MP5: Kernel-Level Thread Scheduling

Cheng-Yun Cheng
UIN: 633002216
CSCE611: Operating System

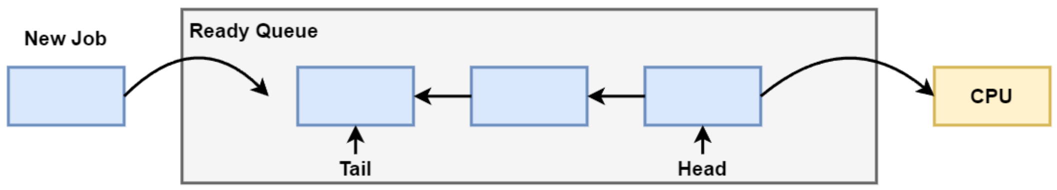
Assigned Tasks

Main: Completed.
Bonus Option 1: Completed.
Bonus Option 2: Completed.
Bonus Option 3: Did not attempt.

System Design

The goal of the machine problem is to implement a FIFO scheduler of multiple kernel-level thread.

- A FIFO scheduler is implemented. Its functionality includes adding new threads, resuming threads, yielding the current thread, and terminating threads.



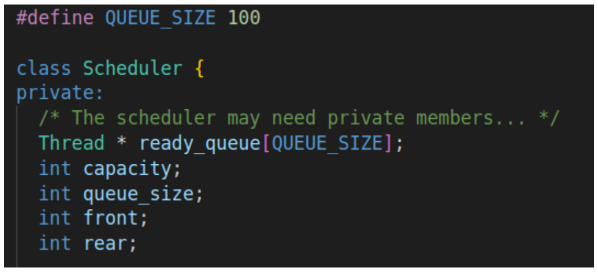
- Interrupts are handled. When a thread is running, interrupt is enabled. On the other hand, when dealing with mutual exclusion, like modifying the ready queue, interrupt should be disabled.
- A basic round-robin is implemented by adding a timer to preempt the current thread.

Code Description (Main)

I changed scheduler.h, scheduler.c, thread.h and thread.c to implement the FIFO scheduler.

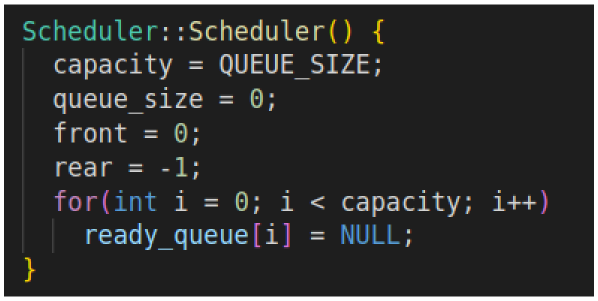
scheduler.h: data member :

- Thread * ready_queue: The ready queue.
- int capacity: The maximum size of the ready queue.
- int queue_size: The size of the ready queue.
- int front: The head of the queue.
- int rear: The tail of the queue.



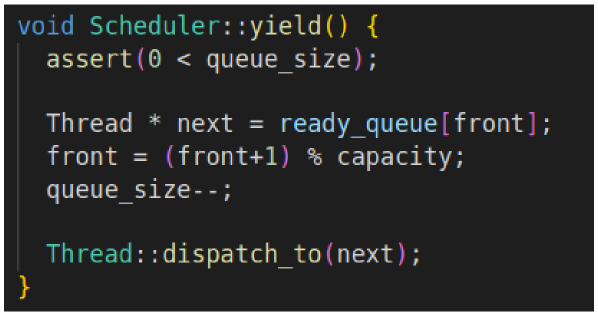
```
#define QUEUE_SIZE 100

class Scheduler {
private:
  /* The scheduler may need private members... */
  Thread * ready_queue[QUEUE_SIZE];
  int capacity;
  int queue_size;
  int front;
  int rear;
```

scheduler.c: Scheduler constructor : Initialize all the data members in Scheduler.



