eOS Project3

2019-12172 이재원

June 28, 2024

Abstract

This report presents the implementation and enhancements of the eOS Project 3, focusing on task management and scheduling within an embedded operating system. Key features include the introduction of dynamic memory allocation for task control blocks (TCBs), the implementation of a priority-based O(1) scheduler, and the integration of alarm mechanisms to handle task waiting states. The ‘eos_create_task‘ function was augmented to initialize task priority and alarms, while the ‘eos_schedule‘ function was improved to support efficient context switching and task state management. The ‘eos_sleep‘ function sets alarms based on task ticks, triggering the ‘eos_wakeup_sleeping_task‘ handler upon timeout to transition tasks from WAITING to READY state. Additionally, the ‘eos_set_alarm‘ function was enhanced to maintain a priority-sorted alarm queue, ensuring timely task wake-ups. The results demonstrate the correct periodic execution of tasks, verified through printed outputs showing the expected task activation patterns.

1 Defined Struct

1.1 core/eos.h

```c
/* TCB (task control block) structure */
typedef struct tcb {
    // To be filled by students: Projects 2, 3, and 4
    int32u_t state;
    addr_t sp; // stack pointer
    int32u_t period;
    eos_alarm_t* alarm;
    _os_node_t* node;
} eos_tcb_t;
```

The alarm pointer was added to eos_tcb_t. Just like _os_node_t* node, alarm is defined as a pointer so dynamic memory allocation is needed. In eos_create_task(), alarm is dynamically allocated using malloc().

2 Functions

2.1 core/task.c

```c
int32u_t eos_create_task(eos_tcb_t *task, addr_t sblock_start, size_t sblock_size, void (*
    entry)(void *arg), void *arg, int32u_t priority){
    PRINT("task: 0x%x, priority: %d\n", (int32u_t)task, priority);
    // To be filled by students: Projects 2 and 3
    addr_t sp = _os_create_context(sblock_start, sblock_size, entry, arg);

    task->sp = sp;
    task->period = 0;

    task->node = (_os_node_t*)malloc(sizeof(_os_node_t));
    task->node->priority = priority;
    task->node->ptr_data = (void*) task;

    task->alarm = (eos_alarm_t*)malloc(sizeof(eos_alarm_t));
    task->alarm->timeout = 0;
    task->alarm->handler = NULL;
    task->alarm->arg = NULL;
    task->alarm->alarm_queue_node.ptr_data = (void*) task->alarm;
    task->alarm->alarm_queue_node.priority = priority;

    task->state = READY;
    _os_add_node_tail(&_os_ready_queue[task->node->priority], task->node);
    _os_set_ready(task->node->priority);

    return 0;
}
```

In this function, priority and alarm initialization was added to project2 function. By default, a task becomes non-periodic task by setting the priority 0. Also, the alarm was allocated and initialized all the variables inside.

```
void eos_schedule() {
    // To be filled by students: Projects 2 and 3

    if (_os_current_task != NULL){
        addr_t sp = _os_save_context();

        if (sp == NULL)
            return;

        _os_current_task->sp = sp;

        // store the current process in wait list if RUNNING (not WAITING)
        if(_os_current_task->state == RUNNING) {
            _os_current_task->state = READY;
            _os_add_node_tail(&_os_ready_queue[_os_current_task->node->priority],
    _os_current_task->node);
            _os_set_ready(_os_current_task->node->priority);
        }
    }

    // Select a next task from waiting list
    _os_node_t* node;
    int32u_t highest_priority = _os_get_highest_priority();
    node = _os_ready_queue[highest_priority];
    _os_remove_node(&_os_ready_queue[highest_priority], node);
    if(_os_ready_queue[highest_priority] == NULL)
        _os_unset_ready(highest_priority);

    // Change the selected task to _os_current_task
    _os_current_task = (eos_tcb_t *)node->ptr_data;
    _os_current_task->state = RUNNING;
    _os_restore_context(_os_current_task->sp);
}
```

There are 2 features added to eos_schedule().

First is the O(1) scheduler. Whenever a task is added to ready queue, _os_set_ready() is called. Also, if a node was removed and there is no remain node left with the same priority from the removed node, _os_unset_ready() is called.

