OS Exercise 2

Operating Systems Reichman University

12.05.2025

Acknowledgments

We would like to thank the Hebrew University for the original idea and examples.

Submission Notes:

- **Submission Deadline:** Via Moodle by `9.06.2025`. **No late submissions will be accepted!**
- **Collaboration:** This assignment is to be done in pairs (i.e., two students per team).
- **Submission Format:** Submit a single ZIP file containing all the requested files.
- **README:** Include a `README.txt` file at the root of the ZIP. This file must contain the submitting students' information. Each line should follow this format:

  ```
  First name, Last name, Student ID
  ```
- **Working Environment:** This assignment requires the use of an Ubuntu OS with x86 64 bit architecture, be sure to work with the appropriate virtual machine/container/device.
- **Compilation Environment:** The checker uses `gcc13` on Ubuntu 24.04 LTS with the C23 standard.
- **Compilation and Execution:** Ensure your code compiles without errors and runs as expected.
- **Testing:** Write your own unit tests to validate your implementation by creating a `testuthreads.c` file containing a `main` function that tests your functions. **Do not submit `testuthreads.c`**. Make sure your submitted files **do not contain** a `main` function.
- **Covered Material:** This assignment covers topics from Presentation 3 and Recitation 3.
- **Header File Modification - You can't Modify the header file (including the structs)**
- **Dynamic Memory Allocation:** - Dynamic memory allocation (i.e malloc) is not allowed (and is not needed).

Introduction

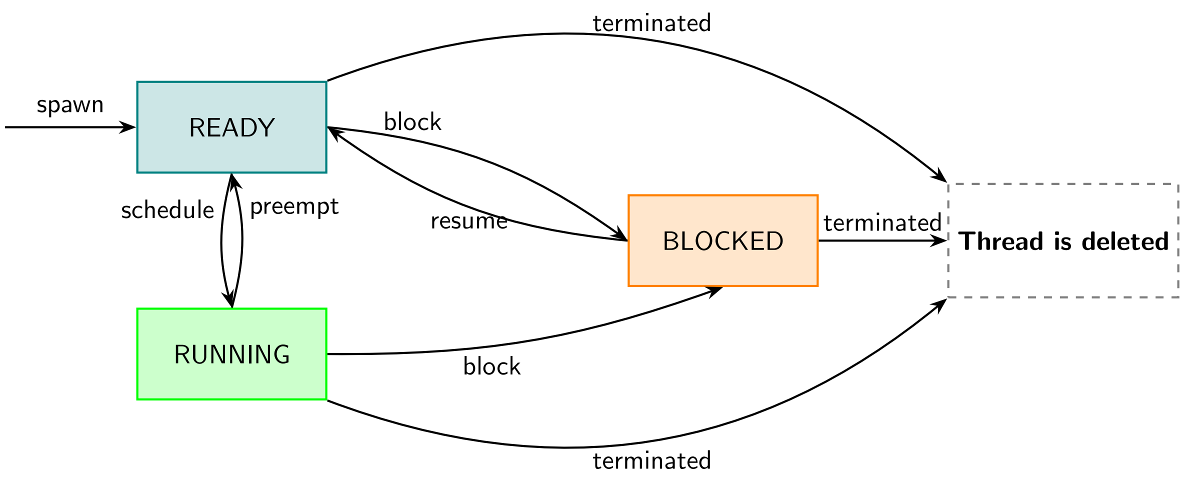
In this assignment, you will build a static library (compiled code without main) that creates and manages user-level threads. Your library's header is defined at `uthreads.h`. Your primary task is to implement all the functions specified in the uthreads API according to their documentation. You may also need to design and implement internal helper functions and data structures. There is no restriction on how many or what form these internal components take.

