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

```c
typedef struct eos_semaphore {
    // To be filled by students: Project 4
    int32u_t count;
    _os_node_t* wait_queue;
    int8u_t queue_type;
} 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.

```c
typedef struct eos_mqueue {
    // To be filled by students: Project 4
    int16u_t queue_size;
    int8u_t msg_size;
    void *queue_start;
    void *front;
    void *rear;
    int8u_t queue_type;
    eos_semaphore_t putsem;
    eos_semaphore_t getsem;
} 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

```c
void eos_init_semaphore(eos_semaphore_t *sem, int32u_t initial_count, int8u_t queue_type)
{
  // To be filled by students: Project 4
  sem->count = initial_count;
  sem->wait_queue = NULL;
  sem-> queue_type = queue_type;
}
```

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

```c
int32u_t eos_acquire_semaphore(eos_semaphore_t *sem, int32s_t timeout)
{
  // To be filled by students: Project 4
  hal_disable_interrupt();

  // if the resource is available
  if (sem->count > 0) {
    sem->count--;
    hal_enable_interrupt();
    return 1;
  }

  // if timeout is negative
  if (timeout < 0) {
    hal_enable_interrupt();
    return 0;
  }

  // Original semaphore: waiting until resource is available
  if (timeout == 0) {
    // execute until reesource is available
    while (1) {
      // push current task into waiting queue, and yield a CPU
      eos_tcb_t* cur_task = eos_get_current_task();
      cur_task->state = WAITING; // Not call eos_sleep() because alarm is not set

      if (sem->queue_type == FIFO)
        _os_add_node_tail(&(sem->wait_queue), (cur_task->node));
      else if (sem->queue_type == PRIORITY)
        _os_add_node_priority(&(sem->wait_queue), (cur_task->node));

      hal_enable_interrupt();
      eos_schedule();

      // after waking up, check if the resource is available
      hal_disable_interrupt();
      if (sem->count > 0) {
        sem->count--;
        hal_enable_interrupt();
        return 1;
      }
    }
  }

  // if timeout is positive, wait until timeout
  while (1) {
    hal_enable_interrupt();
    // call eos_sleep() for setting alarm
    eos_sleep(timeout);
    hal_disable_interrupt();
    if (sem->count > 0) {
      sem->count--;
      hal_enable_interrupt();
      return 1;
    }
  }
}
```

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().

```
void eos_release_semaphore(eos_semaphore_t *sem)
{
  // To be filled by students: Project 4
  sem->count++;
  if (sem->wait_queue) {
    //PRINT("delete alarm and wake up\n")
    eos_tcb_t* wake_task = (eos_tcb_t*)(sem->wait_queue->ptr_data);
    eos_set_alarm(eos_get_system_timer(), wake_task->alarm, 0, NULL, NULL);
    _os_remove_node(&(sem->wait_queue), sem->wait_queue);
    _os_wakeup_sleeping_task((void*)wake_task);
  }
}
```

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

2.2 core/comm.c

```
void eos_init_mqueue(eos_mqueue_t *mq, void *queue_start, int16u_t queue_size, int8u_t
    msg_size, int8u_t queue_type){
    // To be filled by students: Project 4
    // PRINT("message queue: 0x%x at 0x%x || queue_size: %d, msg_size: %d\n", mq,
    queue_start, queue_size, msg_size)
    mq->queue_size = queue_size;
    mq->msg_size = msg_size;

    mq->queue_start = queue_start;
    mq->front = queue_start;
    mq->rear = queue_start;

    mq->queue_type = queue_type;
    eos_init_semaphore(&(mq->putsem), queue_size, queue_type);
    eos_init_semaphore(&(mq->getsem), 0, queue_type);
}
```

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

```
int8u_t eos_send_message(eos_mqueue_t *mq, void *message, int32s_t timeout)
{
    // To be filled by students: Project 4
    if (!eos_acquire_semaphore((&mq->putsem), timeout))  return;

    // read the message by 1 byte
    char* msg = (char*)message;

    for (int i = 0; i < mq->msg_size; i++) {
        // copy the message to rear
        *(char*)(mq->rear) = msg[i];

        // Change rear by increasing and check the
        // pointer is in range of queue by dividing with queue size
        mq->rear = (mq->rear + 1 - mq->queue_start)
            % (mq->queue_size * mq->msg_size + 1) + mq->queue_start;
    }
    // release semaphore
    eos_release_semaphore(&mq->getsem);
}
```

