

eOS Project4

2019-12172 이재원

June 5, 2024

Abstract

This report presents the design and implementation of synchronization mechanisms using semaphores and message queues within the eOS project. We define the structures and functions required to initialize, acquire, and release semaphores and message queues. The `eos_semaphore_t` struct is utilized to manage resource counting and waiting queues, while `eos_mqueue_t` struct handles message passing with a circular queue and synchronization through semaphores. The functions `eos_init_semaphore`, `eos_acquire_semaphore`, and `eos_release_semaphore` are developed to manage semaphore operations. Similarly, `eos_init_mqueue`, `eos_send_message`, and `eos_receive_message` are implemented for message queue operations. The system is tested to ensure proper synchronization and message handling, and the results demonstrate the effectiveness of our implementation in managing concurrent tasks. We provide visual results showing the correct operation of the system under various conditions, including resource contention and queue capacity management.

1 Defined Struct

1.1 core/eos.h

```
1 typedef struct eos_semaphore {
2     // To be filled by students: Project 4
3     int32u_t count;
4     _os_node_t* wait_queue;
5     int8u_t queue_type;
6 } eos_semaphore_t;
```

In this struct, `count` is an unsigned variable that refers to the original meaning of semaphore. `wait_queue` variable is queue used for waiting when `P()` instruction is called. In this project, `P()` is `eos_acquire_semaphore()`. `queue_type` determines the waiting queue is FIFO or priority based.

```
1 typedef struct eos_mqueue {
2     // To be filled by students: Project 4
3     int16u_t queue_size;
4     int8u_t msg_size;
5     void *queue_start;
6     void *front;
7     void *rear;
8     int8u_t queue_type;
9     eos_semaphore_t putsem;
10    eos_semaphore_t getsem;
11 } eos_mqueue_t;
```

`queue_size` is the number of maximum message in a queue. `msg_size` is the byte of data in a message. `queue_start` is the start address of queue. The end address can be calculated because the length of queue is `queue_size * msg_size`. `front` and `rear` is the pointer for circular queue. `queue_type` is FIFO or priority based. `putsem` is a semaphore that checks if the message can be sent. `getsem` is a semaphore that checks if the message can be received.

2 Functions

2.1 core/sync.c

```

1 void eos_init_semaphore(eos_semaphore_t *sem, int32u_t initial_count, int8u_t queue_type)
2 {
3     // To be filled by students: Project 4
4     sem->count = initial_count;
5     sem->wait_queue = NULL;
6     sem->queue_type = queue_type;
7 }

```

This function initialized every member of semaphore structure with input parameters.

```

1 int32u_t eos_acquire_semaphore(eos_semaphore_t *sem, int32s_t timeout)
2 {
3     // To be filled by students: Project 4
4     hal_disable_interrupt();
5
6     // if the resource is available
7     if (sem->count > 0) {
8         sem->count--;
9         hal_enable_interrupt();
10        return 1;
11    }
12
13    // if timeout is negative
14    if (timeout < 0) {
15        hal_enable_interrupt();
16        return 0;
17    }
18
19    // Original semaphore: waiting until resource is available
20    if (timeout == 0) {
21        // execute until resource is available
22        while (1) {
23            // push current task into waiting queue, and yield a CPU
24            eos_tcb_t* cur_task = eos_get_current_task();
25            cur_task->state = WAITING; // Not call eos_sleep() because alarm is not set
26
27            if (sem->queue_type == FIFO)
28                _os_add_node_tail(&(sem->wait_queue), (cur_task->node));
29            else if (sem->queue_type == PRIORITY)
30                _os_add_node_priority(&(sem->wait_queue), (cur_task->node));
31
32            hal_enable_interrupt();
33            eos_schedule();
34
35            // after waking up, check if the resource is available
36            hal_disable_interrupt();
37            if (sem->count > 0) {
38                sem->count--;
39                hal_enable_interrupt();
40                return 1;
41            }
42        }
43    }
44
45    // if timeout is positive, wait until timeout
46    while (1) {
47        hal_enable_interrupt();
48        // call eos_sleep() for setting alarm
49        eos_sleep(timeout);
50        hal_disable_interrupt();
51        if (sem->count > 0) {
52            sem->count--;
53            hal_enable_interrupt();
54            return 1;
55        }
56    }
57 }

```

This function serves different by count and timeout parameters.