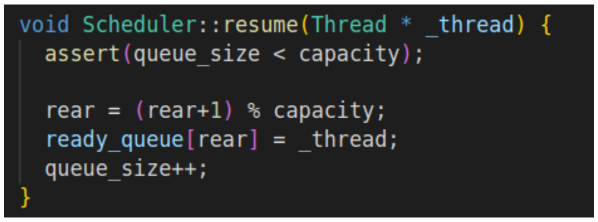
```
Scheduler::Scheduler() {
  capacity = QUEUE_SIZE;
  queue_size = 0;
  front = 0;
  rear = -1;
  for(int i = 0; i < capacity; i++)
    ready_queue[i] = NULL;
}
```

scheduler.c: yield : Select the next thread from the ready queue and load it onto the CPU. Check whether the size of the ready queue is empty or not first. If the queue is not empty, remove the first thread in the queue and update the queue. And then, call the dispatch_to function to invoke the context switch.



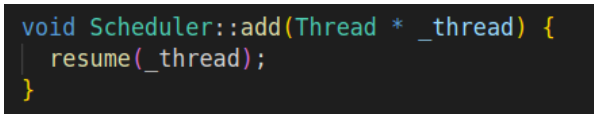
```
void Scheduler::yield() {
  assert(0 < queue_size);

  Thread * next = ready_queue[front];
  front = (front+1) % capacity;
  queue_size--;

  Thread::dispatch_to(next);
}
```

scheduler.c: resume : Add the given thread to the ready queue. Check whether the size of the ready queue is full or not first. If the queue is not full, add the given thread in the end of the queue and update the queue.



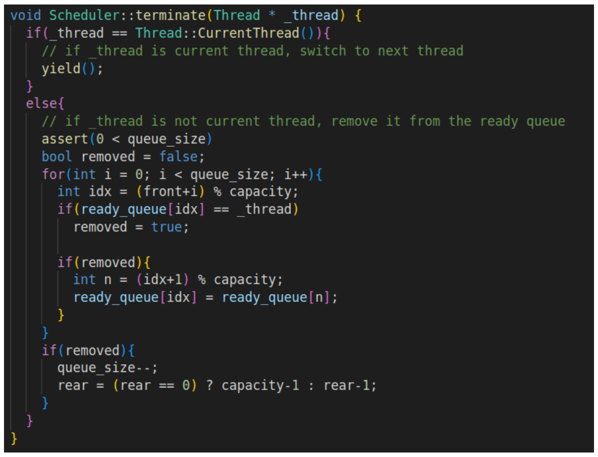
```
void Scheduler::resume(Thread * _thread) {
  assert(queue_size < capacity);

  rear = (rear+1) % capacity;
  ready_queue[rear] = _thread;
  queue_size++;
}
```

scheduler.c: add : Add the given thread to the ready thread. The function is called after thread creation. Just call the resume function.



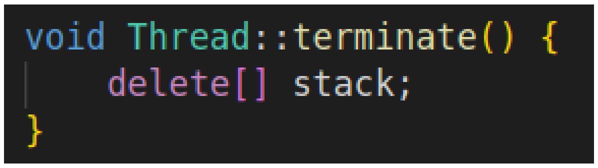
```
void Scheduler::add(Thread * _thread) {
  resume(_thread);
}
```

scheduler.c: terminate : The function is called when the given thread is in preparation for destruction. If the given thread is the current thread, call the yield function to give the CPU to the next thread. If the given thread is not the current thread, remove it from the ready queue and update the queue.