The Threads

Every program begins with a default main thread, which has a user-level thread ID of 0. All other threads must be explicitly created by the user-level library. Each thread is assigned a unique, non-negative integer as its ID. When creating a new user-thread (i.e. uthread_spawn), you must assign it the smallest non-negative integer that is not currently in use (for example, if thread 1 terminates and a new thread is created, it should receive the ID 1). The library must support at most `MAX_THREAD_NUM` threads at any time, including the main thread.

Every thread will always be in one of the following states: `RUNNING`, `BLOCKED`, or `READY`.

Thread State Diagram



Example two user threads with two seperate stacks.

```
/*
 * Simplified cooperative threads demo using sigsetjmp/siglongjmp.
 * This example is written for 64-bit Intel architectures only.
 * Two threads are set up with separate stacks, and they switch control
 * by saving and restoring execution contexts.
 */

#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>

// Define a type for addresses.
typedef unsigned long address_t;

// These indices refer to the positions in the jmp_buf where the stack pointer
    and program counter are stored.
#define JB_SP 6
#define JB_PC 7

// This function adjusts an address for use with sigsetjmp/siglongjmp.
// The inline assembly is used as a black box to "translate" the address.
address_t translate_address(address_t addr) {
    address_t ret;
    asm volatile("xor    %%fs:0x30, %0\n"
                 "rol    $0x11, %0\n"
```

```c
                  : "=g" (ret)
                  : "0" (addr));
    return ret;
}

// Constants for time and stack size.
#define SECOND 1000000       // 1 second in microseconds.
#define STACK_SIZE 4096      // Each thread gets a 4096-byte stack.

// Define a type for thread functions.
typedef void (*thread_entry_point)(void);

// Allocate separate stacks for the two threads.
char stack0[STACK_SIZE];
char stack1[STACK_SIZE];

// Create jump buffers to store the execution contexts for each thread.
sigjmp_buf env[2];
// This variable keeps track of the currently running thread (0 or 1).
int current_thread = -1;

// Function to switch to a specific thread using siglongjmp.
void jump_to_thread(int tid) {
    current_thread = tid;
    siglongjmp(env[tid], 1);
}

// Yield function: save the current thread's state and switch to the other thread.
void yield(void) {
    // Save the current execution context. sigsetjmp returns 0 when saving.
    int ret_val = sigsetjmp(env[current_thread], 1);
    // When coming back from a siglongjmp, sigsetjmp returns a nonzero value.
    if (ret_val == 0) {
        // Switch to the other thread (if current is 0, switch to 1; if 1, switch
            to 0).
        jump_to_thread(1 - current_thread);
    }
    // If ret_val is nonzero, the thread is resuming execution.
}

// The first thread: counts and yields every 3 iterations.
void thread0(void) {
    int counter = 0;
    while (1) {
        counter++;
        printf("Thread 0: counter = %d\n", counter);
        // Yield control after every 3 iterations.
        if (counter % 3 == 0) {
            printf("Thread 0 yielding...\n");
            yield();
        }
        usleep(SECOND);  // Sleep for 1 second.
    }
}

// The second thread: counts and yields every 5 iterations.
void thread1(void) {
    int counter = 0;
    while (1) {
        counter++;
        printf("Thread 1: counter = %d\n", counter);
        // Yield control after every 5 iterations.
        if (counter % 5 == 0) {
            printf("Thread 1 yielding...\n");
            yield();
        }
        usleep(SECOND);  // Sleep for 1 second.
    }
}

// Set up a thread by initializing its jump buffer to use a new stack and start
    at a given function.
void setup_thread(int tid, char *stack, thread_entry_point entry_point) {
    // Calculate the initial stack pointer (start at the top of the stack).
    address_t sp = (address_t)stack + STACK_SIZE - sizeof(address_t);
    // Get the function pointer for the thread's starting function.
    address_t pc = (address_t)entry_point;
    // Save the current context into the jump buffer.
    sigsetjmp(env[tid], 1);
    // Manually set the stack pointer and program counter in the jump buffer.
    env[tid]->__jmpbuf[JB_SP] = translate_address(sp);
    env[tid]->__jmpbuf[JB_PC] = translate_address(pc);
    // Clear the saved signal mask.
    sigemptyset(&env[tid]->__saved_mask);
}

// Initialize both threads.
void setup(void) {
    setup_thread(0, stack0, thread0);
    setup_thread(1, stack1, thread1);
}

// Main entry point: set up threads and start execution.
int main(void) {
    setup();
    // Begin execution with thread 0.
    jump_to_thread(0);
    return 0;
}
```

