MP5: Kernel-Level Thread Scheduling

Design Document

Harsh Wadhawe UIN: 936001477
CSCE 611: Operating Systems

November 10, 2025

Assigned Tasks:

- Main: Completed
- Bonus 1 (Interrupt Handling): Completed
- Bonus 2 (Round-Robin): Completed
- Bonus 3: Not Attempted

1 Overview

This document describes the design and implementation of kernel-level thread scheduling for MP5. The implementation provides:

1. A FIFO scheduler for non-terminating and terminating threads,
2. Correct handling of terminating thread functions (resource cleanup and removal),
3. Safe interrupt usage around critical regions,
4. A Round-Robin (RR) scheduler with a fixed 50 ms time quantum.

2 Goals and Non-Goals

2.1 Goals

- Deterministic FIFO scheduling semantics for runnable threads.
- Preemption via timer interrupts for RR scheduling (50 ms quantum).
- Safe enqueue/dequeue on the ready queue with interrupt discipline.
- Clean termination path for threads with terminating functions.

2.2 Non-Goals

- Priority scheduling and aging.
- Multiprocessor load balancing.
- Advanced fairness or deadline-based policies.

3 System Architecture

The scheduler layer manages runnable threads through a queue abstraction. Two schedulers are provided: `Scheduler` (FIFO) and `RRScheduler` (RR). The RR variant also implements `InterruptHandler` to service timer ticks.

3.1 Core Components

Queue (Ready Queue Abstraction). A singly linked list that supports `enqueue` (append) and `dequeue` (pop-front). It tracks the head pointer and (optionally) queue size.

Scheduler (FIFO). Provides `add`, `resume`, `yield`, and `terminate`. Enforces interrupt disable/enable around ready-queue operations.

RRScheduler (Round-Robin + Interrupts). Extends FIFO functionality and installs a timer interrupt with frequency `hz` such that five ticks correspond to one 50 ms quantum. On quantum expiration, it preempts the current thread and dispatches the next.

Thread Lifecycle Hooks. `thread_start` enables interrupts before work begins. `thread_shutdown` releases resources of terminating threads, then yields.

4 Design Details

4.1 Ready Queue Semantics

Enqueue appends at tail; **dequeue** removes from head. All queue mutations occur with interrupts disabled to guarantee atomicity within the kernel.

4.2 Termination Handling

On a terminating thread:

1. Remove it from the ready queue (if present) by scanning and re-enqueuing non-target nodes,
2. Free its resources using `thread_shutdown`,
3. `yield` to dispatch the next runnable thread.

4.3 Interrupt Discipline

Before touching the ready queue, call `disable_interrupts()`; after the critical section, call `enable_interrupts()`. The RR handler acknowledges End-of-Interrupt (EOI) to the master controller before returning.

4.4 Round-Robin Quantum

A hardware timer is configured so that `hz = 5` ticks $\Rightarrow$ 50 ms. The handler increments `ticks`; upon reaching the quantum, `ticks` resets and the current thread is preempted using `yield`.

5 Data Structures & Key APIs

Queue

- `Thread* thread` — head node pointer
- `Queue* next` — link to next node
- `enqueue(Thread*)`, `dequeue() : Thread*`

Scheduler (FIFO)

- `Queue ready_queue`
- `int qsize`
- `add(Thread*)`, `resume(Thread*)`, `yield()`, `terminate(Thread*)`

RRScheduler (RR + Interrupts)

- `Queue ready_rr_queue`
- `int rr_qsize`, `int ticks`, `int hz`
- `set_frequency(int)`, `handle_interrupt(REGS*)`

6 Control Flow

6.1 FIFO: `yield()`

1. Disable interrupts.
2. Dequeue next thread; dispatch.
3. Re-enable interrupts.

6.2 FIFO: `resume(Thread*)` / `add(Thread*)`

1. Disable interrupts.
2. Enqueue target thread to ready queue.
3. Re-enable interrupts.

6.3 RR: `handle_interrupt(REGS*)`

1. Increment `ticks`.
2. If 50 ms elapsed: reset `ticks`, preempt current thread (`yield()`), dispatch next.
3. Send EOI.