Another feature different from project 2 is that when saving context, it is not always added to the ready queue. This is because tasks with WAITING state should be enqueued to ready queue after the waiting is over. The waiting queue is enqueued at _os_wakeup_sleeping_task().

```
void eos_sleep(int32u_t tick) {
    // To be filled by students: Project 3
    eos_counter_t* system_timer = eos_get_system_timer();
    _os_current_task->state = WAITING;
    int32u_t wait_time = (tick == 0) ? _os_current_task->period : 0;
    eos_set_alarm(
        system_timer,
        _os_current_task->alarm,
        _os_current_task->alarm->timeout + wait_time + tick, // wait time is 0 + tick if
    tick > 0
        _os_wakeup_sleeping_task,
        _os_current_task
    );
    eos_schedule();
}
```

This function sets an alarm according to tick and the priority of the task. When setting the alarm, _os_wakeup_sleeping_task() is passed as a handler, so that when the alarm rings, the handler code is executed.

```c
void _os_wakeup_sleeping_task(void *arg) {
    // To be filled by students: Project 3
    eos_tcb_t *task = (eos_tcb_t *) arg;

    // Add task to waiting queue
    task->state = READY;
    _os_add_node_tail(&_os_ready_queue[task->node->priority], task->node);
    _os_set_ready(task->node->priority);

    eos_schedule();
}
```

This is the handler function that is sent to the alarm. When the alarm is called, it means that the waiting is done. Therefore, it changes the statee to READY and enqueue to the ready queue.

2.2 core/timer.c

```c
void eos_set_alarm(eos_counter_t* counter, eos_alarm_t* alarm, int32u_t timeout, void (*
    entry)(void *arg), void *arg){
    // To be filled by students: Project 3
    _os_remove_node(&counter->alarm_queue, &alarm->alarm_queue_node);

    if(timeout == 0 || entry == NULL) {
        return;
    }

    alarm->timeout = timeout;
    alarm->handler = entry;
    alarm->arg = arg;

    // Enqueue by the increasing order of 64*timeout + priority
    // This makes the queue sorted by timeout.
    // If the timeout is same, it is then sorted by priority.
    alarm->alarm_queue_node.priority
        = (LOWEST_PRIORITY + 1) * timeout + ((eos_tcb_t*)arg)->node->priority;

    _os_add_node_priority(&counter->alarm_queue, &alarm->alarm_queue_node);
}
```

This function can be used in 2 ways.

First is to delete the alarm in the queue. If the timeout is 0 or entry is NULL, it just removes the alarm from alarm queue and returns.

Second is to enqueue the alarm in the queue. Enqueue is done by sorting by the priority. There are 2 priorities implemented in this code. The first priority is the timeout. Second priority is the task priority. The second priority is used when the first priority is the same. As the second priority has range of 0 LOWEST_PRIORITY, an unified priority is calculated by multiplying (LOWEST_PRIORITY + 1) to the timeout and adding the second priority. In this way, multilevel priority queue is implemented.

