LG 부트캠프 7기 프로젝트반

프로젝트 결과보고서

Real-Time Operating Snake

RTOS / 1반 1팀 / 0xF

김정진, 김소연, 심훈, 이성훈

목차구성 CONTENTS COMPOSITION

1. 주제 및 결과 요약

- **주제**
  - RTOS(Real time OS) 기능을 직접 구현하고 해당 기능을 사용하여 Demo application 제작함

- **목표**
  - RTOS 기능 구현
    - ✓ Task & Scheduler
    - ✓ Blocked based delay
    - ✓ Signaling
    - ✓ Queue
    - ✓ Mutex
  - Demo application
    - ✓ Snake 게임 제작

- **결과**
  - 구현한 RTOS 기능으로 Snake 게임 제작
    - ✓ 각 기능 별 Task 4개를 사용
      - ✓ Operating, Game mode change, UART, Display task
    - ✓ 앞서 구현한 모든 RTOS 기능을 사용
      - ✓ Task & Scheduler, Blocked based delay, Signaling, Queue, Mutex

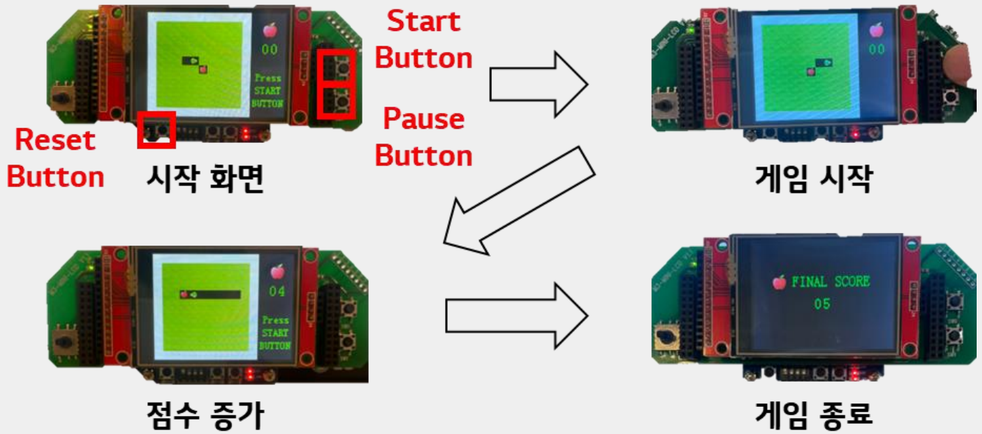
2. 개발 목표 및 개발 결과

- **개발 목표**
  - ➢ 최소 구현 spec을 만족하는 RTOS(Real time OS) 기능 구현
  - ➢ Demo application 제작에 필요한 최소 구현 spec 이상의 RTOS 기능 구현
  - ➢ 제공된 board에서 제공하는 peripheral(LCD, Joy key 등)을 이용한 snake 게임 제작

- **개발 결과**
  - ➢ Task 별로 아래와 같은 기능을 갖는 Snake 게임 구현

Task 1	Task 2	Task 3	Task 4
Operating ✓ Snake Key 입력 ✓ Snake 위치 계산 ✓ Apple 위치 계산 ✓ Score update ✓ 게임 over 조건	**Game mode** ✓ Game Start ✓ Game Pause ✓ Game Over	**UART** ✓ 난이도 조절 ✓ Cheat 설정 ✓ 키보드 조작	**Display** ✓ Snake ✓ Apple ✓ Score ✓ Background

  - ➢ 제작 게임 demo



3. 핵심 기술

- **Task & Scheduler**
  - ➢ Task 생성 및 context switching에 대한 critical section 관리
    - ✓ TCB(Task Control Block) structure를 이용하여 task의 stack 위치, 우선순위, state를 저장 및 관리
    - ✓ Context Switching 시 다른 interrupt가 발생하지 않도록 BASEPRI를 이용해 방지
      이때, fault는 발생할 수 있도록 우선순위 3 이상의 exception만 발생하지 않도록 설정

```
void OS_Tick(void) {
    __set_BASEPRI(0x30);
```

```
typedef struct _tcb{
    unsigned long* top_of_stack;
    int no_task;
    int prio;
    int state;
```