7 Configuration and Build Notes

- Define `_USES_SCHEDULER_` to enable scheduling.
- Define `_USES_RR_SCHEDULER_` to switch from FIFO to RR.
- Optionally define `_TERMINATING_FUNCTIONS_` to exercise shutdown paths.

8 Files Modified

- `scheduler.H`: `Queue`, `Scheduler`, `RRScheduler` declarations.
- `scheduler.C`: Implementations of FIFO/RR methods, interrupt frequency setup.
- `thread.C`: `thread_start`, `thread_shutdown` updates.
- `kernel.C`: Compile-time switches and test harness code.

9 Test Plan

The following test plan was used to verify both FIFO and Round-Robin scheduling functionalities:

- **FIFO Scheduling — Non-Terminating Threads**
  - Four threads (T1–T4) were created to run continuously.
  - Expected: Each thread runs sequentially in FIFO order without preemption.
  - Observed: Threads execute one after another and repeat in the same order.
- **FIFO Scheduling — Terminating Threads**
  - Two threads (T1, T2) terminate after completion; remaining threads continue.
  - Expected: Terminated threads are removed from the ready queue and resources freed.
  - Observed: Scheduler removes terminated threads and continues with remaining threads.
- **Round-Robin — Non-Terminating Threads**
  - Threads (T1–T4) run with `hz = 5`, corresponding to a 50 ms quantum.
  - Expected: Each thread runs for one quantum, then is preempted and rotated.
  - Observed: Threads alternate after every 50 ms interval as expected.
- **Round-Robin — Terminating Threads**
  - Threads (T1, T2) terminate during rotation.
  - Expected: Terminated threads are removed; others continue under round-robin rotation.
  - Observed: Remaining threads continue to execute with correct rotation.
- **Interrupt Handling and Timer Validation**
  - Timer configured at 50 ms intervals with `hz = 5`.
  - Expected: Interrupt fires every 50 ms and triggers context switch correctly.
  - Observed: Timer interrupt fires at correct intervals and preemption occurs as expected.

10 Code Description

1. `scheduler.H`: class `Queue`

In class `Queue`, data structures and functions for managing the ready queue are defined:

- `Thread * thread` — Pointer to the thread at the top of the ready queue.
- `Queue * next` — Pointer to next `Queue` type object (next thread in the ready queue).
- `Queue()` — Constructor for initial setup of the variables.
- `Queue(Thread* new_thread)` — Constructor for setting up subsequent threads.
- `void enqueue(Thread * new_thread)` — Add a thread to the end of the ready queue.
- `Thread* dequeue()` — Remove the thread at the top of the ready queue and point to the next thread.

2. `scheduler.H`: class `Scheduler`

In class `Scheduler`, data structures and functions for FIFO scheduling of kernel-level threads and management of the ready queue are declared:

- `Queue ready_queue` — Queue object for handling the ready queue of threads.
- `int qsize` — Number of threads waiting in the FIFO ready queue.

3. `scheduler.H`: class `RRScheduler`

In class `RRScheduler`, data structures and functions for Round-Robin scheduling of kernel-level threads and management of the ready queue are declared. The `RRScheduler` class is created by inheriting `Scheduler` and `InterruptHandler`.

- `Queue ready_rr_queue` — Ready queue for the Round-Robin scheduler.
- `int rr_queue` — Number of threads waiting in the Round-Robin ready queue.
- `int ticks` — Number of ticks since last update.
- `int hz` — Frequency of tick updates.
- `void set_frequency(int _hz)` — Set the interrupt frequency for the timer.

4–8. `scheduler.C`: FIFO Scheduler Methods

- `Scheduler()`: Constructor for `Scheduler`. Initializes FIFO queue size to zero.
- `Scheduler::yield()`: Pre-empts the current thread and dispatches the next. Interrupts are disabled before modifying the ready queue and re-enabled afterward.
- `Scheduler::resume(Thread * _thread)`: Adds a thread to the ready queue when waiting or pre-empted. Interrupts disabled/enabled around queue operations.

