

PBL 팀 프로젝트 보고서 (6팀)

프로젝트 명	다중 쓰레드 컨텍스트 스위칭 프로그래밍 (Multi-thread context switching programming)			
활동기간	2020년 11월 26일 ~ 2020년 12월 18일			
작성일	2020년 12월 18일			
과목정보	단과대학명	융합공과대학	학부(과)명	융합전자공학과
	교과목명	마이크로프로세서	담당교수명	신동하
팀원	성명 (팀장표시)	학과	학번	담당업무
	이학민(팀장)	융합전자공학과	201910906	프로젝트 진행 및 코드 작성
	강수연	융합전자공학과	201910882	프로젝트 분석 및 코드 작성
	박예슬	융합전자공학과	201910895	프로젝트 분석 및 코드 작성
	안무진	융합전자공학과	201710869	문제 기술 및 회의록 작성
	인정환	융합전자공학과	201710879	참고 문헌 및 개발 과정 정리
	김수혁	융합전자공학과	201710856	문제 풀이 보고서 작성

목차

1. 문제 기술

1.1 문제의 기술

1.2 문제의 중요성

1.3 문제의 활용성

2. 문제 풀이

2.1 문제의 풀이 방법

2.2 풀이에 사용된 마이크로프로세서 기술 설명

2.3 작성한 프로그램 쓰기 및 설명

2.4 수행 시연

3. 문제 풀이(sleep_thread 확장 기능 구현한 경우 작성함)

3.1 문제의 풀이 방법

3.2 작성한 프로그램 쓰기 및 설명

3.3 수행 시연

4. 개발 과정

4.1 개발 일정

4.2 업무 분담

4.3 개발 시 발생한 문제 및 해결한 방법

4.4 회의록 첨부

5. 참고 문헌

5.1 참고 문헌

1. 문제 기술

1.1 문제의 기술

본 문제는 Multi-thread Context Switching을 구현하는 것으로, 하나의 processor에서 다수의 프로그램이 구동될 수 있도록 memory와 register를 사용하여 정보를 기억하고 다시 백업 할 수 있게 적절한 조치를 취한 다음 각각의 thread를 동시에 작동하게 하는 코드를 작성하는 것이다. 한 개의 processor를 사용하여 여러 개의 task를 동시에 수행하는 것처럼 보이기 위해 각 thread의 instruction을 빠르게 순번이 돌아가며 조금씩 수행한다. 따라서 한 개의 thread가 종료되고 다른 thread가 수행되기 위해 현재 수행 중인 thread의 정보를 저장하고 다음으로 수행해야 할 thread의 정보를 불러오는 과정을 거친다. 이와 같은 방식으로 thread가 바뀌며 수행되다가 마지막 순번인 thread(N)의 수행이 종료되면 첫 번째 thread로 돌아와 앞서 설명한 과정을 반복한다. 이러한 과정을 진행하기 위해 thread의 정보를 저장, 교환해주는 작업을 Context Switching이라고 부르고 Context는 thread의 정보인 r0~r12까지, r14(=lr), r15, xpsr을 포함한다.

1.2 문제의 중요성

본 multi-thread 기술은 하나의 core안에 다수의 병렬 작업을 통해 task를 수행한다. CPU의 개수를 늘리지 않고 thread의 작업을 늘리기 때문에 값이 비싼 CPU의 특성을 고려하면 상당한 비용 절감 효과를 누릴 수 있다. 또한 병렬 작업을 수행하여 한 작업량을 분산하여 수행함으로써 시간을 단축하는 효과를 얻는다. 마지막으로 multi-thread 수행 시 global-variable을 공유하여 한 프로그램이 같은 주소 공간 내에 여러 개의 역할이 다른 thread를 가지고 있어 변동성 대처에 우수하다. 이러한 이점들을 통해 multi-thread 기술은 소비자 상품화에 큰 역할을 수행하고 회사 측에서 기간 단축 및 비용 절감으로 이윤창출에 기여할 수 있다.

1.3 문제의 활용성

본 문제의 활용성을 이야기하기 앞서 CPU와 GPU에 대해 간단히 설명하겠다. 우선 CPU는 한 코어에 대량의 multiple 작업을 수행하는 것이고, GPU는 다수의 작은 코어에 대량의 multiple 작업을 수행하는 것이다. 본 문제의 기술(multi-thread)적 부분을 활용하여 극대화한 제품을 GPU라고 이야기할 수 있다. 기존까지의 컴퓨터 시장은 CPU가 지배를 하였지만 AI 발달 및 Graphic 처리 수행의 필요성 등의 시장 수요에 CPU보다 GPU의 multiple 작업 수행이 시간 단축에서 우수하고 가격이 싼 GPU가 더 적절하였고 현재 GPU의 시장규모는 CPU를 압도적으로 추월하였다. 따라서 우리는 현재 multiple 작업 수행의 기초적인 부분을 잘 이해하고 GPU의 등장과 같은 사례처럼 본 기술을 잘 활용할 필요가 있다.

2. 문제 풀이

2.1 문제의 풀이 방법

다수의 thread를 수행시키기 위해서는 다수의 TCB를 관리해야 한다. 이때 TCB(Thread control block)은 하나의 thread에 대한 정보를 저장하고 있는 자료구조로 sp(thread가 사용하는 stack의 top주소), function(thread의 수행 시작 주소), state(thread의 상태로 RUN, READY, WAIT 등) 3가지 요소를 포함한다.

본 문제에서는 6개의 thread(thread0~thread5)를 사용하므로 create_thread함수로 6개의 thread를 만들고, thread0을 시작으로 context-switching을 하도록 프로그래밍 되었다.

current thread를 사용하기 위해서는 context를 stack에 push하고 변화된 sp를 TCB의 sp에 저장해야 한다. 위 과정은 create_thread에서 각 thread의 stack주소를 인수로 받아 수행하게 한다.

Next thread를 사용하기 위해서는 TCB에 저장된 sp를 sp로 복구하고 stack에 저장된 context를 register로 pop해야 한다. 위 과정은 PendSV handler에서 수행하게 한다.

이에 따라 작성한 함수와 변수의 상세한 설명은 2.3에서 하도록 한다.

2.2 폴이에 사용된 마이크로프로세서 기술 설명

1) Multi-threading

Multi-threading이란 하나의 processor 상에서 다수의 프로그램을 동시에 수행시키는 일이다. 다수의 thread가 돌아가면서 processor를 사용하여 자신의 instruction을 조금씩 수행해 나간다. 하나의 thread는 수행되기 위하여 function(수행할 instruction들)과 stack(local variable 등을 저장)이 필요하다. 다수의 thread들은 수행하면서 global variable들은 공유한다. Multi-threading은 thread의 수준과 명령어 수준의 병렬 처리 모두에 신경을 쓰면서 하나의 코어에 대한 이용성을 증가하는 것에 초점을 두고 있다.

1-1) Multi-threading의 장점

① 응답성

대화형 프로그램을 multi-thread화 하면, 프로그램의 일부분이 중단되거나 긴 작업을 수행하더라도 프로그램의 수행이 계속되어 사용자에게 대한 응답성이 증가된다.

② 자원공유

thread는 자동적으로 그들이 속한 프로세스의 자원들과 메모리를 공유한다. 코드 공유의 이점은 한 응용프로그램이 같은 주소 공간 내에 여러 개의 다른 활동성 thread를 가질 수 있다는 점이다.

③ 경제성

프로세스 생성에 메모리와 자원을 할당하는 것은 비용이 많이 든다. thread는 자신이 속한 프로세스의 자원들을 공유하기 때문에, thread를 생성하고 context-switching을 하는 것이 더 경제적이므로 경제성이 좋다.

1-2) Multi-threading의 단점

① 캐시나 TBL와 같은 하드웨어 리소스를 공유할 때 서로를 간섭할 수 있다.

② 하나의 thread만 실행 중인 경우 single thread의 실행시간이 지연될 수 있다.

③ Multi-threading의 하드웨어 지원을 위해 응용프로그램과 운영체제 모두 변화가 필요하다.

Multi-threading의 장단점에 대해 알아보았다. multi-thread의 모델에는 세 가지가 있는데, 그 중 첫 번째는 다대일(Many-to-one)모델이다. 여러 개의 사용자 수준 thread들이 하나의 커널 스레드로 mapping되는 방식으로, 사용자 수준에서 thread관리가 이루어진다. 주로 커널 thread를 사용하지 않는 시스템에서 사용하며, 한 번에 하나의 thread만이 커널에 접근할 수 있다는 단점이 있고, 동시성을 지원하지 못한다. 두 번째는 일대일(One-to-one)모델이다. 사용자 thread들을 각각 하나의 kernel thread로 mapping시키는 방식이다. 사용자 thread가 생성이 되면 그에 따른 kernel thread가 생성되는 것이다. 이렇게 하면 다대일 방식에서 시스템 호출 시 다른 thread들이 중단되는 문제를 해결할 수 있으며 여러 개의 thread를 다중 처리기에 분산하여 동시에 수행할 수 있는 장점이 있다. 마지막으로 다대다(Many-to-many)모델이다. 여러 개의 사용자 thread를 여러 개의 커널 thread로 mapping시키는 모델이다. 위의 두 가지 모델의 문제점을 해결하기 위해 고안되었다.

2) Thread Control Block

Thread Control Block(TCB)은 하나의 thread에 대한 정보를 저장하는 자료구조(structure)이다. 이 TCB가 저장하고 있는 정보들은 다음과 같다. TCB는 아래와 같은 정보들을 포함하고 있으며, 다수의 thread가 수행되는 경우 다수의 TCB를 관리하여야 한다.