TCB structure의 기본 구조

  - ➢ Task 우선순위 및 task state(Ready, Running, Blocked) 기반 scheduling 함수 구현
    - ✓ Systick interrupt를 이용해 일정 시간마다 task의 우선순위 비교 및 scheduling 진행
    - ✓ Max heap 기반 priority queue를 이용하여 실행할 task를 결정
    - ✓ Ready queue에서 우선순위가 가장 높은 task를 next task로 설정

```
void OS_Scheduler(void)
{
    next_tcb = pq_top(&ready_queue);
```

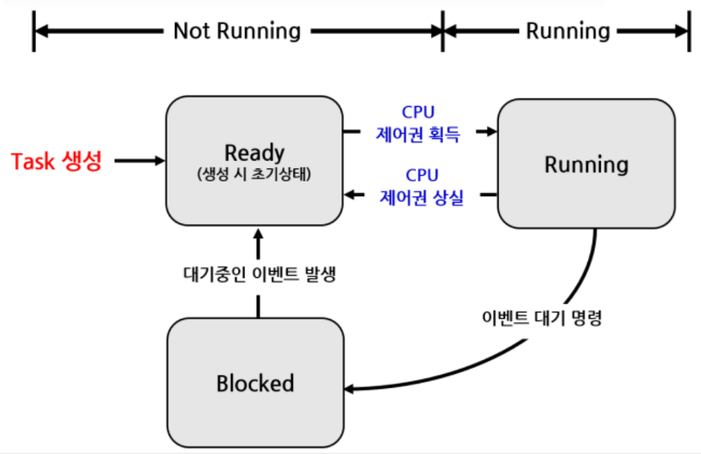
    - ✓ Ready queue에서 우선순위가 가장 높은 task를 pop하고 state를 RUNNING으로 설정

```
    if (next_tcb != NULL && next_tcb != current_tcb) {
        pq_pop(&ready_queue, pq_compare_ready);
        next_tcb->state = STATE_RUNNING;
```

    - ✓ Current task의 state를 ready로 변경하며 running할 task를 current task로 변경

```
        if (current_tcb->state == STATE_RUNNING){
            current_tcb->state = STATE_READY;
            current_tcb->timestamp = system_tick;
            pq_push(&ready_queue, current_tcb, pq_compare_ready);
        }

        current_tcb = next_tcb;
```



Task state block diagram

3. 핵심 기술

- **Blocked based delay**
  - ➢ TCB 구조체에 timestamp 및 delay_until 함수를 이용하여 delay 구현
    - ✓ TCB 구조체에 timestamp 및 delay_until 함수를 추가

```
typedef struct _tcb{
    long long int timestamp;
    long long int delay_until;
```

Delay 구현을 위한 parameter 추가

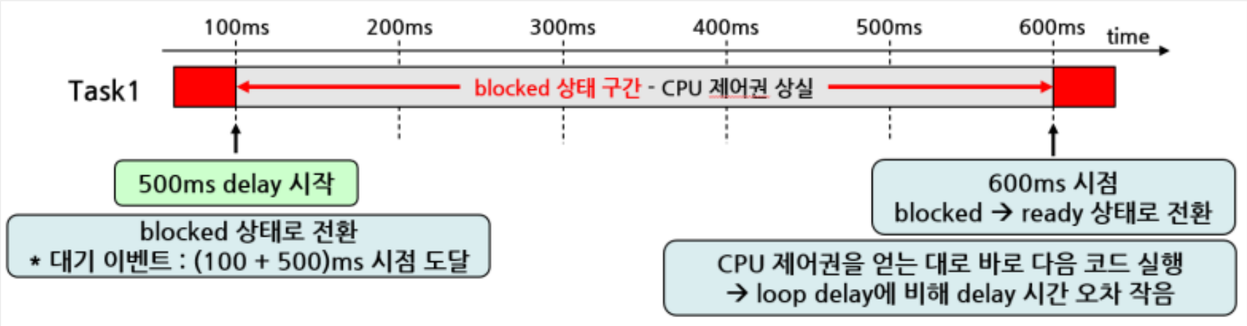
- ✓ Delay를 발생시킨 경우 ready queue에서 삭제하고 delay_until에 delay시켜야 할 시간을 저장하고 blocked queue에 task 저장

```
void OS_Block_Current_Task(int delay) {
    pq_remove(&ready_queue, current_tcb, pq_compare_ready);
    current_tcb -> delay_until = system_tick + delay;

    pq_push(&blocked_queue, current_tcb, pq_compare_delay);
```