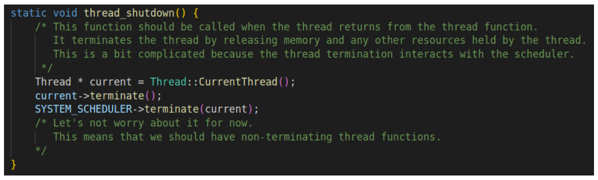
```
void Scheduler::terminate(Thread * _thread) {
  if(_thread == Thread::CurrentThread()){
    // if _thread is current thread, switch to next thread
    yield();
  }
  else{
    // if _thread is not current thread, remove it from the ready queue
    assert(0 < queue_size)
    bool removed = false;
    for(int i = 0; i < queue_size; i++){
      int idx = (front+i) % capacity;
      if(ready_queue[idx] == _thread)
        removed = true;

      if(removed){
        int n = (idx+1) % capacity;
        ready_queue[idx] = ready_queue[n];
      }
    }
    if(removed){
      queue_size--;
      rear = (rear == 0) ? capacity-1 : rear-1;
    }
  }
}
```

thread.c: terminate : Release the memory of stack in the thread. Call delete[] stack.



```
void Thread::terminate() {
    delete[] stack;
}
```

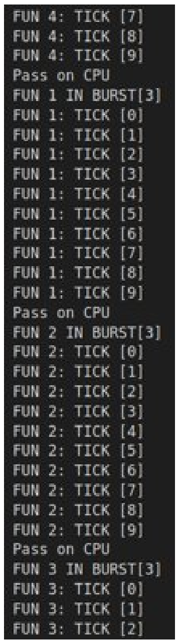
thread.c: thread_shutdown : Before terminating, the current thread releases its stacks by calling Thread::terminate and give the CPU to the next thread by calling Scheduler::terminate.



```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */
    Thread * current = Thread::CurrentThread();
    current->terminate();
    SYSTEM_SCHEDULER->terminate(current);
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
}
```

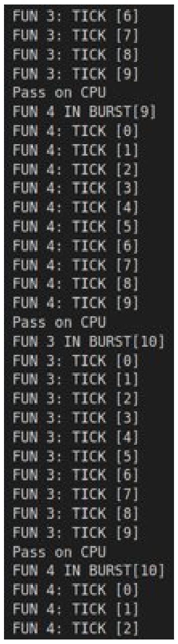
Testing (Main)

For the machine problem, I only use the provided test. In the beginning, there are four threads. Each thread executes the for loop and then passes the CPU to the next thread. After ten iterations, thread 1 and thread 2 terminate. Therefore, thread 3 and thread 4 pass the CPU to each other.



```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Pass on CPU
FUN 1 IN BURST[3]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
Pass on CPU
FUN 2 IN BURST[3]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Pass on CPU
FUN 3 IN BURST[3]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
```

In the beginning, there are 4 threads



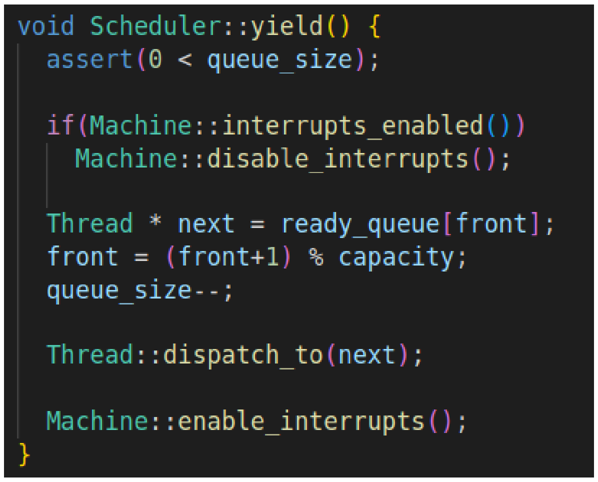
```
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
Pass on CPU
FUN 4 IN BURST[9]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Pass on CPU
FUN 3 IN BURST[10]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
Pass on CPU
FUN 4 IN BURST[10]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
```

After 10 iterations, thread 1 and thread 2 terminate.