Listing 1: Simplified cooperative threads demo using sigsetjmp/siglongjmp (64-bit Intel architectures)

Scheduler

Your library must include a scheduler that manages thread execution using the Round-Robin (RR) scheduling algorithm. This algorithm works by allocating a fixed time slice, called a *quantum*, to each thread when it is in the `RUNNING` state.

Time

The process running time is measured by the Virtual Timer. Virtual time refers to the amount of CPU time a process actually uses, rather than the real (or wall clock) time that passes in the system. This distinction is crucial for understanding how threads are scheduled and managed in this assignment.

- **Virtual Time:**

- Measures the CPU time consumed by the process.
- Advances only when the process (or one of its threads) is actively running.
- Used to allocate and monitor a thread's quantum—i.e., the fixed time slice a thread receives when it is in the `RUNNING` state.
- Typically implemented using the `setitimer` system call with the `ITIMER_VIRTUAL` option.

- **Wall Clock Time:**
  - Represents the actual elapsed time in the real world.
  - Includes all time intervals, whether the process is running, waiting, or blocked.
  - Not used for scheduling in this assignment, as we focus solely on the process's active (virtual) running time.

```c
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <unistd.h>

// Signal handler for SIGVTALRM
void timer_handler(int signum) {
    // This function is called each time the virtual timer expires
    printf("Virtual timer expired\n");
}

int main(void) {
    struct sigaction sa;
    struct itimerval timer;

    // Set up the SIGVTALRM handler using sigaction
    sa.sa_handler = timer_handler;     // Specify our signal handler
    sigemptyset(&sa.sa_mask);          // No signals blocked during the handler
    sa.sa_flags = 0;                   // No special flags

    // Register the SIGVTALRM handler
    if (sigaction(SIGVTALRM, &sa, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    // Configure the virtual timer:
    // The timer will first expire after 1 second, and then every 1 second
       thereafter.
    timer.it_value.tv_sec = 1;         // Initial expiration in seconds
    timer.it_value.tv_usec = 0;        // Initial expiration in microseconds
    timer.it_interval.tv_sec = 1;      // Subsequent intervals in seconds
    timer.it_interval.tv_usec = 0;     // Subsequent intervals in microseconds

    // Start the virtual timer (ITIMER_VIRTUAL counts CPU time used by the
       process)
    if (setitimer(ITIMER_VIRTUAL, &timer, NULL) == -1) {
        perror("setitimer");
        return 1;
    }

    // Busy loop to consume CPU time so that the virtual timer counts down.
    while (1) {
        // Perform some CPU-bound task; this loop will trigger timer signals.
        // Note: In a real application, replace this with useful work.
    }

    return 0;
}
```

Listing 2: Example of using `sigaction` with a virtual timer (ITIMER_VIRTUAL)

Algorithm

The Round-Robin scheduling policy should operate as follows:

1. **Quantum Allocation:** Whenever a thread enters the `RUNNING` state, it is allocated a fixed number of microseconds to execute (the quantum).
2. **Preemption Conditions:** The currently running thread is preempted if:
   (a) Its allocated quantum expires.
   (b) It voluntarily blocks itself (for example, if it is waiting for an event or resource).
   (c) It terminates (either by its own action or via a call from another thread).
3. **Handling Preemption:** When a thread is preempted:
   (a) If the preemption is due to quantum expiration, move the thread to the end of the `READY` queue.
   (b) Immediately schedule the next thread from the front of the `READY` queue, switching it to the `RUNNING` state.
4. **Queue Management:** Every time a thread transitions into the `READY` state from any other state, it should be added to the end of the `READY` queue.
5. **Partial Quantum Usage:** If a thread does not consume its entire quantum (for instance, if it blocks before the quantum expires), the scheduler should immediately start the next thread as though the previous thread had fully used its time slice.