- ✓ systick handler가 발생하는 경우 system_tick을 증가시키고 delay_until이 system_tick보다 작은 경우 delay가 종료됨을 확인

```
system_tick += interrupt_period;  // 시스템 타임스탬프 증가
while (blocked_queue.size > 0 && pq_top(&blocked_queue)->delay_until <= system_tick)
```



Blocked based delay concept

3. 핵심 기술

- **Signaling & Queue**
  - TCB 구조체에서 signal_flag를 통한 signaling 구현
    - ✓ signal_flag를 통해 signal을 정상적으로 수신했는지 확인하고, 정상적으로 수신한 경우 결과를 읽음
    - ✓ Blocked based delay 기능을 이용해 수신하고자 하는 시간동안 task는 blocked 상태가 되며 해당 시간에 다른 task가 cpu를 할당받아 사용 가능함

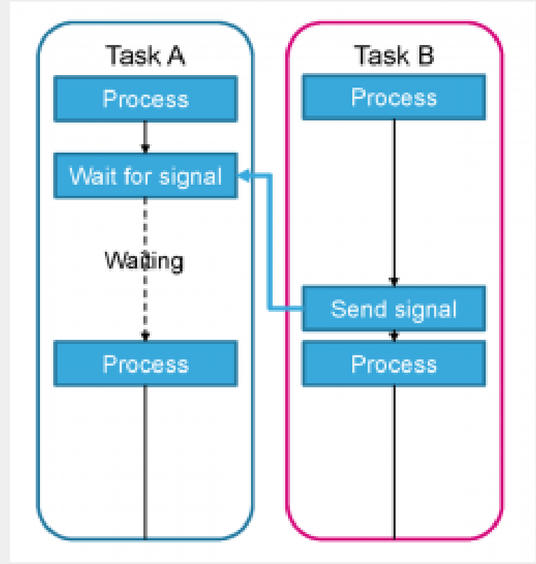
```
int OS_Signal_Wait(int queue_no, void* buffer, int buffer_size, int timeout) {
    if (current_tcb -> signal_flag != SIGNAL_RECEIVED){
        current_tcb -> signal_flag = SIGNAL_WAIT;
        OS_Block_Current_Task(timeout);
```

  - Queue 구현
    - ✓ Queue를 이용하여 입력된 data를 순차적으로 저장하고 출력함
    - ✓ Queue의 저장될 data size를 parameter로 하여 다양한 data type을 지원함
    - ✓ signal_flag를 이용해 예외처리(timeout, queue에 접근 task 권한 확인, queue의 타입확인 등)를 진행하고, 정상적으로 수신한 경우 결과를 읽음

```
if(wait_result == SIGNAL_TIMEOUT) {
    Uart_Printf("Signal Timeout\n");
}
else if(wait_result == SIGNAL_NO_PERMISSION) {
    Uart_Printf("Task 3 didn't create Queue\n");
}
else if(wait_result == SIGNAL_QUEUE_EMPTY) {
    Uart_Printf("Queue is empty\n");
}
else if(wait_result == SIGNAL_WRONG_DATA_TYPE) {
    Uart_Printf("Data Type is wrong\n");
}
else if(wait_result == SIGNAL_NO_ERROR){
    Uart_Printf("Received data is : %d\n", received_data);
```

```
typedef struct Queue {
    Node* front;
    Node* rear;
    int size;
    int data_size;
    Node* free_nodes;
    int no_task;
} Queue;
```