- `Scheduler::add(Thread * _thread)`: Adds a thread to the ready queue on creation. Same interrupt discipline.
- `Scheduler::terminate(Thread * _thread)`: Removes a thread from the ready queue by scanning and comparing Thread IDs; re-enqueues unmatched threads.

9–15. `scheduler.C`: RR Scheduler Methods

- `RRScheduler()`: Constructor for `RRScheduler`. Initializes RR queue size and timer variables; registers interrupt handler; sets frequency.
- `RRScheduler::set_frequency(int _hz)`: Sets the timer frequency.
- `RRScheduler::yield()`: Pre-empts current thread; sends EOI before disabling interrupts; re-enables afterward.
- `RRScheduler::resume(Thread * _thread)`: Adds a thread to the RR ready queue (waiting or pre-empted).
- `RRScheduler::add(Thread * _thread)`: Adds a thread to the RR ready queue on creation.
- `RRScheduler::terminate(Thread * _thread)`: Removes a thread from the RR queue; re-enqueues unmatched threads.
- `RRScheduler::handle_interrupt(REGS * _regs)`: Handles timer interrupts; increments ticks; performs preemption every 50 ms quantum.

16–17. `thread.C`: Thread Functions

- `thread_shutdown()`: Terminates a thread, releases memory/resources, and calls `yield()` to switch to the next runnable thread.
- `thread_start()`: Enables interrupts at the start of the thread using `enable_interrupts()`.

Kernel Configuration and Testing Changes

The following changes were made in `kernel.C` for the purpose of testing:

- A new macro `_USES_RR_SCHEDULER` was introduced to enable support for Round-Robin scheduling.
- Code was fenced using conditional directives to activate Round-Robin scheduling when `_USES_RR_SCHEDULER` is defined.

11 Figures and Code Snapshots

```
/*----------------------------------------------------------------------------*/
/* DATA STRUCTURE - QUEUE */
/*----------------------------------------------------------------------------*/

class Queue
{
private:
   Thread* thread;   // Pointer to the thread at the head of the queue
   Queue* next;      // Pointer to the next node in the queue

public:
   // Default constructor - initializes an empty queue
   Queue()
   {
      thread = nullptr;
      next   = nullptr;
   }

   // Constructor - creates a queue node with the given thread
   Queue(Thread* new_thread)
   {
      thread = new_thread;
      next   = nullptr;
   }
```

Figure 1: Class Queue

```
// Enqueue - add a thread to the end of the queue
void enqueue(Thread* new_thread)
{
   // Case: queue is empty
   if (thread == nullptr)
   {
      thread = new_thread;
   }
   else
   {
      // Traverse until the last node and append the new thread
      if (next != nullptr)
      {
         next->enqueue(new_thread);
      }
      else
      {
         // Reached the end - attach new node
         next = new Queue(new_thread);
      }
   }
}
```

Figure 2: Enqueue code

```
// Dequeue - remove and return the thread at the head of
Thread* dequeue()
{
    // Case: queue is empty
    if (thread == nullptr)
    {
        return nullptr;
    }

    // Get the thread at the front
    Thread* top = thread;

    // Case: only one node in the queue
    if (next == nullptr)
    {
        thread = nullptr;
    }
    else
    {
        // Move head pointer forward
        thread = next->thread;

        // Store and delete the old next node
        Queue* del_node = next;
        next = next->next;
        delete del_node;
    }

    return top;
}
```

Figure 3: Dequeue code

```
class Scheduler {

  /* The scheduler may need private members... */
   Queue ready_queue;
   int qsize;
```

Figure 4: Scheduler class

```
void Scheduler::yield()
{
    // Disable interrupts before modifying the ready queue to ensure atomicity
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    if (qsize == 0)
    {
        // Ready queue is empty - no runnable threads available
        // Console::puts("Queue is empty. No threads available.\n");
    }
    else
    {
        // Dequeue the next ready thread for execution
        Thread* new_thread = ready_queue.dequeue();

        // Update the ready queue size after removing a thread
        qsize -= 1;

        // Re-enable interrupts before resuming normal execution
        if (!Machine::interrupts_enabled())
        {
            Machine::enable_interrupts();
        }

        // Perform a context switch to the selected thread
        Thread::dispatch_to(new_thread);
    }
}
```