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

```
int8u_t eos_receive_message(eos_mqueue_t *mq, void *message, int32s_t timeout)
{
    // To be filled by students: Project 4
    if (!eos_acquire_semaphore(&(mq->getsem), timeout)) return;

    // read the message by 1 byte
    char* msg = (char*)message;

    for (int i =0; i < mq->msg_size; i++) {
        // get the message from the front part of queue
        msg[i] = *((char*)(mq->front));

        // Change front by increasing and check the
        // pointer is in range of queue by dividing with queue size
        mq->front = (mq->front + 1 - mq->queue_start)
            % (mq->queue_size * mq->msg_size + 1) + mq->queue_start;
    }
    // release semaphore
    eos_release_semaphore(&mq->putsem);
}
```

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.

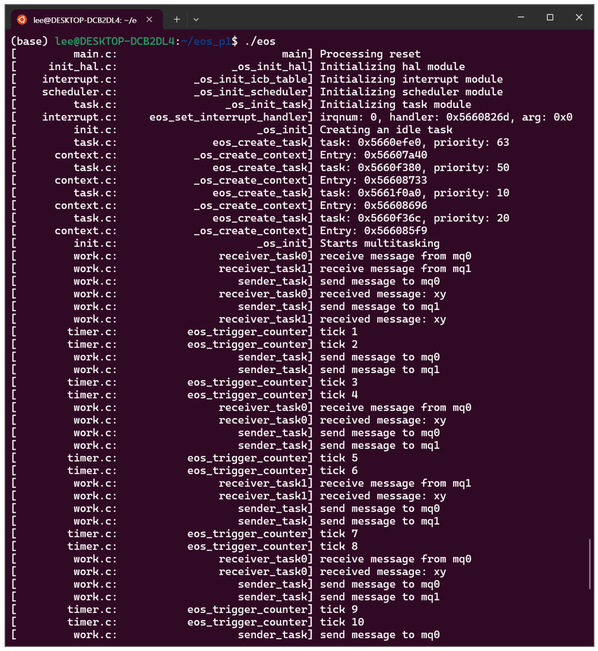

Figure 1: The result of eos executable file. For checking the result easier the number of tick is also printed. In the code submitted, the number of tick is not printed.

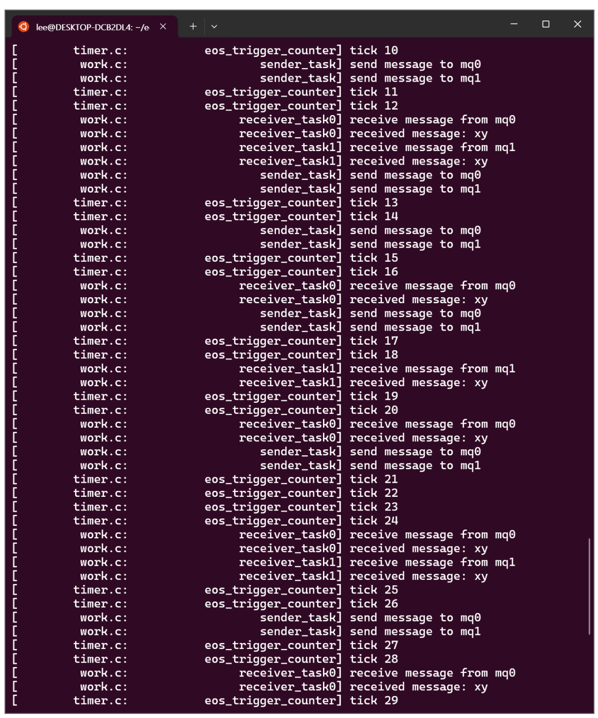

Figure 2: The result of eos executable file. Sending message task is not executed right after tick 22 because the message queue is full.