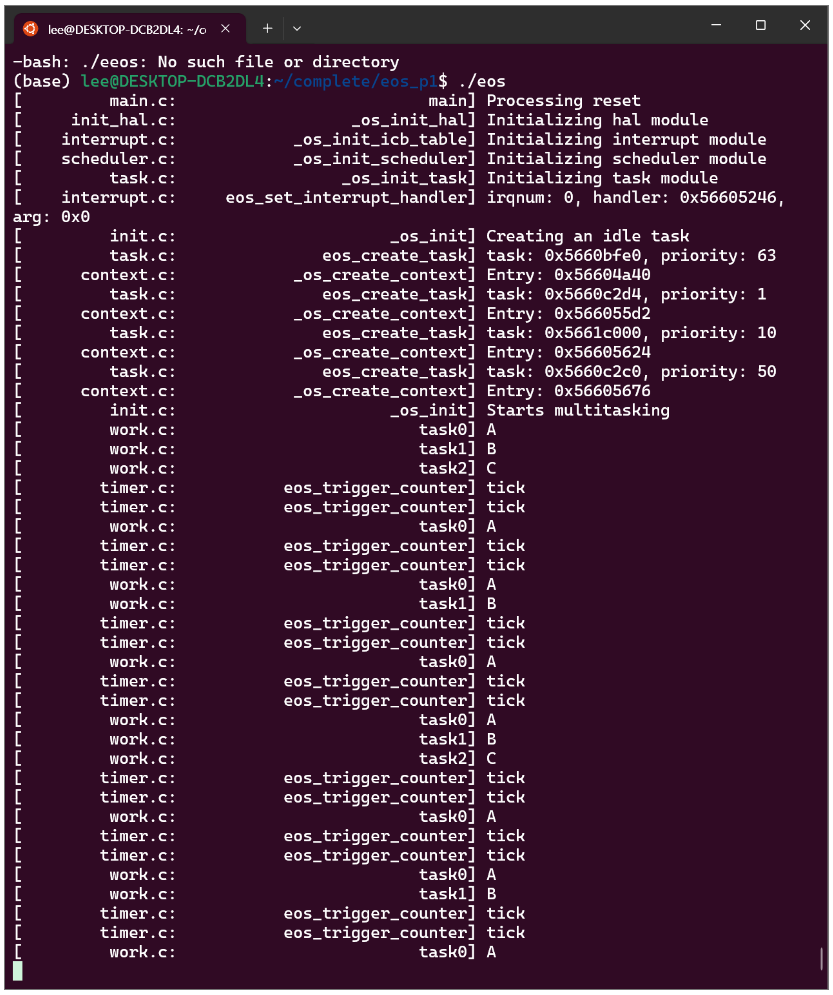
As the node is sorted when enqueing, the dequeue becomes easy because you just have to check the head of the queue.

```c
void eos_trigger_counter(eos_counter_t* counter){
    PRINT("tick\n");
    // To be filled by students: Project 3
    counter->tick++;

    // There could be more than 1 task at the queue,
    // so interation is needed
    while(counter->alarm_queue != NULL) {
        _os_node_t * alarm_node = counter->alarm_queue;

        eos_alarm_t* alarm = (eos_alarm_t*)(alarm_node->ptr_data);
        if(alarm->timeout > counter->tick)
            break;

        // Remove alarm from queue
        eos_counter_t* system_timer = eos_get_system_timer();
        eos_set_alarm(
            system_timer,
            alarm,
            0, // This means remove
            NULL,
            NULL
        );

        // call handler
        alarm->handler(alarm->arg);
    }

}
```

This function check the head of alarm queue and dequeues the node if needed. As the queue is sorted when enqueue, the function only cheks the head. However, there may be a lot of nodes to be dequeued, so while loop is used to check the remaining heads.

The dequeue is implemented by eos_set_alarm() by changing the arguments.

3 Results



```
-bash: ./eeos: No such file or directory
(base) lee@DESKTOP-DCB2DL4:~/complete/eos_p1$ ./eos
[         main.c:                          main] Processing reset
[     init_hal.c:                  _os_init_hal] Initializing hal module
[    interrupt.c:            _os_init_icb_table] Initializing interrupt module
[    scheduler.c:            _os_init_scheduler] Initializing scheduler module
[         task.c:                 _os_init_task] Initializing task module
[    interrupt.c:     eos_set_interrupt_handler] irqnum: 0, handler: 0x56605246,
arg: 0x0
[         init.c:                      _os_init] Creating an idle task
[         task.c:               eos_create_task] task: 0x5660bfe0, priority: 63
[      context.c:            _os_create_context] Entry: 0x56604a40
[         task.c:               eos_create_task] task: 0x5660c2d4, priority: 1
[      context.c:            _os_create_context] Entry: 0x566055d2
[         task.c:               eos_create_task] task: 0x5661c000, priority: 10
[      context.c:            _os_create_context] Entry: 0x56605624
[         task.c:               eos_create_task] task: 0x5660c2c0, priority: 50
[      context.c:            _os_create_context] Entry: 0x56605676
[         init.c:                      _os_init] Starts multitasking
[         work.c:                         task0] A
[         work.c:                         task1] B
[         work.c:                         task2] C
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[         work.c:                         task1] B
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[         work.c:                         task1] B
[         work.c:                         task2] C
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
[         work.c:                         task1] B
[        timer.c:           eos_trigger_counter] tick
[        timer.c:           eos_trigger_counter] tick
[         work.c:                         task0] A
```

Figure 1: The result of eos executable file. A, B, C and tick is printed during execution.

In Figure 1, the user code is executed.

As task0 has period of 2, A is printed every other ticks. task1 has period of 4, so B is printed every 4 ticks. C is printed every 8 ticks.