- ① 스레드 식별자(thread ID)
- ② 스레드 상태(thread state)
생성(create), 준비(ready), 실행(running), 대기(waiting), 완료(terminated) 상태가 있다.
- ③ 프로그램 계수기(program counter)
program counter는 이 프로세스가 다음에 실행할 명령어의 주소를 가리킨다.
- ④ CPU 레지스터 및 일반 레지스터
- ⑤ CPU 스케줄링 정보
우선 순위, 최종 실행시각, CPU 점유시간을 말한다.
- ⑥ 메모리 관리 정보
해당 thread의 주소공간 등을 말한다.
- ⑦ Thread 계정 정보
페이지 테이블, 스케줄링 큐 포인터, 소유자 등을 말한다.
- ⑧ 입출력 상태 정보 (I/O state)
thread에 할당된 입출력장치 목록, 열린 파일 목록 등을 말한다.

3) Context-Switching

Context란 어떤 thread가 수행되는 도중 계산의 중간 상태를 저장하고 있는 register들을 말한다. ARM Cortex-M4에서 어떤 thread의 context는 r0~r12, r13(=sp), r14(=lr), r15(=sp) 및 xpsr로 구성된다. Context Switching은 프로세스 실행 중 다른 프로세스의 CPU사용을 위해 작업 상태를 보관하고 새 프로세스에 상태를 적재하는 작업이다. Context Switching은 다음과 같은 절차로 인하여 발생한다.

- ① 각 TCB는 thread의 우선 순위와 stack pointer를 보유한다.
- ② 준비상태 thread의 TCB 스택은 실행했던 context를 마지막 stack에 넣는다.
- ③ Thread2가 준비되어 실행 순서에 SP가 가리키던 context를 CPU 레지스터에 복사한다.
- ④ SP는 context를 복구하며 삭제한 만큼 변화되어 다시 해당 thread 스택으로 사용한다.
- ⑤ 실행되던 thread가 멈추고 다른 thread로 바뀔 때 반대로 역방향의 thread 스택으로 저장된다.
- ⑥ SP를 push한 만큼 업데이트 한 후 CPU의 사용권에서 벗어난다.

4) SysTick and PendSV handlers

Systick은 kernel에서 Context Switching을 구현할 때 사용된다. ARMv7-M은 24-bit clear-on-write, decrementing, wrap-on-zero counter 기능을 가지는 system timer인 SysTick을 포함하고 있다. 이 SysTick는 kernel에서 time, multi-threading, alarm timer를 구현하는 용도로 사용된다.

PendSV 또한 커널에서 Context-Switching을 구현할 때 사용된다. 일반적으로 가장 낮은 우선순위로 설정되고 SCS에 있는 ICSR(Interrupt Control and State Register)설정을 하면 나중에 asynchronous하게 발생한다. 현재 수행 중인 exception이 있으면 이 exception의 handler가 return된 후 자동으로 발생된다. 이처럼 위 기능들이 Context Switching을 수행하기 위한 작업을 살펴보면 다음과 같다.

4-1) SysTick handler

- ① 변수 tick을 1 증가시킨다. (1 tick = 1/100 sec)
- ② current thread의 상태를 수정한다. (RUN -> READY)
- ③ next thread를 선정한다. (thread 나열 순서에서 다음 thread)
- ④ next thread의 상태 수정한다. (READY -> RUN)
- ⑤ PendSV exception을 pending시킨다. (SCB의 ICSR의 PENDSVSET을 1로 set한다.)

4-2) PendSV handler

- ① current thread의 context(r4~r11 부분)를 stack에 push하고 수정된 SP를 current thread의 TCB에 저장한다.
- ② current thread를 next thread로 변경한다.
- ③ next thread의 TCB의 SP에서 SP를 복구하고 stack에서 next thread의 context(r4~r11 부분)를 pop한다.

2.3 작성한 프로그램 쓰기 및 설명

```
void SysTick_Handler(void)
{
    // Increment tick.
    tick = tick + 1;

    // Update tid_current and thread state.
    tcb_array[tid_current].state = STATE_READY;
    tid_current = (tid_current+1) % NO_OF_THREADS;
    tcb_array[tid_current].state = STATE_RUN;

    // Update tcb_next for PendSV handler.
    tcb_next = &tcb_array[tid_current];

    // Make PendSV exception pending.
    *(volatile unsigned int *) SCB_ICSR = 0x10000000;
}
```

<프로그램 설명>

Systick_Handler 는 Systick Interrupt 를 발생시킨다. Systick Interrupt 이 발생할 때 프로그램에서 어떠한 변화가 나타나야 하는지 담고 있는 함수이다. Systick Interrupt 는 주어진 문제에 따르면 10ms 마다 발생한다. Systick Interrupt 가 발생할 때 tick 의 값을 1 증가시킨다. 현재 수행중인 thread 의 TCB 는 tcb_array[tid_current]로 나타낸다. 다음에 수행할 thread 를 위해 현재 수행 중인 thread 를 잠시 중단시키고 준비 상태로 만들어 주어야 하므로 현재 TCB 의 상태를 STATE_READY 로 바꿔준다. 다음 thread 의 id 로 넘어가기 위해 tid_current 에 1 을 더해준다. 이때 % 연산자는 tid_current 값이 NO_OF_THREADS 의 값인 6 에 도달하지 않게 하기 위하여 붙여주었다. 다시 말하면 thread0 에서 thread5 사이에서만 값이 변화될 수 있도록 범위를 제한한 것이다. 지금까지의 과정을 통해 tid_current 의 값이 변하였고 tcb_array[tid_current]는 다음에 수행될 TCB 의 정보를 가리킨다. 이 TCB 의 상태를 STATE_RUN 으로 바꿔주면 다음 thread 를 수행시킬 수 있다. PendSV handler 의 다음 TCB 즉, tcb_next 를 새로고침 해주기 위해 tcb_next 에 다음에 수행할 thread 상태를 담고 있는 tcb_array[tid_current]의 주소를 대입해준다. PendSV exception pending 을 만들기 위하여 SCB_ICSR 레지스터의 28 번째 비트에 1 을 대입해 주었다.


```
// Create all threads.  
create_thread(&tcb_array[0], thread0_function,  
              &thread0_stack[SIZE_OF_STACK - 1]);  
create_thread(&tcb_array[1], thread1_function,  
              &thread1_stack[SIZE_OF_STACK - 1]);  
create_thread(&tcb_array[2], thread2_function,  
              &thread2_stack[SIZE_OF_STACK - 1]);  
create_thread(&tcb_array[3], thread3_function,  
              &thread3_stack[SIZE_OF_STACK - 1]);  
create_thread(&tcb_array[4], thread4_function,  
              &thread4_stack[SIZE_OF_STACK - 1]);  
create_thread(&tcb_array[5], thread5_function,  
              &thread5_stack[SIZE_OF_STACK - 1]);
```

<프로그램 설명>

thread0부터 thread5까지 모든 thread를 생성하는 과정이다. create_threadX는 parameter를 총 3개 갖는 함수로 첫 번째 parameter는 thread의 정보를 담고 있는 TCB의 주소, 두 번째 parameter는 보드에서 파일 수행 시 화면 출력 역할을 수행하는 threadX_function, 세 번째 parameter는 thread 생성 시 context의 정보를 담을 수 있는 변수의 주소값이다. thread0부터 thread5까지 총 6개의 thread가 있으므로 각 thread에 맞게 6번 선언하였다.

```

void create_thread(TCB * tcb, void (*function) (void), unsigned int *sp)
{
    *--sp = 0x01000000;        // xpsr (Thumb=1)
    *--sp = (unsigned int) function;    // pc(=r15)
    *--sp = 0x00000000;        // lr(=r14)
    *--sp = 0x00000000;        // r12
    *--sp = 0x00000000;        // r11
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;

    tcb->sp = (unsigned int) sp;
    tcb->function = (unsigned int) function;
    tcb->state = STATE_READY;
}

```

<프로그램 설명>

create_thread 는 thread 를 수행하기 전 생성하는 함수이다. 각 thread 의 초기 정보를 설정해야하는데, sp 를 주소를 감소해가며 *연산자를 통해 주소에 해당하는 값에 접근하여 각 레지스터의 값을 알맞게 초기화하였다. xpsr 에 해당하는 부분에는 0x01000000, r15 에 해당하는 부분에는 unsigned int 형의 function 을 대입한다. 나머지 레지스터인 r0~r12 는 모두 기본 세팅인 0 으로 초기화한다. sp 에 context 의 정보들을 저장하고 나면 현재 thread 의 TCB stack pointer 에 접근하여 초기화한 sp 의 값을 넣어준다. 현재 TCB 의 function 에는 unsigned int 형의 현재 function 을 대입한다. 위 함수는 thread 를 생성하는 함수로 thread 가 수행되는 것은 아니기 때문에 상태는 READY 로 설정한다.

```
// Thread stacks and functions.
```

```
extern unsigned int thread0_stack[];  
extern unsigned int thread1_stack[];  
extern unsigned int thread2_stack[];  
extern unsigned int thread3_stack[];  
extern unsigned int thread4_stack[];  
extern unsigned int thread5_stack[];
```

```
extern void thread0_function(void);  
extern void thread1_function(void);  
extern void thread2_function(void);  
extern void thread3_function(void);  
extern void thread4_function(void);  
extern void thread5_function(void);
```

