**강의명: 실시간 커널**

**실습 번호: 4**

**실습 제목: Semaphore(세마포)**

**학생 이름: 황귀훈**

**학번: 201710885**

**1. 태스크 동기화**

**1.1**

번호1 태스크 동기화는 실시간 커널 uC/OS-II의 세마포 서비스를 사용하여 두 태스크 Task1() 및 Task2()의 수행 과 정을 동기화 하는 프로그램으로 두개의 Task가 공유 자원인 screen에 서로 돌아가면서 출력한다. 이 때 Task1()이 출력을 마치면 Task2()가 출력하고, Task2가 출력을 마치면 Task1()이 출력을 해야 한다. 즉 두 태스크를 상호 배제해야 한다. 따라서 이 문제를 해결하기 위해 Sem1, Sem2로 2개의 세마포를 사용해야한다. Sem1은 Task1이 pending하기 위한 용도이며 Sem2는 Taks2가 pending하기 위한 용도로 사용된다. 모범 수행 화면과 같이 Task1()이 먼저 출력 되어야 하기 때문에 Sem1은 초기값을 1로 생성하고, Sem2는 초기값을 0으로 생성하여 Task1()이 먼저 출력된 뒤 Task2()가 출력되도록 하였다.

**1.2**

**void Task1(void \*pdata)**

**{**

**unsigned char err;**

**for(;;){**

**OSSemPend(Sem1, 0, &err);**

**UARTputcolor(31);**

**UARTprintf("[Task1]");**

**OSSemPost(Sem2);**

**OSTimeDly(OS\_GetRandom(200));**

**}**

**}**

Task1()은 붉은 글씨로 "[Task1]"을 출력한 후 OS\_GetRandom(200) tick 시간 동안 wait하는 일을 반복 수행해야 한다.

두 태스크를 상호 배제하기 위해 두개의 세마포를 각각 Sem1의 초기값=1, Sem2의 초기값=0으로 생성한 상태이다. Task1이 OSSemPend(Sem1, 0, &err)를 통해 공유자원인 screen에 출력하는 동안 Sem2의 경우 초기값이 0이기 때문에 Task1이 수행되는 동안 Task2를 수행할 수 없어 Task1이screen을 독점하는 것이다.

UARTputcolor(31)을 함으로써 출력되는 문구의 색을 붉은 색으로 할당하고 UARTprintf()을 통해 “[Task1]”이 출력되도록 설정하였다. 출력을 마치면 Task2가 수행될 수 있도록 Sem2를 post하고 OSTimeDly(OS\_GET(Random(200))를 호출함으로써 0~199의 범위의 랜덤값 tick 시간 동안 wait 하도록 했다. 이러한 과정을 반복 수행하기 위해 for(;;) 무한루프 속에 프로그램을 작성 하였다.

**void Task2(void \*pdata)**

**{**

**unsigned char err;**

**for(;;){**

**OSSemPend(Sem2, 0, &err);**

**UARTputcolor(34);**

**UARTprintf("[Task2]");**

**OSSemPost(Sem1);**

**OSTimeDly(OS\_GetRandom(200));**

**}**

**}**

Task2()는 푸른 글씨로 "[Task2]"을 출력한 후 OS\_GetRandom(200) tick 시간 동안 wait하는 일을 반복 수행해야 한다.

Task1에서 post한 Sem2를 Task1이 pend하게된다. 이때, Sem2의 초기값은 0이었지만 Task1에서Post를 하게 되면 1이 증가되기 때문에 Sem2의 초기값이 0에서 1이 되어 Task2를 수행할 수 있다. UARTputcolor(34)을 함으로써 출력되는 문구의 색을 푸른 색으로 할당하고 UARTprintf()을 통해 “[Task2]”이 출력되도록 설정하였다. 출력을 마치면 Task1이 수행될 수 있도록 Sem1를 post하고 OSTimeDly(OS\_GET(Random(200))를 호출함으로써 0~199의 범위의 랜덤값 tick 시간 동안 wait 하도록 했다. 이러한 과정을 반복 수행하기 위해 for(;;) 무한루프 속에 프로그램을 작성 하였다.