Library Functions

The API for your thread library is defined in the header file `uthread.h`. When these functions are called, they will cause state transitions as shown in the state diagram. A thread may invoke a function on its own ID (affecting its state) or on another thread's ID (affecting that thread's state).

Notes

1. You must implement and manage a data structure (such as a queue or list) to maintain the threads in the `READY` state. You are free to use any additional data structures that help achieve the required functionality.
2. If a thread is both blocked and sleeping, it should remain inactive until both conditions are resolved: its sleep duration has expired and another thread resumes it.
3. The main thread (i.e. `tid == 0`) operates using the same stack, program counter, and registers that were active when `uthread_init` was called. Therefore, you do not need to allocate a separate stack or manually set its SP and PC. Nonetheless, it should be managed just like any other thread during context switches.
4. **You must protect your "critical" state update sections with signal masking** to prevent a user-level context switch (that is caused by the timer signal) in order to prevent state corruption. If any of your masking operations fail, then print to stderr "system error: masking failed" and exit the process with an exit code of 1.

5. You **are not allowed to use dynamic allocations (i.e malloc)**, You may use a static array to manage the thread's TCBs and the thread Stacks.

Simplifying Assumptions

You may assume the following:

1. Every thread will call `uthread_terminate` before returning, whether it terminates itself or is terminated by another thread.
2. The allocated stack space for each spawned thread is sufficient and will not be exceeded during execution.
3. Neither the main thread nor any thread created via the uthreads library will manipulate timer signals (specifically `SIGVTALRM`) by sending, masking, or setting interval timers for them.

Error Messages

Your library should output error messages exclusively to `stderr`. No output should be sent to `stdout`. Use the following formats:

System Call Failure: When a system call fails (e.g., memory allocation), print a single line in the following format:

```
system error: text
```

Here, `text` should briefly describe the error. After printing, immediately call `exit(1)`.

Thread Library Failure: When a function in your thread library encounters an error (e.g., invalid input), print a single line in the following format:

```
thread library error: text
```

Again, `text` should provide a brief description of the error. Then, return the appropriate error value from the function.