Code Description (Option 1)

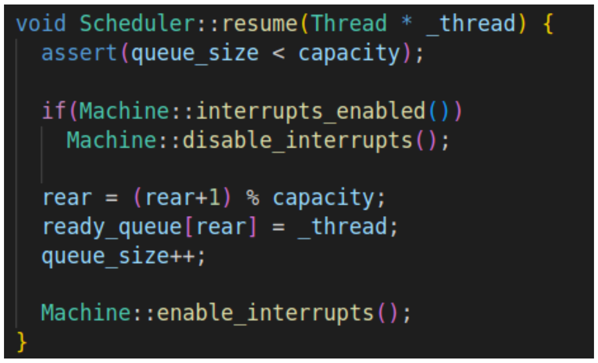
I changed scheduler.c and thread.c to handle the interrupts.

scheduler.c: yield : Disable interrupts before modifying the ready queue. Enable interrupts again after the modification.



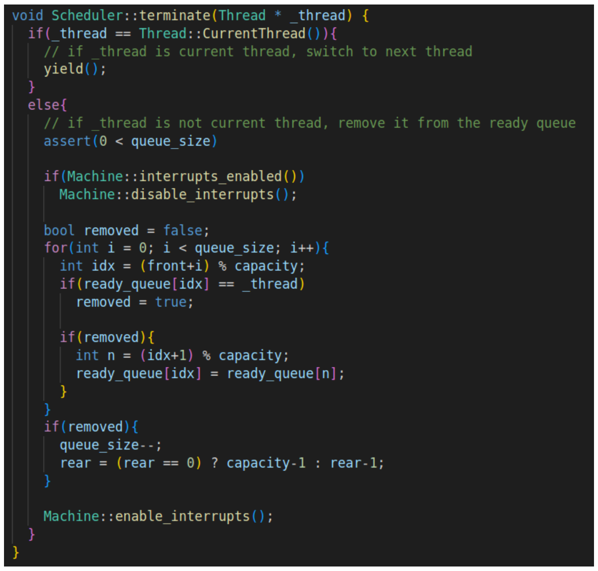
```
void Scheduler::yield() {
  assert(0 < queue_size);

  if(Machine::interrupts_enabled())
    Machine::disable_interrupts();

  Thread * next = ready_queue[front];
  front = (front+1) % capacity;
  queue_size--;

  Thread::dispatch_to(next);

  Machine::enable_interrupts();
}
```

scheduler.c: resume : Disable interrupts before modifying the ready queue. Enable interrupts again after the modification.



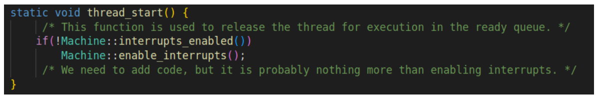
```
void Scheduler::resume(Thread * _thread) {
  assert(queue_size < capacity);

  if(Machine::interrupts_enabled())
    Machine::disable_interrupts();

  rear = (rear+1) % capacity;
  ready_queue[rear] = _thread;
  queue_size++;

  Machine::enable_interrupts();
}
```

scheduler.c: terminate : Disable interrupts before modifying the ready queue. Enable interrupts again after the modification.



```
void Scheduler::terminate(Thread * _thread) {
  if(_thread == Thread::CurrentThread()){
    // if _thread is current thread, switch to next thread
    yield();
  }
  else{
    // if _thread is not current thread, remove it from the ready queue
    assert(0 < queue_size)

    if(Machine::interrupts_enabled())
      Machine::disable_interrupts();

    bool removed = false;
    for(int i = 0; i < queue_size; i++){
      int idx = (front+i) % capacity;
      if(ready_queue[idx] == _thread)
        removed = true;

      if(removed){
        int n = (idx+1) % capacity;
        ready_queue[idx] = ready_queue[n];
      }
    }
    if(removed){
      queue_size--;
      rear = (rear == 0) ? capacity-1 : rear-1;
    }

    Machine::enable_interrupts();
  }
}
```