**1.3**

텍스트이(가) 표시된 사진

자동 생성된 설명

**2. 생산자 소비자 문제**

**2.1**

생산자 소비자 문제는 생산자는 데이터를 만들어 버퍼에 저장하고, 소비자는 버퍼에 있는 데이터를 꺼내 소비하는 문제이다. 버퍼가 비어 있을 경우 소비자는 버퍼에서 데이터를 빼낼 수 없어 기다려야 하고, 버퍼가 꽉 차 있는 경우 생산자는 버퍼에 데이터를 넣을 수 없으므로 기다려야 한다. 이때 버퍼는 공유자원 이므로 버퍼에 대한 접근, 즉 버퍼에 데이터를 넣고 빼는 일들이 상호배제 되어야 한다. 세마포를 사용하여 공유 자원인 버퍼를 상호배제 할 것인데 Deadlock상태를 방지하기 위해 세마포 Sem\_full, Sem\_empty를 생성하여 소비자와 생산자에게 상태를 알려주고 한 Task가 세마포를 사용할 동안 다른 Task는 wait을 유지하도록 하였다. 또 하나의 공유 자원인 screen의 상호배제를 위해 세마포가 추가적으로 하나 더 필요하다.

**2.2**

생산자 소비자 문제는 Sem\_full, Sem\_empty, Sem 총 3개의 세마포가 필요하다.

Sem\_full은 소비자와 버퍼의 관계를 제어하는 세마포로 현재 버퍼에 들어있는 데이터의 수를 알려준다. 소비자의 세마포는 비어 있을 때 block이 되어야 하므로 초기 값을 0으로 설정하였다.

Sem\_empty는 생산자와 버퍼의 관계를 제어하는 세마포로 버퍼에 들어갈 수 있는 데이터의 수를나타낸다. 따라서, 초기값으로 BUFFER\_SIZE-1만큼의 값을 갖는다. 데이터가 계속 버퍼에 들어가는데 버퍼가 BUFFER\_SIZE-1만큼 들어가게 되면 생산자는 block이 되어야한다.

Sem은 공유자원인 screen을 각 task가 상호 배제할 용도의 세마포로 화면 출력을 위해 초기값을 1로 설정하였다.

Sem\_full: 초기값 = 0, 버퍼 내에 소비할 아이템이 있는지를 나타낸다.

Sem\_empty: 초기값 = buffer size-1, 버퍼 내에 저장할 공간이 있는지를 나타낸다.

Sem: 초기값 = 1, 공유자원인 screen에 대한 접근을 통제.

**2.3**

**void TaskProducer1(void \*pdata)**

**{**

**INT32U item;**

**INT8U err1, err2;**

**for(;;){**

**item = OS\_GetRandom(10000);**

**OSSemPend(Sem\_empty, 0, &err1);**

**OSSemPend(Sem, 0, &err2);**

**Buffer.Rear=AddToBuffer(item);**

**UARTputgoto(0,3);**

**UARTprintf("producer1 dly=%d: %d",producer\_delay,item);**

**OSSemPost(Sem\_full);**

**OSSemPost(Sem);**

**OSTimeDly(producer\_delay);**

**}**

**}**

TaskProducer1(): 이 태스크는 버퍼에 빈 공간이 있으면 OS\_GetRandom(10000) 값을 가지는 item을 버퍼에 넣고, 화면 위치 (0,3)에 이 태스크의 이름, producer\_delay tick 수 및 item을 출력한 후, producer\_delay tick 시간 동안 wait하는 작업을 반복한다.

모든 태스크는 반복적으로 일을 하기 때문에 모두 for(;;) 무한루프 안에 작성해야 한다.

먼저 item이라는 변수를 선언하고 OS\_GetRandom(10000)함수를 호출하여 0~9999까지의 랜덤 값을 item에 저장했다. 생산자와 버퍼의 관계를 제어하는 세마포인 Sem\_empty를 사용하여 버퍼에 남은 공간을 확인한다. 이때, Sem\_empty의 값이 0이 아니므로 pending을 진행할 수 있다. 또한, 공유자원인 screen을 다른 task들로부터 상호배제 시키기위해 세마포 Sem도 pending하여 사용하였다. Sem의 경우 초기값이 1이므로 pending이 가능하다.

이와 같이 상호배제된 상태에서 item을 AddtoBuffer()를 사용하여 버퍼에 넣어주었다. 그리고 화면 위치 (0,3)에 이 태스크의 이름, producer\_delay tick 수 및 item을 출력하기 위해 UARTputgoto(0,3)를 호출하여 커서를 옮긴 뒤 UARTprintf()를 사용하여 태스크의 이름, producer\_delay tick 수, item값을 출력했다.