The Library:

```
#ifndef _UTHREADS_H
#define _UTHREADS_H

#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <stdbool.h>
#include <sys/time.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

/* ==================================================================== */
/*                          Static Constants                            */
/* ==================================================================== */

/** Maximum number of threads (including the main thread). */
#define MAX_THREAD_NUM 100

/** Stack size per thread (in bytes). */
#define STACK_SIZE 4096

/**
```

```c
 * @brief Function pointer type for a thread's entry point.
 *
 * Each thread's entry function must take no arguments and return void.
 */
typedef void (*thread_entry_point)(void);

/* ================================================================= */
/*                        Internal Data Structures                    */
/* ================================================================= */

/**
 * @brief Enumeration of possible thread states.
 */
typedef enum {
    THREAD_UNUSED = 0, /**< Slot is unused. */
    THREAD_READY,      /**< Thread is ready to run. */
    THREAD_RUNNING,    /**< Thread is currently executing. */
    THREAD_BLOCKED,    /**< Thread is blocked (explicitly or sleeping). */
    THREAD_TERMINATED  /**< Thread has finished execution (internal use only). */
} thread_state_t;

/**
 * @brief Thread Control Block (TCB)
 *
 * Each thread (except for the main thread) has its own allocated stack and
    context.
 * The TCB stores all metadata required for managing the thread.
 */
typedef struct {
    int tid;                      /**< Unique thread identifier. */
    thread_state_t state;         /**< Current thread state. */
    sigjmp_buf env;               /**< Jump buffer for context switching using
        sigsetjmp/siglongjmp. */
    int quantums;                 /**< Count of quantums this thread has executed.
        */
    int sleep_until;              /**< Global quantum count until which the thread
        should sleep (0 if not sleeping). */
    thread_entry_point entry;     /**< Entry point function for the thread. */
} thread_t;

/* ================================================================= */
/*                          External Interface                        */
/* ================================================================= */

/**
 * @brief Initializes the user-level thread library.
 *
 * This function must be called before any other thread library function.
 * It sets up internal data structures, initializes the main thread (tid == 0) as
    running,
 * and configures the timer for quantum management. The main thread uses the
    process's regular stack.
 *
 * @param quantum_usecs Length of a quantum in microseconds (must be positive).
 * @return 0 on success; -1 on error (e.g., if quantum_usecs is non-positive).
 */
int uthread_init(int quantum_usecs);

/**
```

```c
 * @brief Creates a new thread.
 *
 * Allocates a new TCB and separate stack for the thread (using a char array).
 * The thread is added to the end of the READY queue.
 * Calling this function with a NULL entry_point or exceeding MAX_THREAD_NUM is
   an error.
 *
 * @param entry_point Pointer to the thread's entry function (must not be NULL).
 * @return On success, returns the new thread's ID; on failure, returns -1.
 */
int uthread_spawn(thread_entry_point entry_point);

/**
 * @brief Terminates a thread.
 *
 * Terminates the thread with the specified tid and releases all resources
   allocated for it.
 * The thread is removed from all scheduling structures. If no thread with the
   given tid exists,
 * it is considered an error. Terminating the main thread (tid == 0) will
   terminate the entire process
 * (after releasing allocated resources).
 *
 * @param tid Thread ID to terminate.
 * @return 0 on success; -1 on error. (Note: if a thread terminates itself or if
   the main thread terminates,
 * the function does not return.)
 */
int uthread_terminate(int tid);

/**
 * @brief Blocks a thread.
 *
 * Moves the thread with the given tid into the BLOCKED state.
 * A blocked thread may later be resumed using uthread_resume.
 * It is an error to block a thread that does not exist or to block the main
   thread (tid == 0).
 * Blocking a thread that is already BLOCKED is a no-op.
 *
 * @param tid Thread ID to block.
 * @return 0 on success; -1 on error.
 */
int uthread_block(int tid);

/**
 * @brief Resumes a blocked thread.
 *
 * Moves a thread from the BLOCKED state to the READY state.
 * If the thread is already in RUNNING or READY state, this call has no effect.
 * It is an error if no thread with the given tid exists.
 *
 * @param tid Thread ID to resume.
 * @return 0 on success; -1 on error.
 */
int uthread_resume(int tid);

/**
 * @brief Puts the running thread to sleep.
 *
 * Blocks the currently running thread for a specified number of quantums.
 * The current quantum is not counted; sleeping begins with the next quantum.
 * After the sleep period expires, the thread is moved to the end of the READY
   queue.
 * It is an error for the main thread (tid == 0) to call this function.
 *
 * @param num_quantums Number of quantums to sleep.
 * @return 0 on success; -1 on error.
 */
int uthread_sleep(int num_quantums);

/**
 * @brief Returns the calling thread's ID.
 *
 * @return The thread ID of the calling thread.
 */
int uthread_get_tid();

/**
 * @brief Returns the total number of quantums since the library was initialized.
 *
 * The count starts at 1 immediately after uthread_init. Every new quantum,
   regardless of cause,
 * increments this counter.
 *
 * @return Total number of quantums.
 */
int uthread_get_total_quantums();

/**
 * @brief Returns the number of quantums the thread with the specified tid has
   run.
 *
 * For a thread in the RUNNING state, the current quantum is included.
 * An error is returned if no thread with the given tid exists.
 *
 * @param tid Thread ID.
 * @return Number of quantums for the specified thread; -1 on error.
 */
int uthread_get_quantums(int tid);

/* ================================================================= */
/*              Internal Helper Functions and Structures             */
/* ================================================================= */
/*
 * The following declarations are intended for internal use by the thread library.
 * They provide guidance on how to structure your implementation using
   sigsetjmp/siglongjmp
 * and manually managed stacks.
 */

/**
 * @brief Scheduler: Selects the next thread to run.
 *
 * This function examines the READY queue and selects the next thread for
   execution.
 * It handles state transitions and triggers a context switch.
 */
void schedule_next(void);
```

```c

/**
 * @brief Context switch helper.
 *
 * Uses sigsetjmp and siglongjmp to save the current thread's context and restore
     the context of the next thread.
 *
 * @param current Pointer to the current thread's TCB.
 * @param next Pointer to the next thread's TCB.
 */
void context_switch(thread_t *current, thread_t *next);

/**
 * @brief Timer signal handler.
 *
 * Registered as the handler for timer signals, this function updates global
     quantum counters
 * and initiates a scheduling decision when a quantum expires.
 *
 * @param signum The signal number (e.g., SIGVTALRM).
 */
void timer_handler(int signum);

/**
 * @brief Initializes a thread's jump buffer.
 *
 * Sets up the thread's jump buffer for context switching by manually assigning
     the stack pointer
 * and program counter using sigsetjmp/siglongjmp techniques. You may need to
     perform architecture-specific
 * address translation (see provided reference implementation) when initializing
     the context.
 *
 * @param tid Thread ID.
 * @param stack Pointer to the thread's allocated stack (a char array).
 * @param entry_point Pointer to the thread's entry function.
 */
void setup_thread(int tid, char *stack, thread_entry_point entry_point);

#endif /* _UTHREADS_H */
```