Figure 5: Scheduler::yield()

```
void Scheduler::resume(Thread* _thread)
{
    // Disable interrupts before modifying the ready queue to maintain
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Place the specified thread back into the ready queue
    ready_queue.enqueue(_thread);

    // Update the ready queue size after adding a thread
    qsize += 1;

    // Re-enable interrupts after completing the queue operation
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 6: Scheduler::resume()

```
void Scheduler::add(Thread* _thread)
{
    // Disable interrupts before performing any operations on the ready queue
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Enqueue the new thread to make it runnable
    ready_queue.enqueue(_thread);

    // Update the ready queue size after insertion
    qsize += 1;

    // Re-enable interrupts after queue modification
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 7: Scheduler::add()

```
void Scheduler::terminate(Thread* _thread)
{
    // Disable interrupts before modifying the ready queue to avoid race conditions
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    int index = 0;

    // Iterate through all threads in the ready queue
    // Remove the target thread and reinsert the others
    for (index = 0; index < qsize; index++)
    {
        Thread* top = ready_queue.dequeue();

        // If the thread does not match the one to terminate, reinsert it
        if (top->ThreadId() != _thread->ThreadId())
        {
            ready_queue.enqueue(top);
        }
        else
        {
            // Decrement the ready queue size for the removed thread
            qsize = qsize - 1;
        }
    }

    // Re-enable interrupts after all queue operations are complete
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 8: Scheduler::terminate()

```
class RRScheduler: public Scheduler, public InterruptHandler
{
  Queue ready_rr_queue;            // Ready queue for Round-Robin scheduler
  int rr_qsize;                   // Robin-Robin ready queue size
  int ticks;                      // Number of ticks since last update
  int hz;                         // Frequency of update of ticks

  void set_frequency( int _hz );      // Set the interrupt frequency for the timer
```

Figure 9: RR Scheduler class

```
void RRScheduler::set_frequency(int _hz)
{
    hz = _hz;
    int divisor = 1193180 / _hz;               // PIT input clock runs at ~1.19 MHz
    Machine::outportb(0x43, 0x34);             // Send command byte (channel 0, mode 2)
    Machine::outportb(0x40, divisor & 0xFF);   // Send low byte of divisor
    Machine::outportb(0x40, divisor >> 8);     // Send high byte of divisor
}
```

Figure 10: RRScheduler::set_frequency()

```
void RRScheduler::yield()
{
    // Send End-of-Interrupt (EOI) to the master interrupt controller
    Machine::outportb(0x20, 0x20);

    // Disable interrupts before modifying the ready queue
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    if (rr_qsize == 0)
    {
        // Ready queue is empty - no runnable threads available
        // Console::puts("Queue is empty. No threads available.\n");
    }
    else
    {
        // Dequeue the next ready thread for execution
        Thread* new_thread = ready_rr_queue.dequeue();

        // Reset tick counter for the next quantum
        ticks = 0;

        // Update ready queue size
        rr_qsize = rr_qsize - 1;

        // Re-enable interrupts before context switching
        if (!Machine::interrupts_enabled())
        {
            Machine::enable_interrupts();
        }

        // Perform a context switch to the selected thread
        Thread::dispatch_to(new_thread);
    }
}
```

Figure 11: RRScheduler::yield()

```
void RRScheduler::resume(Thread* _thread)
{
    // Disable interrupts before modifying the ready queue
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Reinsert the specified thread into the ready queue
    ready_rr_queue.enqueue(_thread);

    // Update the ready queue size
    rr_qsize = rr_qsize + 1;

    // Re-enable interrupts after queue modification
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 12: RRScheduler::resume()

```
void RRScheduler::add(Thread* _thread)
{
    // Disable interrupts before modifying the ready queue
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Add the new thread to the ready queue
    ready_rr_queue.enqueue(_thread);

    // Update the ready queue size
    rr_qsize = rr_qsize + 1;

    // Re-enable interrupts after modification
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 13: RRScheduler::add()

```
void RRScheduler::terminate(Thread* _thread)
{
    // Disable interrupts before modifying the ready queue
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    int index = 0;