If $count > 0$, this means there are available resources, so it simply decrease the count value and returns.

If $count = 0$, this means there is no available resource currently, so it needs to wait or fail. If timeout is -1,

it just fails. When timeout is 0, it waits until another task wakes it up. If *timeout* > 0, it waits for timeout and checks again. This is implemented by eos.sleep().

```

1 void eos_release_semaphore(eos_semaphore_t *sem)
2 {
3     // To be filled by students: Project 4
4     sem->count++;
5     if (sem->wait_queue) {
6         //PRINT("delete alarm and wake up\n")
7         eos_tcb_t* wake_task = (eos_tcb_t*)(sem->wait_queue->ptr_data);
8         eos_set_alarm(eos_get_system_timer(), wake_task->alarm, 0, NULL, NULL);
9         _os_remove_node(&(sem->wait_queue), sem->wait_queue);
10        _os_wakeup_sleeping_task((void*)wake_task);
11    }
12 }

```

This increases the semaphore and waits another task in waiting queue.

2.2 core/comm.c

```

1 void eos_init_mqueue(eos_mqueue_t *mq, void *queue_start, int16u_t queue_size, int8u_t
   msg_size, int8u_t queue_type){
2     // To be filled by students: Project 4
3     // PRINT("message queue: 0x%x at 0x%x || queue_size: %d, msg_size: %d\n", mq,
   queue_start, queue_size, msg_size)
4     mq->queue_size = queue_size;
5     mq->msg_size = msg_size;
6
7     mq->queue_start = queue_start;
8     mq->front = queue_start;
9     mq->rear = queue_start;
10
11    mq->queue_type = queue_type;
12    eos_init_semaphore(&(mq->putsem), queue_size, queue_type);
13    eos_init_semaphore(&(mq->getsem), 0, queue_type);
14 }

```

This function initializes all the member variables in the mqueue struct.

The front and rear pointer should be initialized with the same address because it needs to be empty. It doesn't need to be the start pointer of the queue, but for convenience, it is initialized as the start pointer

```

1 int8u_t eos_send_message(eos_mqueue_t *mq, void *message, int32s_t timeout)
2 {
3     // To be filled by students: Project 4
4     if (!eos_acquire_semaphore(&(mq->putsem), timeout)) return;
5
6     // read the message by 1 byte
7     char* msg = (char*)message;
8
9     for (int i = 0; i < mq->msg_size; i++) {
10        // copy the message to rear
11        *(char*)(mq->rear) = msg[i];
12
13        // Change rear by increasing and check the
14        // pointer is in range of queue by dividing with queue size
15        mq->rear = (mq->rear + 1 - mq->queue_start)
16        % (mq->queue_size * mq->msg_size + 1) + mq->queue_start;
17    }
18    // release semaphore
19    eos_release_semaphore(&(mq->getsem);
20 }

```

This function implements task with 5 steps.

1. putsem to check if it is available.
2. If failed, return.
3. If succeed, copy the message to the rear of queue.

4. Change rear by increasing and check the pointer is in range of queue by dividing with queue size.
5. getsem

```

1 int8u_t eos_receive_message(eos_mqueue_t *mq, void *message, int32s_t timeout)
2 {
3     // To be filled by students: Project 4
4     if (!eos_acquire_semaphore(&(mq->getsem), timeout)) return;
5
6     // read the message by 1 byte
7     char* msg = (char*)message;
8
9     for (int i = 0; i < mq->msg_size; i++) {
10        // get the message from the front part of queue
11        msg[i] = *((char*)(mq->front));
12
13        // Change front by increasing and check the
14        // pointer is in range of queue by dividing with queue size
15        mq->front = (mq->front + 1 - mq->queue_start)
16                % (mq->queue_size * mq->msg_size + 1) + mq->queue_start;
17    }
18    // release semaphore
19    eos_release_semaphore(&(mq->putsem));
20 }

```

This function implements task with 5 steps.

