount);
        recalculate sleep amount
    }
}


void thread_timeout_handler(int signum) {
    run_scheduler(state of current thread);
}
```

Assignment 2: API functions

```
int mythreads_init() {
    initialize threads list
    initialize semaphore list
    add main thread to the list

    initialize alarm and signal handler
    disarm the alarm

    initialize main thread fields
    return 1
}
```

```
void mythreads_exit() {
    destroy main coroutine
    destroy threads list
    destroy semaphore list
}
```

Assignment 2: API functions

```
int mythreads_yield() {
    run_scheduler(READY)
    return 1
}
```

```
int mythreads_sleep(int secs) {
    disable alarm
    set sleep until timestamp on current thread

    run_scheduler(SLEEPING)
    return 1
}
```

Assignment 2: API functions

```
int mythreads_join(mythr_t *thr) {
    disable alarm and store previous alarm
    if thr does not exist or has terminated
        enable previous alarm
        return 1

    set joining thread of currently running thread to thr
    run_scheduler(BLOCKED)
    return 1
}
```

Assignment 2: API functions

```
int mythreads_create(mythr_t *thr, void (body)(void *), void *arg) {
    disable alarm and store previous alarm
    if thr exists
        enable previous alarm
        return error

    add thr to thread list

    initialize thread fields

    if thread list size is 2
        reset alarm
    else
        enable previous alarm

    return 1
}
```

Assignment 2: API functions

```
int mythreads_destroy(mythr_t *thr) {
    disable alarm and store previous alarm
    if thr does not exist or has terminated
        enable previous alarm
        return 1

    destroy thread coroutine
    remove thread from thread list

    if thread list size is 1
        enable previous alarm
    else
        reset alarm

    return 1
}
```

Assignment 2: API functions

```
int mythreads_sem_create(mysem_t *s, int val) {
    disable alarm and store previous alarm
    if semaphore already exists
        restore previous alarm
        return error

    add semaphore to semaphores list

    set semaphore value to val
    initialize semaphore waiting queue

    restore previous alarm
    return 1
}
```

Assignment 2: API functions

```
int mythreads_sem_destroy(mysem_t *s) {
    disable alarm and store previous alarm
    if semaphore does not exists
        restore previous alarm
        return error

    destroy waiting list of semaphore
    remove semaphore from semaphores list

    restore previous alarm
    return 1
}
```

Assignment 2: API functions

```
int mythreads_sem_up(mysem_t *s) {
    disable alarm and store previous alarm
    if semaphore does not exists
        restore previous alarm
        return error

    if semaphore value is 1
        restore previous alarm
        return 0
    increment semaphore value

    if semaphore value is less than or equal to 0
        remove next thread from waiting queue of semaphore
        set the state of the next thread to ready

    restore previous alarm
    return 1
}
```

Assignment 2: Additional challenges

During the testing phase of the library, we discovered that the scanf function posed issues when reading from the standard input: The function is frequently interrupted by the alarm mechanism, causing it to return immediately with an interrupted error.

In order to fix this issue we wanted to implemented our own scanf function which would solve this problem. We would also need to make some changes to the scheduler:

The scheduler checks if standard input has been updated. Upon detecting an update, the scheduler switches to any thread that was waiting for input. The thread will consume the new data in the standard input only if the format string matches the input. If the format string does not match, the data remains unaltered. This allows for multiple threads that are waiting for new input to process the standard input.

Assignment 3

Assignment 3: Variables

```
mythreads_sem_t mtx;
mythreads_sem_t readers_queue;
mythreads_sem_t writers_queue;
int readers_waiting = 0;
int writers_wating = 0;
int readers = 0;
int writers = 0;
mythreads_sem_create(mtx, 1);
mythreads_sem_create(writers_queue, 0);
mythreads_sem_create(readers_queue, 0);
```

Assignment 3: Reader function

```
void reader(void *arg) {
    mythread_sem_down(mtx);
    if (writers are wating OR a writer in CS) {
        increase readers_waiting by 1;
        mythreads_sem_up(mtx);
        mythreads_sem_down(readers_queue);
        if (readers are waiting) {
            decrease readers_waiting by 1;
            increase readers by 1;
            mythreads_sem_up(readers_queue);
        }
        else { mythreads_sem_up(mtx); }
    }
    else {
        increase readers by 1;
        mythreads_sem_up(mtx);
    }

    /* read */
```

Assignment 3: Reader function

```
    mythreads_sem_down(mtx);
    decrease readers by 1;
    if (NO readers in CS AND writers blocked) {
        decrease writers_wating by 1;
        increase writers by 1;
        mythreads_sem_up(writers_queue);
    }
    mythreads_sem_up(mtx);
}
```

Assignment 3: Writer function

```
void writer(void *arg) {
    mythreads_sem_down(mtx);
    if (readers OR writers in CS) {
        increase writers_waiting by 1;
        mythreads_sem_up(mtx);
        mythreads_sem_down(writers_queue);
    } else {
        increase writers by 1;
        mythreads_sem_up(mtx);
    }

    /* write */
```

Assignment 3: Writer function

```
    mythreads_sem_down(mtx);
    decrease writers by 1;
    if (readers_waiting > 0) {
        decrease readers_waiting by 1;
        decrease readers by 1;
        mythreads_sem_up(readers_queue);
    } else {
        if (there are blocked writers) {
            decrease writers_waiting by 1;
            increase writers by 1;
            mythreads_sem_up(writers_queue);
        }
        mythreads_sem_up(mtx);
    }

}
```