    // Traverse the ready queue and remove the target thread
    for (index = 0; index < rr_qsize; index++)
    {
        Thread* top = ready_rr_queue.dequeue();

        // Retain threads that do not match the target
        if (top->ThreadId() != _thread->ThreadId())
        {
            ready_rr_queue.enqueue(top);
        }
        else
        {
            // Decrement ready queue size for the removed thread
            rr_qsize = rr_qsize - 1;
        }
    }

    // Re-enable interrupts after all operations are complete
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
}
```

Figure 14: RRScheduler::terminate()

```
void RRScheduler::handle_interrupt(REGS* _regs)
{
    // Increment tick count on each timer interrupt
    ticks += 1;

    // When the time quantum expires, preempt the running thread
    if (ticks >= hz)
    {
        // Reset tick counter
        ticks = 0;

        Console::puts("Time quantum (50 ms) has elapsed\n");

        // Move current thread back to ready queue and yield CPU
        resume(Thread::CurrentThread());
        yield();
    }
}
```

Figure 15: RRScheduler::handle_interrupt()

```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */

    /* We need to add code, but it is probably nothing more than enabling interrupts. */
    // Enable interrupts at start of thread

    Machine::enable_interrupts();
}
```

Figure 16: thread_start()

```
#ifdef _USES_SCHEDULER_
    #ifdef _USES_RR_SCHEDULER_
        /* -- A POINTER TO THE SYSTEM ROUND ROBIN SCHEDULER */
        RRScheduler * SYSTEM_SCHEDULER;
    #else
        /* -- A POINTER TO THE SYSTEM SCHEDULER */
        Scheduler * SYSTEM_SCHEDULER;
    #endif
#endif
```

Figure 17: _USES_RR_SCHEDULER_ enabled