<프로그램 설명>

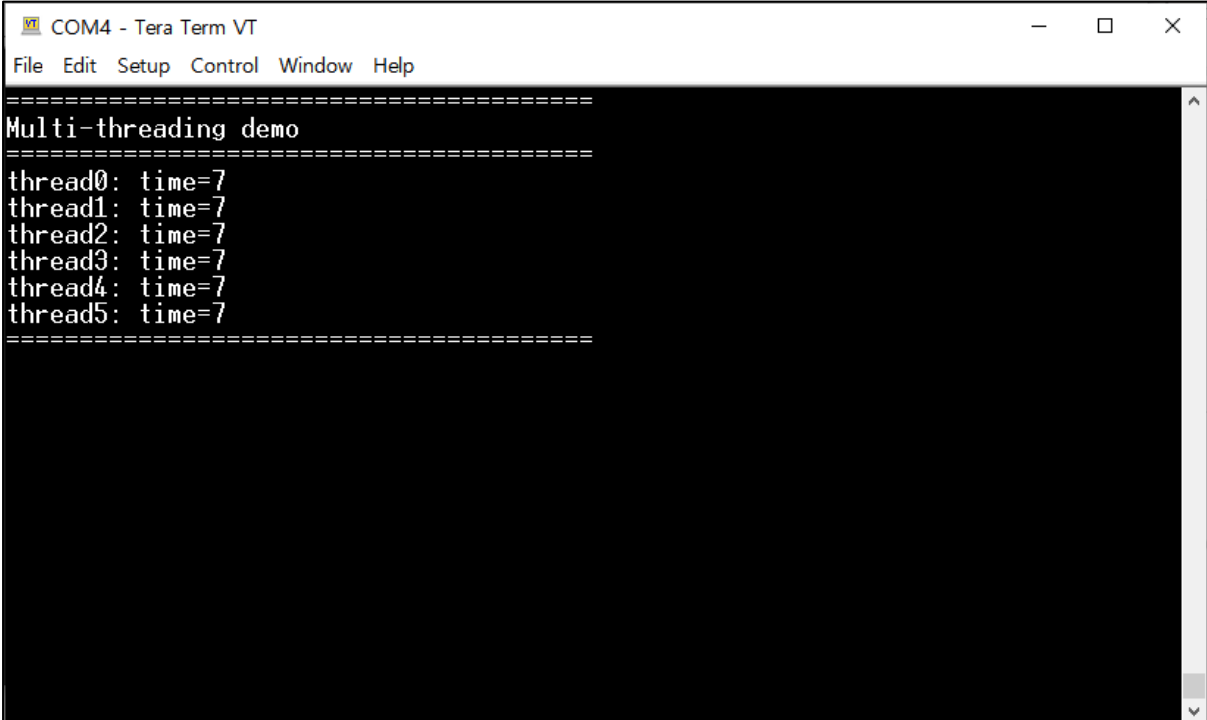
thread 를 생성(create)할 때 thread0 부터 thread5 까지의 thread 의 context 정보를 임시로 저장할 수 있는 공간인 threadX_stack[]을 사용하기 위해 unsigned int 형의 threadX_stack[]을 선언하였다. 이때 threadX_stack[] 에서의 X 는 0 부터 5 까지의 정수이다.

thread0 부터 thread5 까지의 threadX_function()함수를 사용할 수 있도록 threadX_function()함수의 prototype 을 선언하였다. 이때 threadX_function()에서의 X 는 0 부터 5 까지의 정수이다.

여기서 모든 변수와 함수는 다른 소스파일에서도 접근할 수 있도록 extern 을 사용하였다.

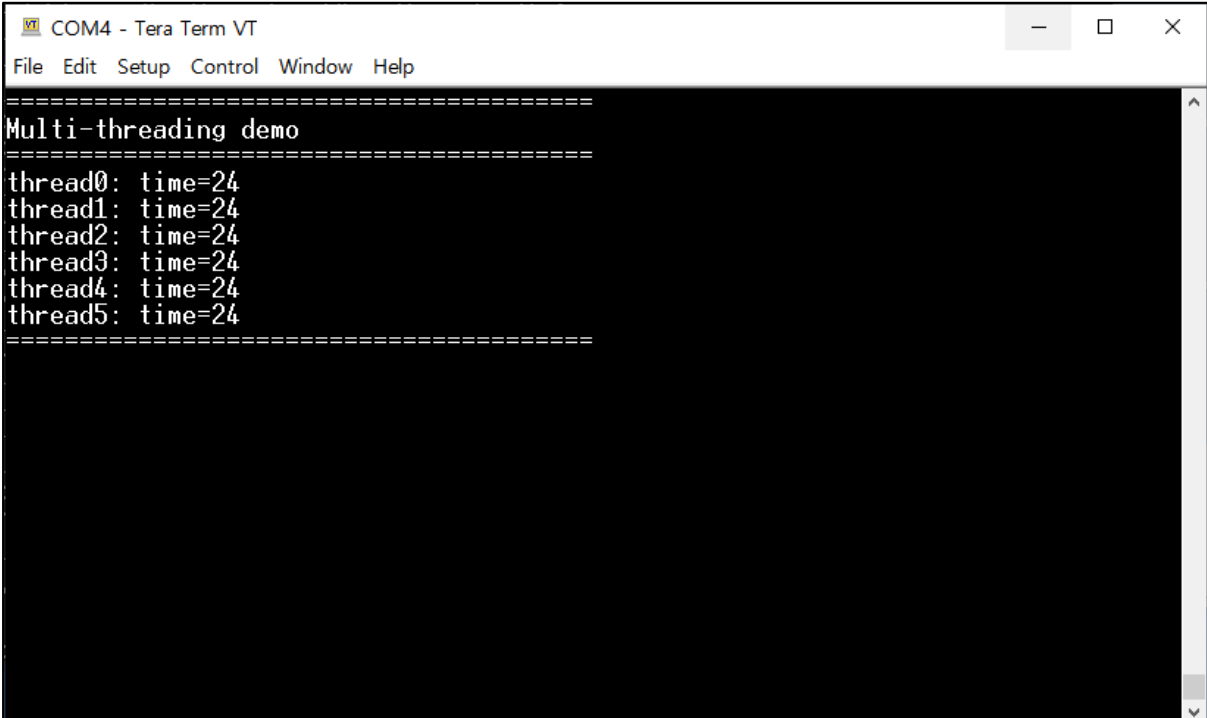
2.4 수행 시연

다음 첨부한 사진은 threads.bin 파일을 보드에서 수행하였을 때 보여지는 화면이다.



```
COM4 - Tera Term VT
File Edit Setup Control Window Help
=====
Multi-threading demo
=====
thread0: time=7
thread1: time=7
thread2: time=7
thread3: time=7
thread4: time=7
thread5: time=7
=====
```

수행 후 7초 경과한 화면.



```
COM4 - Tera Term VT
File Edit Setup Control Window Help
=====
Multi-threading demo
=====
thread0: time=24
thread1: time=24
thread2: time=24
thread3: time=24
thread4: time=24
thread5: time=24
=====
```

수행 후 24 초 경과한 화면.

3. 문제 풀이

3.1 문제의 풀이 방법

본 문제는 2.1 에 기술한대로 첫 번째 문제와 기본적인 방식은 동일하나, sleep_thread 라는 함수를 구현하고 Thread control block 에 context switching 을 기다리는 상태인 WAIT 와 thread 의 sleep_tick 을 나타내는 변수를 추가하게 된다. Thread(1~5) 함수 내에서 sleep_thread 함수가 call 될 때, 각 다른 ticks 시간이 인수로 주어지며 sleep_thread 함수는 그 시간동안 thread 가 수행되지 않도록 기다리게 한다. 매초 마다 모든 thread 가 동시에 수행되며 그 결과가 출력되었던 기본 문제와는 다르게 본 문제에서는 ticks 시간 동안은 processor 를 수행하지 않고 WAIT 하게 함으로써 각 시간마다 수행되는 thread 가 다르며 매초 마다 수행된 thread 를 출력하게 된다. 이 문제를 풀이하기 위해서는 SysTick_Handler 에서 sleep_tick 이 0 이 아닌 thread 를 탐색하여 수행되지 않도록 상태를 WAIT 으로 변경해주어야 하고 시간이 흐름에 따라 sleep_tick 이 감소되도록 수정해야한다. 또한, next thread 는 sleep 중이 아닌 thread 로 선정해야 한다. 따라서 바로 다음 thread 를 next thread 로 선정했던 기본 문제와는 다르게 각 thread 의 sleep_tick 를 차례로 탐색하여 0 이 아닌 thread 를 next thread 로 선정하는 방법을 사용해야한다. 이러한 방식으로 각 thread 는 가지게 된 sleep_tick 시간동안 수행되지 않고 대기하게 되며 시간이 흘러 sleep_tick 이 0 이 되었을 때 수행된다. 따라서 기본 문제와는 다르게 Time 별로 수행되는 thread 가 무엇인지 출력되도록 하였다.