Queue 구조



Siganling concept

3. 핵심 기술

- **Mutex**
  - ➢ Waiting queue를 활용한 mutex 구현
    - ✓ Mutex를 선점하면 waiting queue에 등록하고 해당 mutex를 Lock

```
int Take_Mutex(int mutex_id, int task_related) {
        mutex_list[mutex_id]->locked = 1;
            enqueue(&mutex_list[mutex_id]->waiting_queue, &task_no);
```

```
typedef struct _mutex {
    int allocated;
    int no_task;
    int locked;
    Queue waiting_queue;
} Mutex;
```

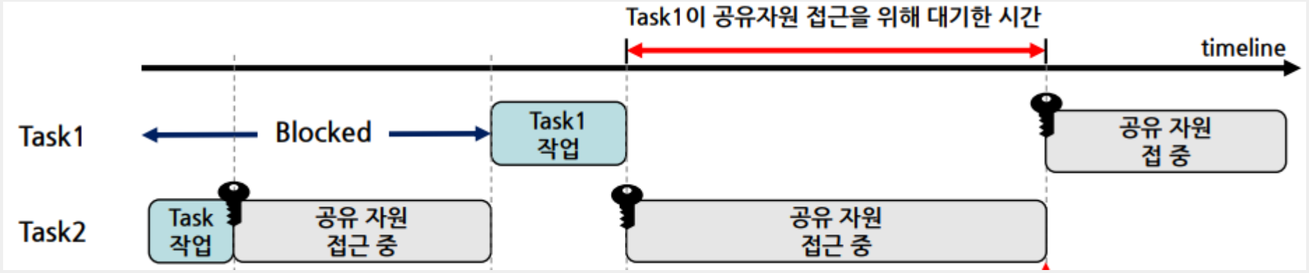
    - ✓ 공유 자원 접근을 완료하면 waiting queue에서 dequeue 진행하고 해당 mutex를 unlock

```
void Give_Mutex(int mutex_id, int task_related) {
        dequeue(&mutex_list[mutex_id]->waiting_queue, &next_tcb_no, NO_ALLOCATED_TASK);
        mutex_list[mutex_id]->locked = 1;
```

    - ✓ 공유 자원을 사용하는 곳의 앞, 뒤에 Mutex를 적용하고, Uart printf에 적용했을 때 정상 동작함을 확인

```
void Uart1_Printf_From_Task(char *fmt,...)
{
    va_list ap;
    char string[128];

    Take_Mutex(uart_mutex_id, TASK_RELATED);
    va_start(ap, fmt);
    vsprintf(string, fmt, ap);
    Uart1_Send_String(string);
    va_end(ap);
    Give_Mutex(uart_mutex_id, TASK_RELATED);
```



Mutex concept

4. 결과 분석 및 기대 효과

- **결과 분석 - Test case 기반 구현 내용의 robustness 확인**
  - Task & Scheduler 및 Blocked based delay
    - ✓ 다양한 우선순위의 task를 만들었을 때 우선순위에 맞추어 정상 동작 확인
    - ✓ 같은 우선순위의 task가 있는 경우 번갈아 가면서 동작함을 확인
    - ✓ Task를 spec 기준 최대한 많이 만들었을 때 정상 동작 확인
    - ✓ Context switching 시 critical section에서 강제로 interrupt 발생 시 critical section 수행 후 interrupt 발생 확인
  - Signaling & Queue
    - ✓ 예외처리가 필요한 상황을 TC로 구현하여 해당 상황과 일치하는 경고문 발생 확인
    - ✓ Hardware interrupt를 발생시켜 Queue에 data를 입력하고 정상적으로 출력함을 확인
  - Mutex
    - ✓ 2개의 task가 공유자원을 경쟁하여 사용하는 경우 mutex를 이용하여 정상 동작 확인
    - ✓ Priority inversion 상황을 만들어 priority inheritance 기능의 정상 동작 확인