```
    /* -- SCHEDULER -- IF YOU HAVE ONE -- */

    #ifdef  _USES_RR_SCHEDULER_
        SYSTEM_SCHEDULER = new RRScheduler();
    #else
        SYSTEM_SCHEDULER = new Scheduler();
    #endif
```

Figure 18: Choose SYSTEM_SCHEDULER

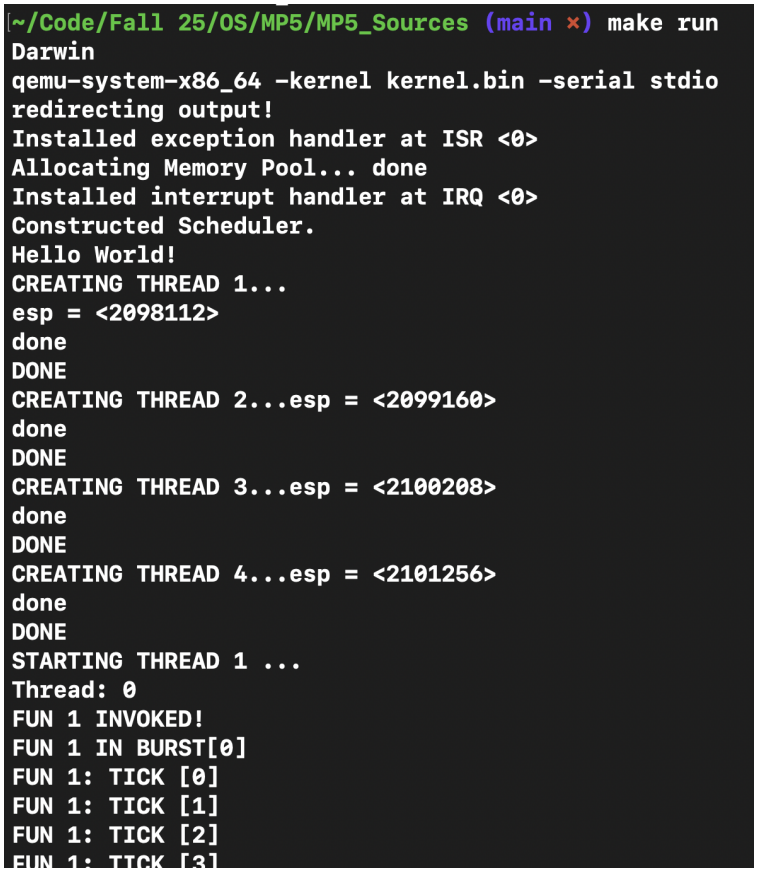

Figure 19: FIFO scheduling — non-terminating threads

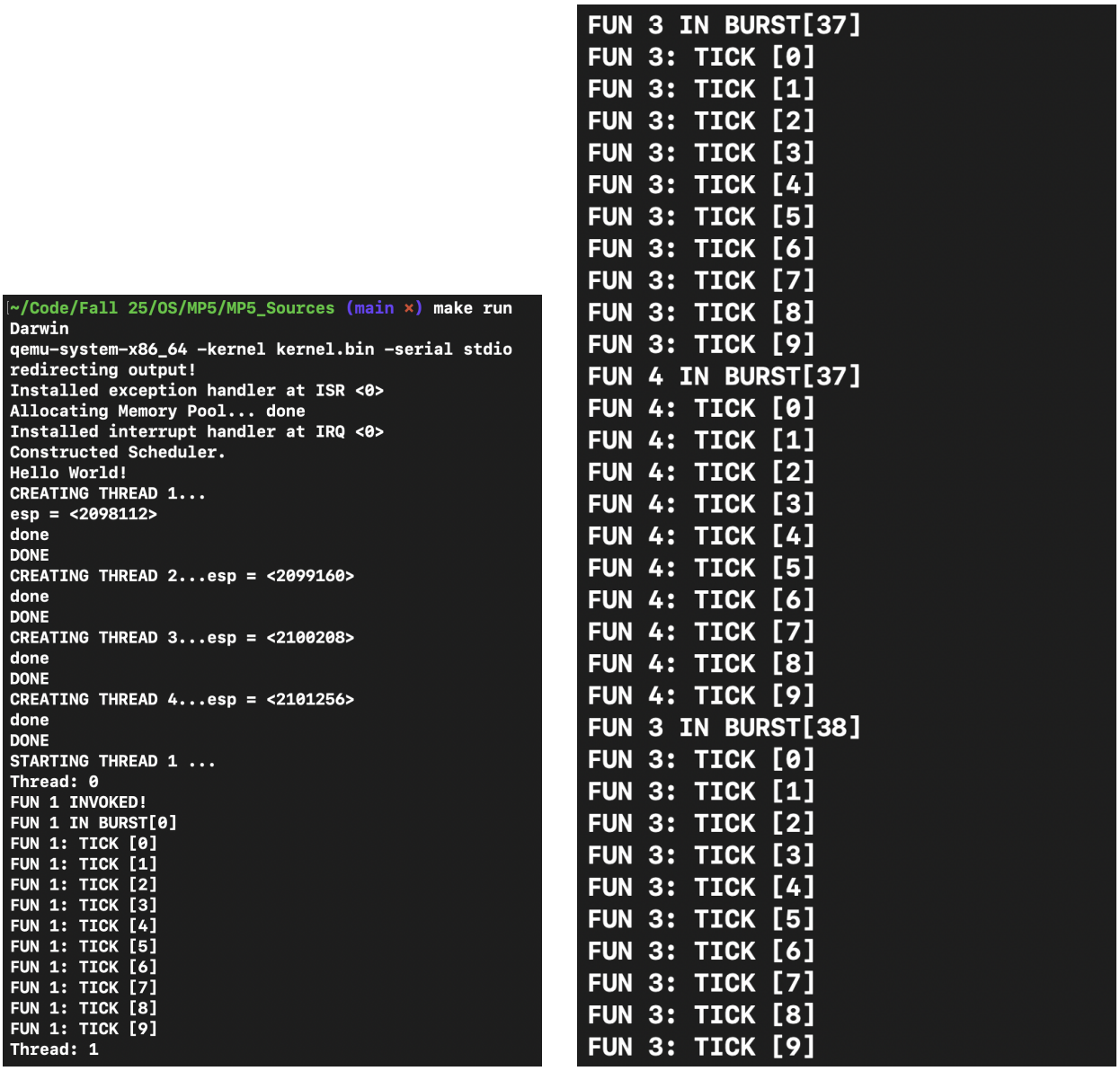