Listing 3: User-level thread library header file (uthreads.h)

Background Reading and Resources

To gain a complete understanding of the relevant system calls, please consult the following man pages:

- `setitimer (2)`
- `getitimer (2)`
- `sigaction (2)`
- `sigsetjmp (3)`
- `siglongjmp (3)`
- `sigprocmask (2)`
- `sigemptyset`, `sigaddset`, `sigdelset`, `sigfillset`, `sigismember (3)`

- sigpending (2)
- sigwait (3)

Example of using sigaction with sigint.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Signal handler for SIGINT
void signal_handler(int signum) {
    printf("Received signal %d (SIGINT). Exiting gracefully...\n", signum);
    exit(0);
}

int main(void) {
    struct sigaction sa;

    // Set up the sigaction structure to specify the signal handler
    sa.sa_handler = signal_handler;  // Set the signal handler function
    sigemptyset(&sa.sa_mask);         // No additional signals blocked during
        execution of the handler
    sa.sa_flags = 0;                  // No special flags

    // Register the signal handler for SIGINT
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("Program is running. Press Ctrl+C to trigger SIGINT.\n");

    // Main loop: the program will continue running until it receives SIGINT
    while (1) {
        sleep(1);
    }

    return 0;
}
```

Listing 4: Example of using sigaction to handle SIGINT in a general case

Final Notes

- **Read the Instructions Carefully:** Ensure that you adhere to all restrictions and requirements.
- **Testing:** Test your code thoroughly with various inputs to ensure correctness, especially under concurrent execution.
  - **Also test without a debugger** since debuggers may affect synchronization behavior and hide bugs.
  - **Debugging:** Start with a small number of threads (e.g., one, then two, then five) and increase gradually.
  - Use debug printouts to help identify issues, but **make sure to remove or comment out** any debug messages before submission as they may affect grading.

- **Code Quality:** Write clean, well-commented code to help the graders understand your implementation. Comments will also help you understand what you are trying to achieve.
- **Unit Tests:** You are (more than) encouraged to use unit tests to validate your synchronization mechanisms.
- **Grading Reminder:** A happy grader is a merciful grader!

Good Luck!