1. putsem to check if it is available.
2. If failed, return.
3. If succeed, copy the message from the front of queue.
4. Change front by increasing and check the pointer is in range of queue by dividing with queue size.
5. getsem

3 Results

In Figure 1, the user code is executed.

First the task tries to receive a message. However, as there are no messages sent, the receiving task is blocked by a semaphore until message is sent. This occurs before tick 1.

After receiving message, the receiving message task is executed by a period of 4 or 6. However, the sending task is executed by period of 2, so sending task is executed more frequently than receiving tasks. Therefore, after tick 1, there is no case that the message queues are empty. Also, this means that there could be a case that the message queues can be full.

This phenomenon can be seen at figure 2, where no task is executed right after tick 22.

```

(base) lee@DESKTOP-DCB2DL4: ~/eos $ ./eos
main.c:      [main] Processing reset
init_hal.c:  _os_init_hal[] Initializing hal module
interrupt.c: _os_init_tick_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: 0x5660826d, arg: 0x0
init.c:      _os_init[] Creating an idle task
task.c:      eos_create_task[] task: 0x5660efef0, priority: 63
context.c:   _os_create_context[] Entry: 0x56607a40
task.c:      eos_create_task[] task: 0x5660f380, priority: 50
context.c:   _os_create_context[] Entry: 0x56608733
task.c:      eos_create_task[] task: 0x5661f480, priority: 10
context.c:   _os_create_context[] Entry: 0x56608696
task.c:      eos_create_task[] task: 0x5660f36c, priority: 20
context.c:   _os_create_context[] Entry: 0x566085f9
init.c:      _os_init[] Starts multitasking
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task1[] receive message from mq1
work.c:      sender_task[] send message to mq0
work.c:      receiver_task0[] received message: xy
work.c:      sender_task[] send message to mq1
work.c:      receiver_task1[] received message: xy
timer.c:      eos_trigger_counter[] tick 1
timer.c:      eos_trigger_counter[] tick 2
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 3
timer.c:      eos_trigger_counter[] tick 4
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 5
timer.c:      eos_trigger_counter[] tick 6
work.c:      receiver_task1[] receive message from mq1
work.c:      receiver_task1[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 7
timer.c:      eos_trigger_counter[] tick 8
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 9
timer.c:      eos_trigger_counter[] tick 10
work.c:      sender_task[] send message to mq0

```

Figure 1: The result of eos executable file. For checking the result easier the number of tick is also printed. In the code submitted, the number of tick is not printed.

```

timer.c:      eos_trigger_counter[] tick 10
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 11
timer.c:      eos_trigger_counter[] tick 12
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      receiver_task1[] receive message from mq1
work.c:      receiver_task1[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 13
timer.c:      eos_trigger_counter[] tick 14
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 15
timer.c:      eos_trigger_counter[] tick 16
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 17
timer.c:      eos_trigger_counter[] tick 18
work.c:      receiver_task1[] receive message from mq1
work.c:      receiver_task1[] received message: xy
timer.c:      eos_trigger_counter[] tick 19
timer.c:      eos_trigger_counter[] tick 20
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 21
timer.c:      eos_trigger_counter[] tick 22
timer.c:      eos_trigger_counter[] tick 23
timer.c:      eos_trigger_counter[] tick 24
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
work.c:      receiver_task1[] receive message from mq1
work.c:      receiver_task1[] received message: xy
timer.c:      eos_trigger_counter[] tick 25
timer.c:      eos_trigger_counter[] tick 26
work.c:      sender_task[] send message to mq0
work.c:      sender_task[] send message to mq1
timer.c:      eos_trigger_counter[] tick 27
timer.c:      eos_trigger_counter[] tick 28
work.c:      receiver_task0[] receive message from mq0
work.c:      receiver_task0[] received message: xy
timer.c:      eos_trigger_counter[] tick 29

```

Figure 2: The result of eos executable file. Sending message task is not executed right after tick 22 because the message queue is full.