3.2 작성한 프로그램 쓰기 및 설명

```
void SysTick_Handler(void)
{
    // Increment tick.
    tick = tick + 1;

    tcb_array[tid_current].state = STATE_READY;

    for(int i=0; i< NO_OF_THREADS; i++)
    {
        if(tcb_array[i].sleep_tick != 0)
        {
            tcb_array[i].state = STATE_WAIT;
            tcb_array[i].sleep_tick--;

            if(tcb_array[i].sleep_tick == 0)
                tcb_array[i].state = STATE_READY;
        }
    }

    tid_current = 0;

    do
    {
        tid_current = (tid_current+1) % NO_OF_THREADS;
    }while(tcb_array[tid_current].sleep_tick!=0);

    tcb_array[tid_current].state = STATE_RUN;

    tcb_next = &tcb_array[tid_current];

    // Make PendSV exception pending.
    *(volatile unsigned int *) SCB_ICSR = 0x10000000;
}
```

<프로그램 설명>

Systick_Handler 는 Systick Interrupt 를 발생시킨다. Systick Interrupt 이 발생할 때 프로그램에서 어떠한 변화가 나타나야 하는지 담고 있는 함수이다. Systick Interrupt 는 주어진 문제에 따르면 10ms 마다 발생한다. Systick Interrupt 가 발생할 때 tick 의 값을 1 증가시킨다. 현재 수행중인 thread 의 TCB 는 tcb_array[tid_current]로 나타낸다. 다음에 수행할 thread 를 위해 현재 수행 중인 thread 를 잠시 중단시키고 준비 상태로 만들어 주어야 하므로 현재 TCB 의 상태를 STATE_READY 로 바꿔준다.

thread 의 sleep_tick 이 0 이 아닐 경우 thread 를 수행시키면 안되므로 sleep 모드인 STATE_WAIT 상태로 바꿔준다. 또한 모든 thread 의 sleep_tick 은 sleep_tick 이 0 이 아닐 경우 10ms 마다 Systick Interrupt 가 발생할 때 1 씩 감소해야 한다. sleep_tick 을 1 감소하고 만약 sleep_tick 이 0 이 아닌 thread 를 찾으면 그 thread 의 상태를 READY 로 바꿔준다.

보드에서 수행 시 화면 출력에서 thread 가 작은 수부터 순서대로 나올 수 있도록 tid_current 를 0 으로 바꾸어 주었다. 다음으로 수행할 thread 의 id 로 넘어가기 위해 sleep 모드가 아닌 thread 가 나올 때 까지 tid_current 에 1 을 더해준다. 이때 % 연산자는 tid_current 값이 NO_OF_THREADS 의 값인 7 에 도달하지 않게 하기 위하여 붙여주었다. 다시 말하면 thread0 에서 thread6 사이에서만 값이 변화될 수 있도록 범위를 제한한 것이다.

지금까지의 과정을 통해 tid_current 의 값이 변하였고 tcb_array[tid_current]는 다음에 수행될 TCB 의 정보를 가리킨다. 이 TCB 의 상태를 STATE_RUN 으로 바꿔주면 다음 thread 를 수행시킬 수 있다. PendSV handler 의 다음 TCB 즉, tcb_next 를 새로고침 해주기 위해 tcb_next 에 다음에 수행할 thread 상태를 담고 있는 tcb_array[tid_current]의 주소를 대입해준다. PendSV exception pending 을 만들기 위하여 SCB_ICSR 레지스터의 28 번째 비트에 1 을 대입해 주었다.

```
// Create all threads.
create_thread(&tcb_array[0], thread0_function,
              &thread0_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[1], thread1_function,
              &thread1_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[2], thread2_function,
              &thread2_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[3], thread3_function,
              &thread3_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[4], thread4_function,
              &thread4_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[5], thread5_function,
              &thread5_stack[SIZE_OF_STACK - 1]);
create_thread(&tcb_array[6], thread6_function,
              &thread6_stack[SIZE_OF_STACK - 1]);
```

<프로그램 설명>

thread0부터 thread6까지 모든 thread를 생성하는 과정이다. create_threadX는 parameter를 총 3개 갖는 함수로 첫 번째 parameter는 thread의 정보를 담고 있는 TCB의 주소, 두 번째 parameter는 보드에서 파일 수행 시 화면 출력 역할을 수행하는 threadX_function, 세 번째 parameter는 thread 생성 시 context의 정보를 담을 수 있는 변수의 주소값이다. thread0부터 thread6까지 총 7개의 thread가 있으므로 각 thread에 맞게 7번 선언하였다.

```
// Pretend the first thread is running.
tid_current = 0;
tcb_current = &tcb_array[0];
tcb_current->state = STATE_RUN;
tcb_current->sleep_tick = 0;
```

<프로그램 설명>

빨간색 글씨만 추가로 작성한 코드이다.

thread0 이 RUN 하는 상태로 만들 때, 현재 TCB 의 sleep_tick 값은 0 이 되어야 하므로 tcb_current 의 sleep_tick 의 값을 0 으로 초기화 해주었다.


```

void create_thread(TCB * tcb, void (*function) (void), unsigned int *sp)
{
    *--sp = 0x01000000;          // xpsr (Thumb=1)
    *--sp = (unsigned int) function; // pc(=r15)
    *--sp = 0x00000000;          // lr(=r14)
    *--sp = 0x00000000;          // r12
    *--sp = 0x00000000;          // r11
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;
    *--sp = 0x00000000;

    tcb->sp = (unsigned int) sp;
    tcb->function = (unsigned int) function;
    tcb->state = STATE_READY;
    tcb->sleep_tick = 0;
}

```

<프로그램 설명>

create_thread 는 thread 를 수행하기 전 생성하는 함수이다. 각 thread 의 초기 정보를 설정해야하는데, sp 를 주소를 감소해가며 *연산자를 통해 주소에 해당하는 값에 접근하여 각 레지스터의 값을 알맞게 초기화하였다. xpsr 에 해당하는 부분에는 0x01000000, r15 에 해당하는 부분에는 unsigned int 형의 function 을 대입한다. 나머지 레지스터인 r0~r12 는 모두 기본 세팅인 0 으로 초기화한다. sp 에 context 의 정보들을 저장하고 나면 현재 thread 의 TCB stack pointer 에 접근하여 초기화한 sp 의 값을 넣어준다. 현재 TCB 의 function 에는 unsigned int 형의 현재 function 을 대입한다. 위 함수는 thread 를 생성하는 함수로 thread 가 수행되는 것은 아니기 때문에 상태는 READY 로 설정한다. 또한 thread 를 생성했을 당시, STATE_READY 상태이므로 WAIT 이 아니기 때문에 sleep_tick=0 으로 설정하였다.

```

void sleep_thread(unsigned int ticks)
{
    tcb_current->sleep_tick = ticks;
    tcb_current->state = STATE_WAIT;
    while(tcb_current->state == STATE_WAIT) printf("");
}

```

<프로그램 설명>

sleep_thread 는 ticks 를 받아 현재 thread TCB 의 sleep_tick 에 저장하고 현재 thread 의 상태를 WAIT 으로 바꿔주는 함수이다. 이때 WAIT 상태일 때는 sleep_thread 를 벗어나면 안되기 때문에 현재 thread 의 상태가 STATE_WAIT 이면 아무 것도 출력하지 않는 printf 구문을 추가해 아무 일도 일어나지 않은 채로 sleep_thread 함수를 종료하지 못하도록 만들었다.

만약 while(tcb_current->state == STATE_WAIT) printf(""); 가 없다면, 다시 말해 sleep_thread 를 빠져나가지 못하게 하는 코드를 넣지 않는다면 threadX_function 의 다음 부분으로 넘어가기 때문에 본 보고서 4.3 의 2)와 같은 오류가 발생한다.

```

#define NO_OF_THREADS      7
#define SIZE_OF_STACK      128

#define STATE_RUN          0
#define STATE_READY        1
#define STATE_WAIT         2

```

<프로그램 설명>

빨간색 글씨만 추가로 작성한 코드이다.

sleep_tick 이 0 이 아닐 경우 Context Switching 을 대기하는 상태로 만들어주어야 하기 때문에 WAIT 하는 상태를 STATE_WAIT 으로 정의해주었고 값은 2 로 설정하였다.

```
// The thread control block.

typedef struct tcb_struct {
    unsigned int sp;           // thread stack pointer
    unsigned int function;     // thread function
    unsigned int state;        // thread state (RUN, READY, ...)
    unsigned int sleep_tick;   // sleep tick
} TCB;
```

<프로그램 설명>

빨간색 글씨만 추가로 작성한 코드이다.

thread 의 정보를 저장하는 TCB(Thread Control Block)에 sleep_tick 을 저장할 수 있도록 sleep_tick 변수를 선언하였다.

```
extern void create_thead(TCB * tcb, void (*function) (void), unsigned int *sp);
extern void SysTick_init(int hz);
extern void sleep_thread(unsigned int ticks);
```

<프로그램 설명>

빨간색 글씨만 추가로 작성한 코드이다.

sleep_thread 함수를 사용할 수 있도록 sleep_thread 함수의 prototype 을 선언하였다.

```
// Thread stacks and functions.
```

```
extern unsigned int thread0_stack[];  
extern unsigned int thread1_stack[];  
extern unsigned int thread2_stack[];  
extern unsigned int thread3_stack[];  
extern unsigned int thread4_stack[];  
extern unsigned int thread5_stack[];  
extern unsigned int thread6_stack[];
```

```
extern void thread0_function(void);  
extern void thread1_function(void);  
extern void thread2_function(void);  
extern void thread3_function(void);  
extern void thread4_function(void);  
extern void thread5_function(void);  
extern void thread6_function(void);
```