- **기대 효과**
  - CPU의 효율적인 사용을 통해 Snake 게임과 같은 여러가지 task를 효율적으로 구동하는 Snake 게임 구현
    - ✓ 실행 시간 보장을 통한 실시간 응답성 향상
    - ✓ CPU 및 메모리 자원 관리 최적화
    - ✓ Task scheduling을 통한 복잡도 감소 및 동시 실행

5. 향후 연구 과제

- **추후 개선점**
  - Task state 세분화를 통한 구체적인 task 상태를 표현
    - ✓ Task suspended 구현(Suspended Ready, Blocked)
    - ✓ Blocked state 세분화(Blocked I/O, Blocked resource)

  - Application에 따라 같은 우선순위를 갖는 task에 대한 scheduling 방법 선택
    - ✓ 현재 구현 방법은 모든 준비 상태의 task에게 시간을 균등하게 할당하는 Round robin scheduling 형태임
    - ✓ First-come, first-served 방법
    - ✓ 실행 시간이 가장 짧은 task를 가장 먼저 실행

  - Signaling & Queue
    - ✓ Signal/Queue 간 우선순위 구현

  - Mutex
    - ✓ Multi-threading을 사용하는 상황을 위한 recursive mutex를 구현

6. 프로젝트 수행 후기

- **김정진**
  - 느낀점: RTOS의 구조와 동작 원리를 실습을 통해 깊이 이해할 수 있었습니다. 특히, 우선순위 상속과 컨텍스트 스위칭 등의 핵심 기능을 구현하며 실무 지식을 쌓을 수 있었습니다.
  - 어려웠던 점: 동기화와 컨텍스트 스위칭 구현에서 많은 어려움이 있었습니다. 특히, Mutex를 통한 임계 구역 보호와 우선순위 상속 구현 과정에서 디버깅 및 테스트가 매우 복잡했습니다.
  - 교육 종료 소감: RTOS 프로젝트를 수행하며 실무에서의 응용 가능성을 체험하는 매우 유익한 시간이었습니다.

- **심훈**
  - 느낀점: 현업에서 가전 OS 개발을 하면서 Free RTOS 기반으로 하는 일을 했는데 직접 만들어보니 OS 가 대단한 일을 하고 있음을 느낄 수 있었습니다.
  - 어려웠던 점: Task 간의 Signal을 주고 Wait 하면서 기다리고 queue등에 대한 개념과 구현 모두 어려웠습니다.
  - 교육 종료 소감: 동료들과 협업해서 하는 프로젝트는 처음이었는데 모듈단위로 나누어서 일을 해야함을 배울 수 있었습니다. 훌륭한 동료들과 같이 프로젝트를 할 수 있어서 감사했습니다.

- **김소연**
  - 느낀점: RTOS를 사용하는 환경은 주로 메모리가 제한적이므로 해당 부분에 대해서 주의를 기울여 설계해야 한다는 것을 느꼈습니다.
  - 어려웠던 점: 프로젝트에 사용한 보드의 메모리가 작아서 제한적인 메모리를 어떻게 사용할 것인지 설계하는 부분이 어려웠습니다.
  - 교육 종료 소감: 프로젝트 일정이 빠듯했는데 무사히 마무리하게 되어서 뿌듯합니다. 팀원분들과 협업하며 많이 배울 수 있는 유익한 시간이었습니다.

- **이성훈**
  - 느낀점: 모든 SW 개발자들에게 경의를 표합니다.
  - 어려웠던 점: 과거 학부에서 RTOS를 배웠으나 직접 구현한 적은 없어 이를 low level로 구현하는 데에 어려웠습니다.
  - 교육 종료 소감: 이 교육을 계기로 좀 더 발전할 수 있었습니다. 마지막 신입사원 교육이라 마음이 아프네요.