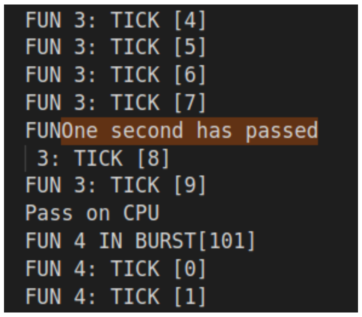
thread.c: thread_start : The function is called when a thread first time runs on the CPU. Just enable interrupts.



```
static void thread_start() {
      /* This function is used to release the thread for execution in the ready queue. */
    if(!Machine::interrupts_enabled())
        Machine::enable_interrupts();
      /* We need to add code, but it is probably nothing more than enabling interrupts. */
}
```

Testing (Option 1)

I only use the provided test. After enabling interrupts, when the timer fires, ”One second has passed” is printed.

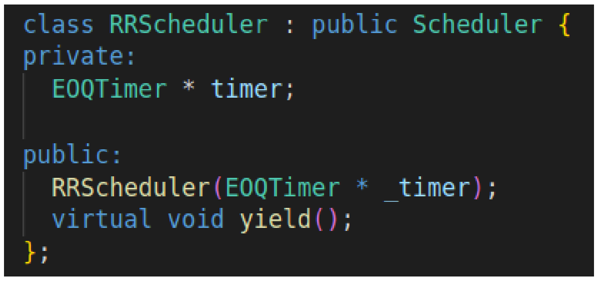


```
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUNOne second has passed
 3: TICK [8]
FUN 3: TICK [9]
Pass on CPU
FUN 4 IN BURST[101]
FUN 4: TICK [0]
FUN 4: TICK [1]
```

Code Description (Option 2)

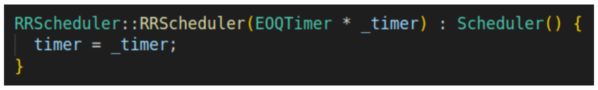
I changed scheduler.h, scheduler.c, simple_timer.h, and simple_timer.c to implement EOQTimer and RRScheduler.

scheduler.h: RRScheduler : The RRScheduler class is derived from the Scheduler class. The difference between these two class is that the RRScheduler class has a EOQ timer.



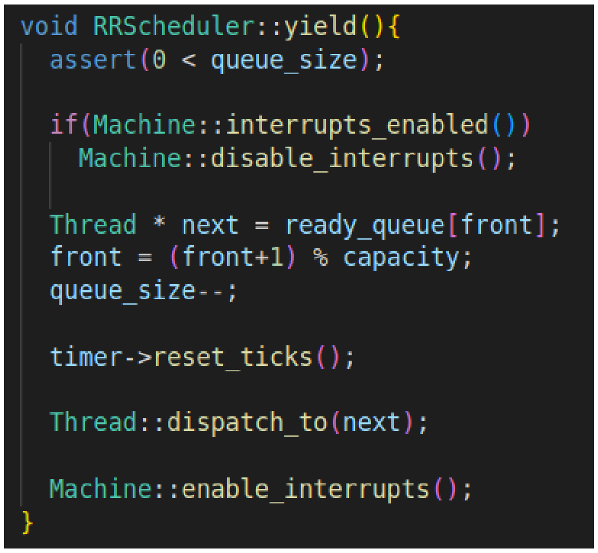
```
class RRScheduler : public Scheduler {
private:
  EOQTimer * timer;

public:
  RRScheduler(EOQTimer * _timer);
  virtual void yield();
};
```

scheduler.c: RRScheduler constructor : Call Scheduler's constructor and set the timer.



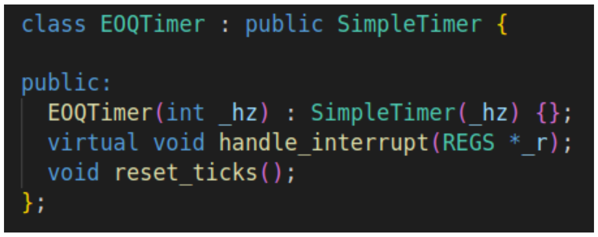
```
RRScheduler::RRScheduler(EOQTimer * _timer) : Scheduler() {
  timer = _timer;
}
```

scheduler.c: RRScheduler yield : The function is as same as the yield function in the Scheduler class, except that reset the timer before invoking the context switch.