<프로그램 설명>

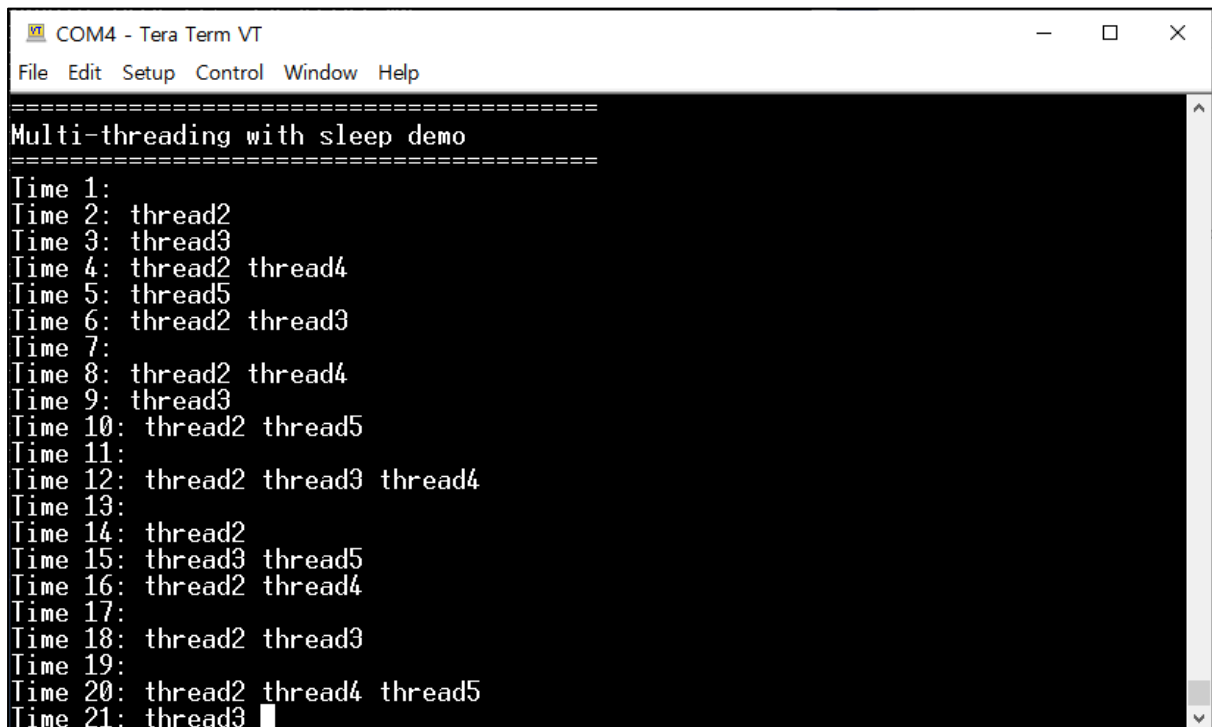
thread 를 생성(create)할 때 thread0 부터 thread6 까지의 thread 의 context 정보를 임시로 저장할 수 있는 공간인 threadX_stack[]을 사용하기 위해 unsigned int 형의 threadX_stack[]을 선언하였다. 이때 threadX_stack[] 에서의 X 는 0 부터 6 까지의 정수이다.

thread0 부터 thread6 까지의 threadX_function()함수를 사용할 수 있도록 threadX_function()함수의 prototype 을 선언하였다. 이때 threadX_function()에서의 X 는 0 부터 6 까지의 정수이다.

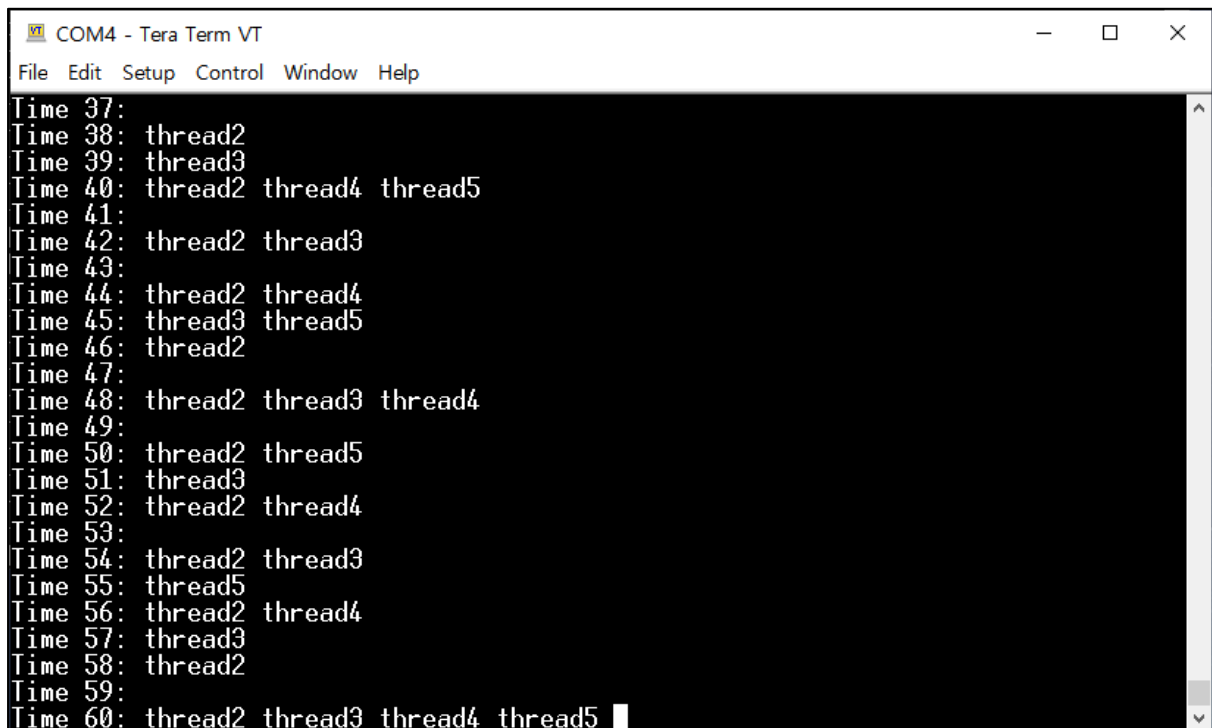
여기서 모든 변수와 함수는 다른 소스파일에서도 접근할 수 있도록 extern 을 사용하였다.

3.3 수행 시연

다음 첨부한 사진은 threads-sleep.bin 파일을 보드에서 수행하였을 때 보여지는 화면이다.



```
COM4 - Tera Term VT
File Edit Setup Control Window Help
=====
Multi-threading with sleep demo
=====
Time 1:
Time 2: thread2
Time 3: thread3
Time 4: thread2 thread4
Time 5: thread5
Time 6: thread2 thread3
Time 7:
Time 8: thread2 thread4
Time 9: thread3
Time 10: thread2 thread5
Time 11:
Time 12: thread2 thread3 thread4
Time 13:
Time 14: thread2
Time 15: thread3 thread5
Time 16: thread2 thread4
Time 17:
Time 18: thread2 thread3
Time 19:
Time 20: thread2 thread4 thread5
Time 21: thread3
```



```
COM4 - Tera Term VT
File Edit Setup Control Window Help
Time 37:
Time 38: thread2
Time 39: thread3
Time 40: thread2 thread4 thread5
Time 41:
Time 42: thread2 thread3
Time 43:
Time 44: thread2 thread4
Time 45: thread3 thread5
Time 46: thread2
Time 47:
Time 48: thread2 thread3 thread4
Time 49:
Time 50: thread2 thread5
Time 51: thread3
Time 52: thread2 thread4
Time 53:
Time 54: thread2 thread3
Time 55: thread5
Time 56: thread2 thread4
Time 57: thread3
Time 58: thread2
Time 59:
Time 60: thread2 thread3 thread4 thread5
```

4. 개발 과정

4.1 개발 일정

날짜	개발 일정
11/26(목) ~ 11/27(금)	역할 분담
11/28(토) ~ 11/29(일)	주어진 문제 분석, 1.1~1.3에 관한 토의
11/30(월) ~ 12/01(화)	2.1~2.3 문제 분석
12/01(화) ~ 12/03(목)	2.1~2.3 코드 작성 및 버그 수정
12/04(금) ~ 12/06(일)	3.1~3.3 문제 분석과 코드 작성 및 버그 수정, 코드 설명 추가
12/06(일) ~ 12/10(목)	보고서 작성 및 보완

4.2 업무 분담

	성명	업무
코드	이학민	회의 진행, 코드 작성 및 참고 문헌 탐색
	강수연	코드 작성, 보완 아이디어 제공 및 참고문헌 탐색
	박예슬	코드 작성, 보완 아이디어 제공 및 참고문헌 탐색
보고서	안무진	문제 기술 작성, 참고문헌 탐색, 회의록 작성
	김수혁	문제 풀이 작성 및 보고서 디자인 수정
	인정환	참고 문헌, 개발 과정 정리 및 회의록 작성