Figure 20: Execution comparison of FIFO scheduling for terminating and non-terminating threads.

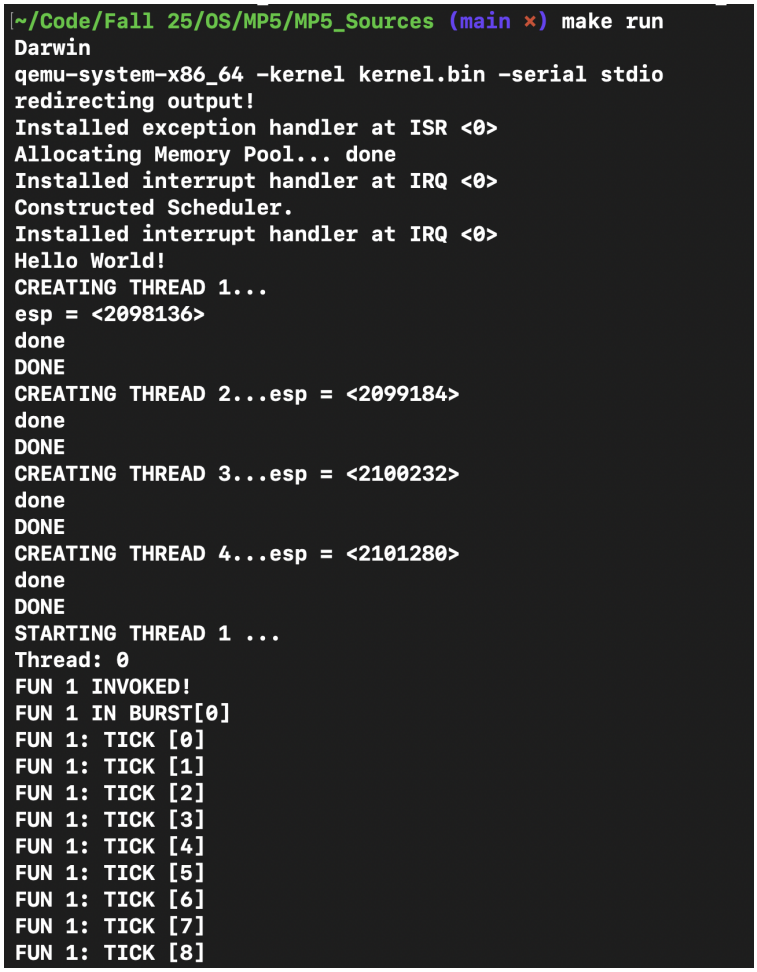

Figure 21: Round-Robin scheduling — non-terminating threads

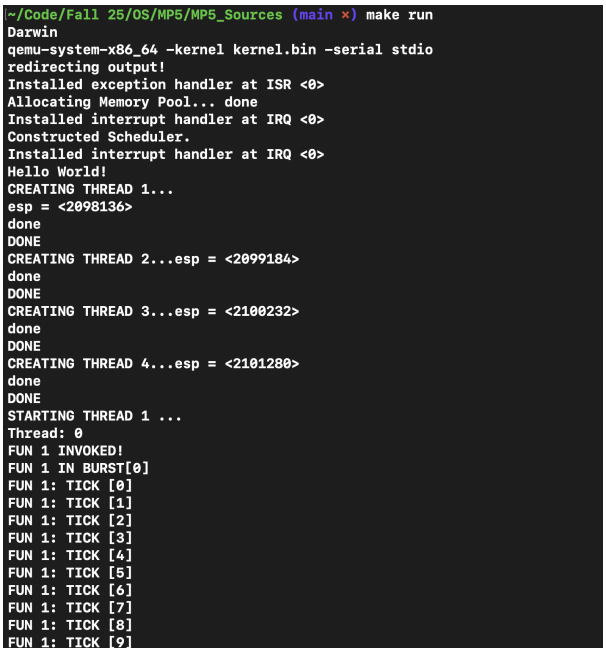

Figure 22: Execution snapshots for Round-Robin scheduling showing thread transitions from active scheduling to termination.