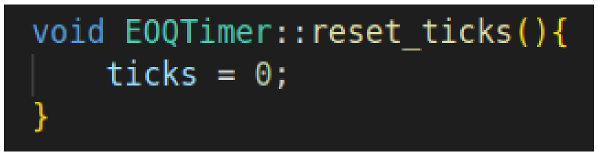
```
void RRScheduler::yield(){
  assert(0 < queue_size);

  if(Machine::interrupts_enabled())
    Machine::disable_interrupts();

  Thread * next = ready_queue[front];
  front = (front+1) % capacity;
  queue_size--;

  timer->reset_ticks();

  Thread::dispatch_to(next);

  Machine::enable_interrupts();
}
```

simple_timer.h: EOQTimer : The EOQTimer class is derived from the SimpleTimer.



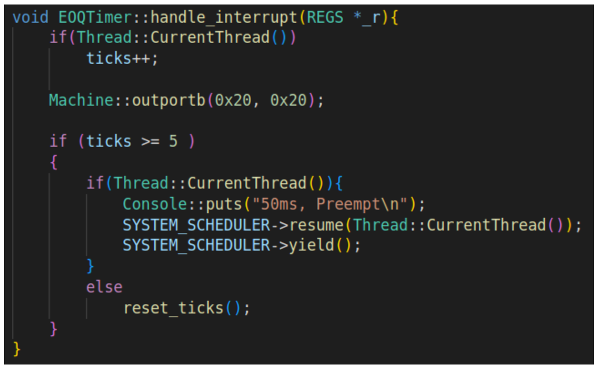
```
class EOQTimer : public SimpleTimer {

public:
  EOQTimer(int _hz) : SimpleTimer(_hz) {};
  virtual void handle_interrupt(REGS *_r);
  void reset_ticks();
};
```

simple_timer.c: EOQTimer reset_ticks : Set the ticks as zero.



```
void EOQTimer::reset_ticks(){
    ticks = 0;
}
```

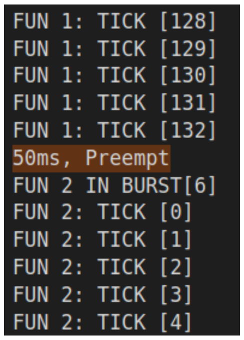
simple_timer.c: EOQTimer handle_interrupt : The function is called when the timer fires. If the timer fires and there is a thread running on the CPU, increase the ticks. If 50 ms passes (ticks > 5) and the current thread is still running on the CPU, put it back to the ready queue and give control to the next thread. If there is no running thread, just reset the timer. In addition, to handle interrupt correctly, an EOI message have to be sent to the master interrupt controller. It can be done by calling Machine::outportb().



```
void EOQTimer::handle_interrupt(REGS *_r){
    if(Thread::CurrentThread())
        ticks++;

    Machine::outportb(0x20, 0x20);

    if (ticks >= 5 )
    {
        if(Thread::CurrentThread()){
            Console::puts("50ms, Preempt\n");
            SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
            SYSTEM_SCHEDULER->yield();
        }
        else
            reset_ticks();
    }
}
```

Testing (Option 2)

I only use the provided test, but the SimpleTimer is replaced with the EOQTimer and the Scheduler is replaced with the RRScheduler. Furthermore, to test whether the round-robin scheduler works or not, I modified thread 1 and thread 3 to extend the running time. In each iteration, they print 200 sentences.



```
FUN 1: TICK [128]
FUN 1: TICK [129]
FUN 1: TICK [130]
FUN 1: TICK [131]
FUN 1: TICK [132]
50ms, Preempt
FUN 2 IN BURST[6]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
```