4.3 개발 시 발생한 문제 및 해결한 방법

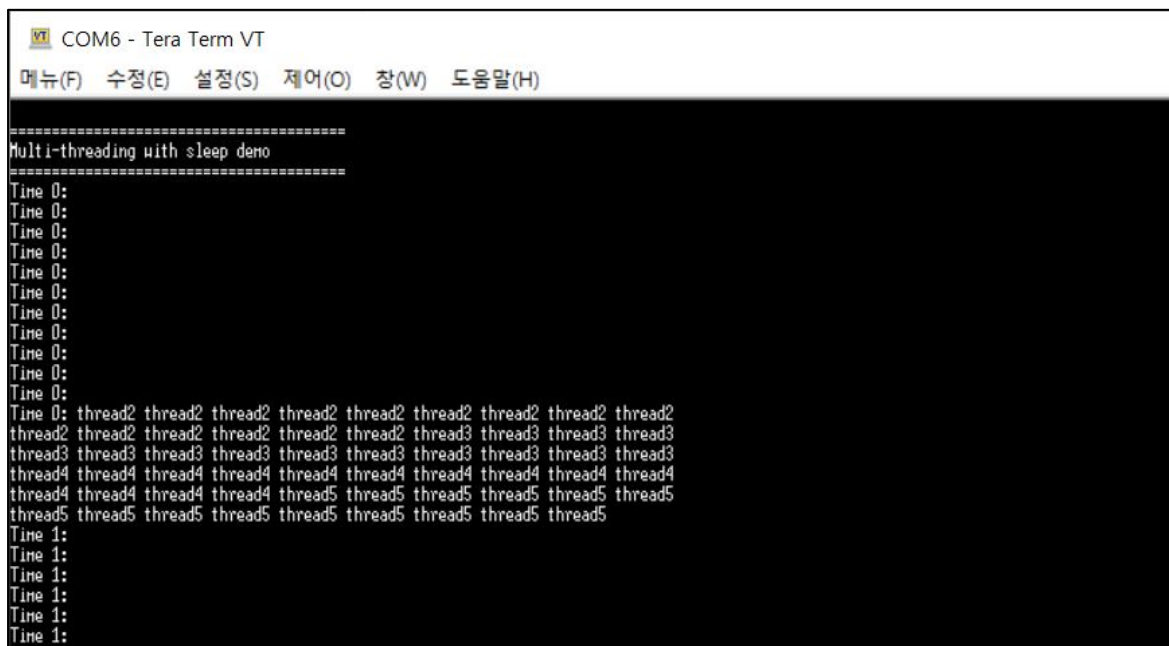
1)

코로나19 사회적 거리두기 2단계 격상으로 인해 본 마이크로프로세서 교과목의 수업 방식이 비대면으로 전환되었고 이에 따라 팀프로젝트에 관한 논의를 오프라인에서 진행하는 것이 어려워졌다. 따라서 온라인 만남을 택하기로 하였고 카카오톡 채팅방을 개설하여 조원들과 팀 프로젝트를 진행하기로 결정하였다. 본 보고서의 회의록의 마지막에 카카오톡 채팅방에서 팀원들이 상호 작용하는 모습을 화면 캡처하여 첨부하였다.

2)

기본 문제 코드 구현 후 보드에서 수행해보았을 때 각 thread0부터 thread5까지는 출력이 되었으나 tick의 값이 증가하지 않았다. 코드를 조원들과 함께 분석하고 토의한 끝에 SysTick_handler의 tcb_array[tid_current]에서 tid_current의 최대값은 (NO_OF_THREADS-1)임에도 불구하고 (NO_OF_THREADS-1)을 넘어가서 프로그램이 원하는 대로 작동되지 않는다는 것을 알게 되었다. tid_current의 값이 (NO_OF_THREADS-1)을 넘어가지 않도록 tid_current = tid_current+1에 %NO_OF_THREADS를 추가해 tid_current가 0 ~ (NO_OF_THREADS-1) 사이에서만 변할 수 있도록 만들어 주었고 tick이 증가되지 않는 문제가 해결되었다.

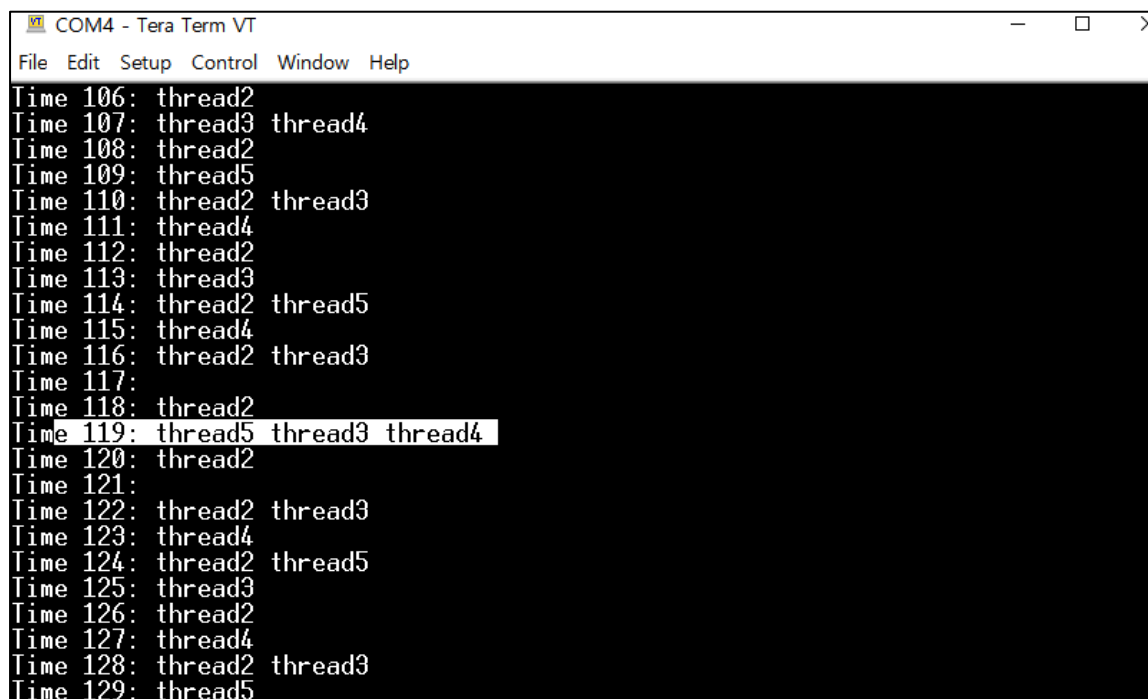
3) 추가 문제 구현 시 생긴 오류 1



추가 문제 코드 구현 후 보드에서 수행하였을 때 같은 Time과 thread가 여러 번 출력되는 현상이 발생하였다. 코드를 면밀히 분석해본 결과 threads.c 파일에서의 threadX_function부분에서 sleep_thread가 수행되고 sleep_tick이 0이 될 때까지 sleep_thread가 종료되지 않아야 thread를 출력하는 printf가 한 번씩만 수행된다는 것을 알게 되었다. 따라서 thread가 WAIT상태일 때는 아무 것도 출력하지 않는 printf("")를 반복하며 sleep_thread를 강제로 빠져나가지 못하게 코드를 만들었다.

4) 추가 문제 구현 시 생긴 오류 2

```
=====
Multi-threading with sleep demo
=====
Time 1:
Time 2: thread2
Time 3: thread3
Time 4: thread2 thread4
Time 5: thread5
Time 6: thread2 thread3
Time 7:
Time 8: thread2 thread4
Time 9: thread3 thread5
Time 10: thread2
Time 11:
Time 12: thread2 thread3 thread4
Time 13:
```



```
COM4 - Tera Term VT
File Edit Setup Control Window Help
Time 106: thread2
Time 107: thread3 thread4
Time 108: thread2
Time 109: thread5
Time 110: thread2 thread3
Time 111: thread4
Time 112: thread2
Time 113: thread3
Time 114: thread2 thread5
Time 115: thread4
Time 116: thread2 thread3
Time 117:
Time 118: thread2
Time 119: thread5 thread3 thread4
Time 120: thread2
Time 121:
Time 122: thread2 thread3
Time 123: thread4
Time 124: thread2 thread5
Time 125: thread3
Time 126: thread2
Time 127: thread4
Time 128: thread2 thread3
Time 129: thread5
```

4.3의 3)에서 발생한 문제를 해결하여 각 Time과 thread를 한번씩만 출력하게 만들고 난 뒤 thread5의 위치가 잘못되었고 thread 또한 순서대로 출력되지 않는 문제점이 발견되었다. 이를 해결하기 위하여 SysTick_handler에서 tcb_next를 선정하기 전 tid_current를 0으로 초기화하였다. tid_current가 0부터 순차적으로 조건문에 대응되면 위와 같은 문제점을 해결할 수 있었다.

4.4 회의록 첨부

회 의 록		
11/26(목)	안 건	프로젝트 전체의 진행을 담당할 팀장 결정, 역할 분담
	결 과	이학민 학생의 지원으로 팀장이 결정되었다. 역할 분담 투표를 만들어 놓고 11/27(금)까지 맡고 싶은 역할을 투표하기로 하였다.
11/27(금)	안 건	역할 분담 투표 마무리 및 프로젝트 업무 상세 분담
	방 식	카카오톡의 공개 중복 허용 투표를 이용하였다. (온라인 투표)
	결 과	<p>코드 업무 : 이학민, 강수연, 박예슬, 안무진(중복투표)</p> <p>보고서 업무 : 김수현, 인정환, 안무진(중복투표)</p> <p>중복 투표한 안무진 학생이 보고서 업무를 맡으면서 코드 업무 3명 보고서 업무 3명으로 역할을 분담하였다.</p> <div data-bbox="691 1102 1189 1505"> </div> <div data-bbox="691 1532 1189 2004"> </div>

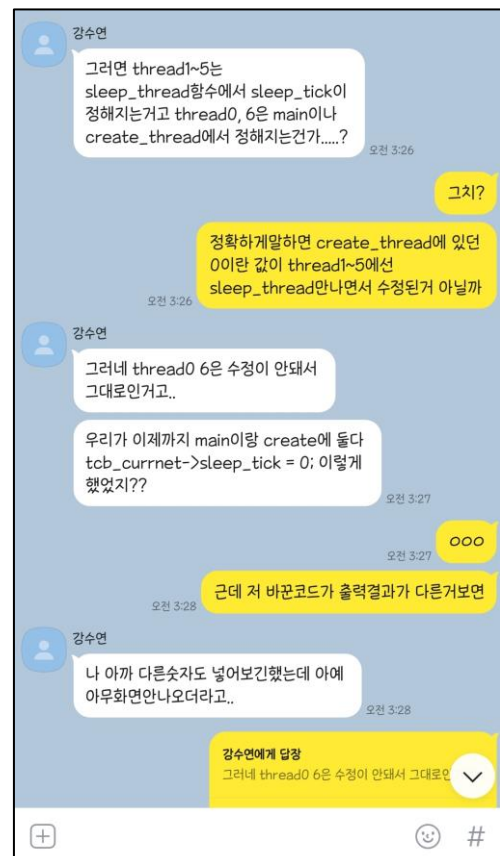
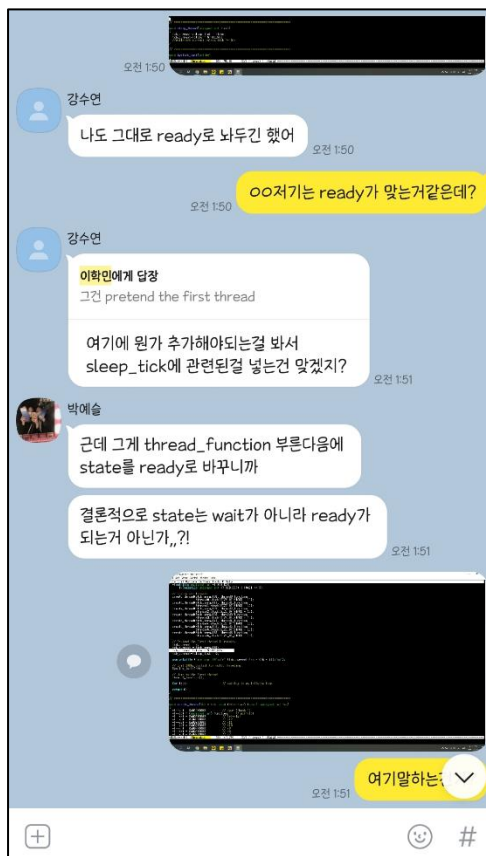
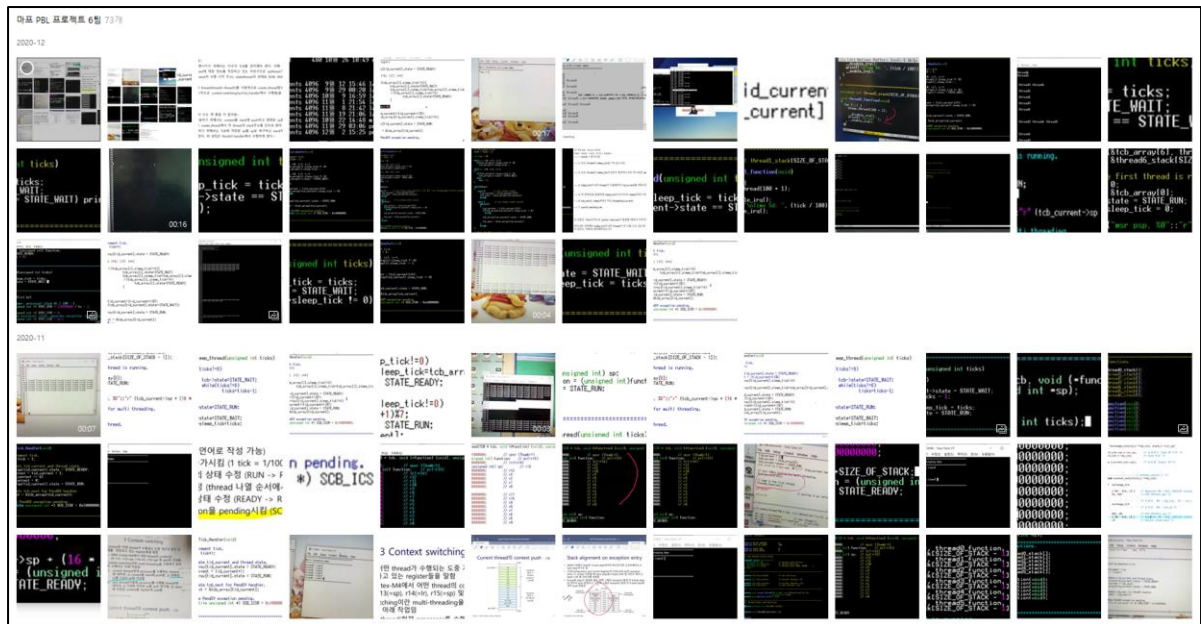
		<p>보고서 담당 학생인 안무진, 김수혁, 인정환 3명이 카카오톡 대화방에서 의견을 나눈 결과, 안무진 학생이 보고서 1번 작성, 김수혁 학생이 보고서 2,3번 작성, 인정환 학생이 보고서 4번을 작성하기로 결정되었다.</p> <p>또한 코드 작성자는 코드 작성을 할 때 실시간으로 토의하며 진행하는 방식을 택하였다.</p>
11/29(일)	안 건	본 프로젝트 문제에 대한 기초 조사
	방 식	Multi-thread context switching의 개념, 중요성, 활용성을 조사한 후 정리하였다. 안무진 학생이 인텔 공식 홈페이지 등을 참고하여 조사하고 정리한 자료를 카카오톡 채팅방에 공유하였다.
12/01(화)	안 건	기본 문제 조사 및 project-threads 폴더 함수의 prototype 이해
	방 식	project-multi-thread context switching 파일의 기본 문제 내용을 숙지한 후 의견을 교환하였다. 이학민 학생이 context-switching의 기본 개념과 어떻게 코드를 구현할지에 대해 간단하게 정리하여 카카오톡 채팅방에 공유하였다. 코드 작성자들은 문제 내용을 숙지한 후 코드 작성을 시작하였다.
12/02(수)	안 건	코드 작성 진행 상황 실시간 보고
	방 식	코드 작성자들은 자신이 작성한 코드가 업데이트 될 때마다 카카오톡 채팅방에 자신이 작성한 내용을 공유하여 업무의 효율성을 높였다. 기본 문제의 코드 작성을 마쳤다.
12/03(목)	안 건	작성한 프로그램 코드에 대한 상호작용
	결 과	코드 작성자들이 자신이 짠 코드에 대해 설명하는 시간을 가졌다. 코드 설명을 하며 코드에서의 출력 결과를 다시 한번 점검해 보았고 이상이 없었다. 박예슬 학생의 의견을 수렴하여 tid_current의 변화 범위를 수정하는 코드를 if문 대신 %연산자를 사용하여 간단하게 줄였다. 또한 인정환 학생의 의견을 수렴하여 %5 대신 %NO_OF_THREADS를 사용하여 추가 문제와의 통일성을 높였다.
12/04(금)	안 건	추가 문제 조사 및 project-threads-sleep 폴더 내 함수 prototype 이해
	방 식	project-multi-thread context switching 파일의 추가 문제 내용을 숙지한 후 토의하였다. 이 날 코드 작성자들은

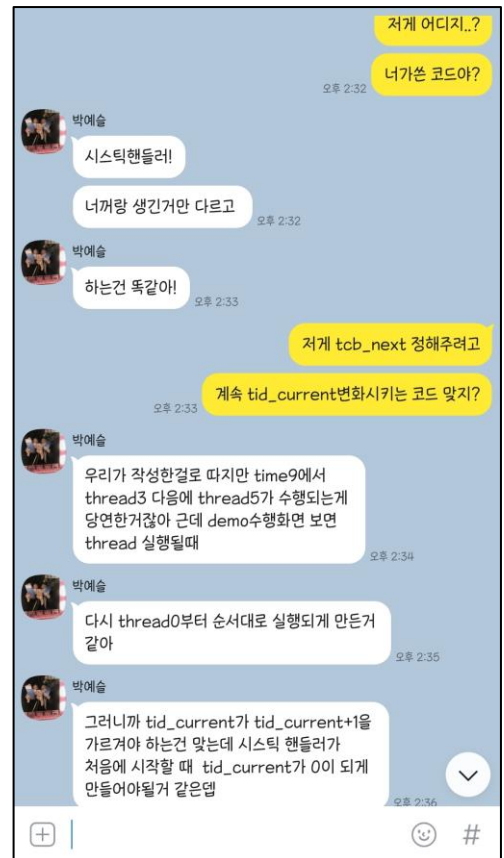
		추가 문제 내용을 숙지한 후 코드 작성을 시작하였다.
12/05(토)	안 건	코드 작성 진행 상황 실시간 보고
	방 식	코드 작성자들은 자신이 작성한 코드가 업데이트 될 때마다 카카오톡 채팅방에 자신이 작성한 내용을 공유하여 업무의 효율성을 높였다.
	안 건	작성된 코드의 간단한 설명 추가
	결 과	보고서 작성자들이 코드를 보다 쉽게 이해할 수 있도록 별도의 간단한 설명을 추가해달라는 의견이 나왔다. 따라서 코드 작성자들은 보고서 작성자들을 위하여 작성한 코드에 대한 기초적인 설명을 추가하기로 하였다. 박예슬 학생과 강수연 학생이 각각 보고서 2.1과 2.2의 흐름을 제시하였고 이학민 학생이 지금까지 작성한 전체적인 코드에 대하여 부가적인 설명을 덧붙였다.
12/06(일)	안 건	작성한 프로그램 코드에 대한 상호작용
	결 과	코드 작성자들이 자신이 짠 코드에 대해 설명하는 시간을 가졌다. 코드 설명을 하며 코드에서의 출력 결과를 다시 한번 점검해 보았고 이상이 없었다. 작성된 코드를 이해한 보고서 작성자들은 보고서의 문제 풀이 부분을 작성하기 시작하였다.
12/07(월)	안 건	문제 풀이에 사용된 마이크로프로세서 기술에 대한 상호작용
	방 식	김수혁 학생이 풀이에 사용된 마이크로프로세서 기술에 대해 조사하여 카카오톡 채팅방에 공유하였다. 조사한 내용을 다시 한번 정리하여 보고서 2.2 부분의 작성을 마쳤다.
12/08(화)	안 건	보고서 디자인
	방 식	작성한 코드에 대한 가독성이 떨어진다는 의견이 나왔다. 따라서 김수혁 학생이 보고서의 가독성을 높이기 위하여 작성한 코드 부분의 디자인을 수정하였다.
12/09(수)	안 건	회의록 정리
	방 식	인정환 학생이 그동안 작성해온 회의록을 보고서 4.4 부분에 추가하였다.
12/10(목)	안 건	보고서 최종 점검 및 제출
	방 식	팀원이 전부 모여 보고서 취합 및 정리를 진행하였다. 보고서의 번호 서식을 보다 깔끔하게 정리하였고 더 수정할 것이 없다고 판단되어 팀장 이학민 학생이 대표로 보고서 파일을 e-campus에 제출하였다.

* 프로젝트 상호작용 자료 첨부

전체적인 프로젝트 진행과 프로그램 작성 및 코드 보완 관련한 상호작용은 다음과 같이 카카오톡으로 자신의 의견 또는 자신이 만든 코드를 공유하며 의견을 교환하는 식으로 진행하였다. 프로젝트 진행 중 오고 간 대화 내용은 단순히 프로젝트의 중간 과정이기 때문에 최종적인 코드 및 보고서와는 약간 다른 내용이 포함되어 있을 수 있음을 밝힌다.

<코드 작성 중 상호작용>





인정환

혹시 실행은 되나요?

그 SysTick_Handler 부분에

오후 1:45

네 실행 되는 코드예요. 보드에서 수행해보시면 됩니다

오후 1:46

인정환

tcb_next = &tcb_array[tid_current];
이부분 혹시 index가 왜 tid_current인지 설명해 주실 수 있나요?

오후 1:47

우리는 멀티스레딩을 구현하기 위하여 여러 task를 백로그 할당한다 합니다. 이 때 현재 task에서 다른 task에 대하여 강제적으로 task의 실행을 중단하여 주고, 이 때 백업된 다른 task의 실행을 continue 하고 그 뒤를 잇는 순서대로 재실행되는 형태를 context switching 이라고 하며 task의 context를 불러와 주어야 하는 과정은 실행하는 곳으로 구현됩니다. 이 과정을 context switching 이라고 하는 것입니다.

주어진 문제에서는 thread0부터 thread5까지의 순서대로 task가 주어진다고 멀티스레딩으로 수행되기 때문에 context switching 시 주어진 task의 백로그가 바뀌어가는 것을 볼 수 있습니다.

context switching의 구현방식은 과정은 thread0 -> thread1 -> thread2 -> thread3 -> thread4 -> thread5 -> thread0 -> ... 이 되는 식입니다.

tcb_current는 지금 thread의 번호를 나타내는데 변수라고 할 수 있습니다. (thread의 경우 tcb_current = 0 이면 tcb_current = 1의 주소를 가리키며 즉 0-5 사이에서 번호가 바뀌게 하여)

tcb_current = &tcb_array[tid_current] 이라고 하는 것을 볼수있습니다.

tcb_next는 다음 thread에 대한 변수이고, tid의 값을들여 tcb_current의 value 이 1을 가질때마다 tcb_next = &tcb_array[tid_current] 이라고 하면 tcb_next = &tcb_array[tid_current]로 바뀌게 됩니다.

오후 2:04

인정환

감사합니다

오후 2:05

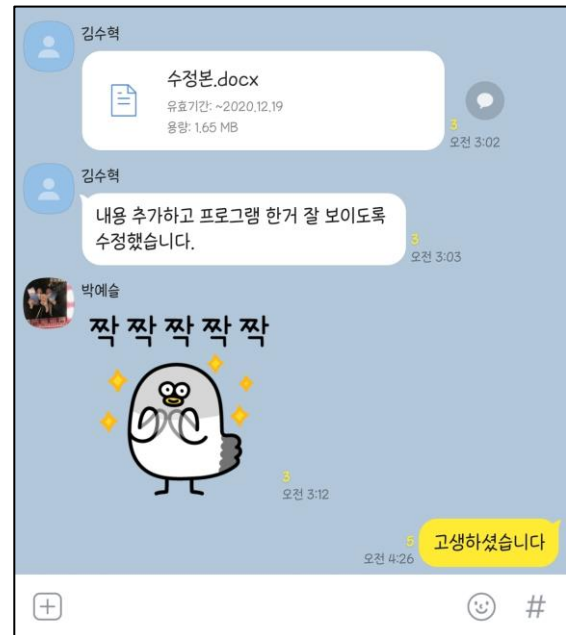
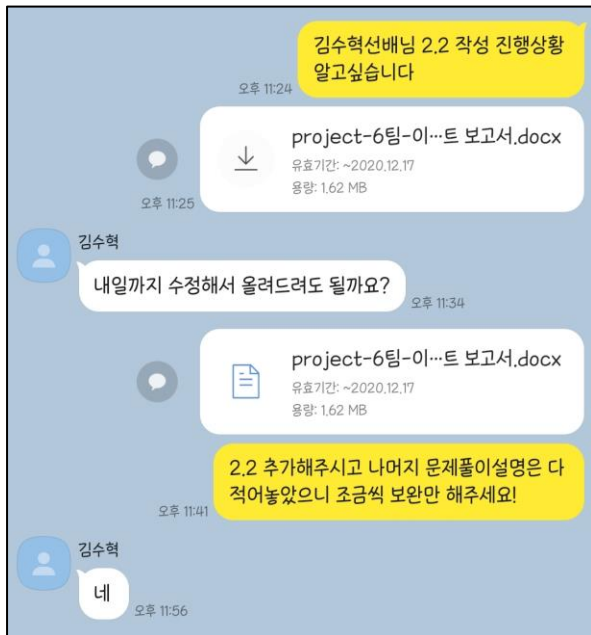
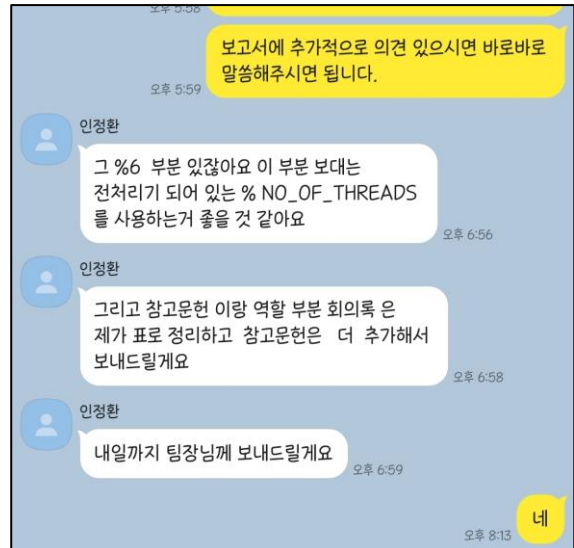
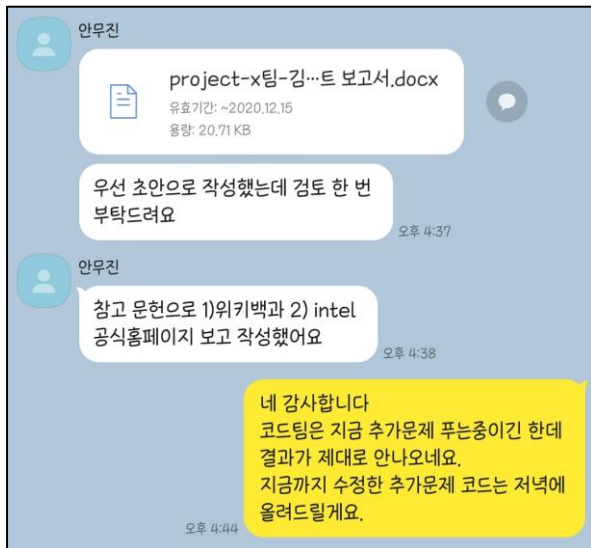
다른분들이 답이 필요하실 수도 있으니까 간단하게만 적어놨어요. 콘텍스트 스위칭의 기본적인 개념도 써놔요니 이해가 잘 안되시는 분은 한번 읽어보세요.

오후 2:05

인정환

팀장님 혹시 두번째 문제도 하실 생각인가요?

오후 2:06



5. 참고 문헌

5.1 참고 문헌

- 1) project-multi-thread context switching 및 본 마이크로프로세서 교과목 전체 강의자료
- 2) 2010-ARM-Cortex-M4 Devices Generic Guide
- 3) 2015-ARM-ARM Cortex-M4 Technical Reference Manual
- 4) <https://kldp.org/node/90993>
- 5) <https://www.youtube.com/watch?v=-4HKhwIH3FQ>
- 6) <https://blog.naver.com/1seola/221927602331>
- 7) 인텔 공식 홈페이지 www.intel.co.kr
- 8) 위키백과 ko.wikipedia.org

끝.