출력을 마친 뒤 Sem\_full을 post함으로써 세마포 Sem\_full의 값이 1 증가시켰다. 이것은 버퍼에 item 하나가 들어왔음을 알려준다. 그리고 출력을 마쳤으므로 Sem을 post하여 다른 Task가 출력을 수행 할 수 있도록 하였다. 마지막으로 producer\_delay tick 시간 동안 wait 하도록 OSTimeDly()을 사용했다. TaskProducer1에서 사용한 Sem\_empty와 Sem\_full은 유한한 버퍼를 사용하기 위해 동기화하는 기능을 수행한다. TaskProducer1은 버퍼에 데이터가 입력되는 것을 세마포인 Sem\_full을 통해 TaskConsumer에게 알려주는 작업을 한다.

**void TaskProducer2(void \*pdata)**

**{**

**INT32U item;**

**INT8U err1, err2;**

**for(;;){**

**item = OS\_GetRandom(10000);**

**OSSemPend(Sem\_empty, 0, &err1);**

**OSSemPend(Sem, 0, &err2);**

**Buffer.Rear=AddToBuffer(item);**

**UARTputgoto(0,4);**

**UARTprintf("producer2 dly=%d: %d",producer\_delay,item);**

**OSSemPost(Sem\_full);**

**OSSemPost(Sem);**

**OSTimeDly(producer\_delay);**

**}**

**}**

TaskProducer2(): 이 태스크는 버퍼에 빈 공간이 있으면 OS\_GetRandom(10000) 값을 가지는 item을 버퍼에 넣고, 화면 위치 (0,4)에 이 태스크의 이름, producer\_delay tick 수 및 item을 출력한 후 producer\_delay tick 시간 동안 wait하는 작업을 반복한다.

모든 태스크는 반복적으로 일을 하기 때문에 모두 for(;;) 무한루프 안에 작성해야 한다.

먼저 item이라는 변수를 선언하고 OS\_GetRandom(10000)함수를 호출하여 0~9999까지의 랜덤 값을 item에 저장했다. 생산자와 버퍼의 관계를 제어하는 세마포인 Sem\_empty를 사용하여 버퍼에 남은 공간을 확인한다. 이때, Sem\_empty의 값이 0이 아니므로 pending을 진행할 수 있다. 또한, 공유자원인 screen을 다른 task들로부터 상호배제 시키기위해 세마포 Sem도 pending하여 사용하였다. Sem의 경우 TaskProducer1이 출력을 마친 뒤 post하여 1이므로 pending이 가능하다.

이와 같이 상호배제된 상태에서 item을 AddtoBuffer()를 사용하여 버퍼에 넣어주었다. 그리고 화면 위치 (0,4)에 이 태스크의 이름, producer\_delay tick 수 및 item을 출력하기 위해 UARTputgoto(0,4)를 호출하여 커서를 옮긴 뒤 UARTprintf()를 사용하여 태스크의 이름, producer\_delay tick 수, item값을 출력했다.

출력을 마친 뒤 Sem\_full을 post함으로써 세마포 Sem\_full의 값이 1 증가시켰다. 이것은 버퍼에 item 하나가 들어왔음을 알려준다. 그리고 출력을 마쳤으므로 Sem을 post하여 다른 Task가 출력을 수행 할 수 있도록 하였다. 마지막으로 producer\_delay tick 시간 동안 wait 하도록 OSTimeDly()을 사용했다. TaskProducer2에서 사용한 Sem\_empty와 Sem\_full은 유한한 버퍼를 사용하기 위해 동기화하는 기능을 수행한다. TaskProducer2은 버퍼에 데이터가 입력되는 것을 세마포인 Sem\_full을 통해 TaskConsumer에게 알려주는 작업을 한다.

**void TaskConsumer1(void \*pdata)**

**{**

**INT32U item;**

**INT8U err1, err2;**

**for(;;){**

**OSSemPend(Sem\_full, 0, &err2);**

**OSSemPend(Sem, 0, &err1);**

**item=RemoveFromBuffer();**

**UARTputgoto(0,6);**

**UARTprintf("consumer1 dly=%d: %d",consumer\_delay,item);**

**OSSemPost(Sem\_empty);**

**OSSemPost(Sem);**

**OSTimeDly(consumer\_delay);**

**}**

**}**

TaskConsumer1(): 이 태스크는 버퍼에 어떤 item이 있으면 버퍼에서 item을 제거하고, 화면 위치 (0, 6)에 이 태스크의 이름, consumer\_delay tick 수 및 제거한 item을 출력한 후 consumer\_delay tick 시간 동안 wait하는 작업을 반복한다.

모든 태스크는 반복적으로 일을 하기 때문에 모두 for(;;) 무한루프 안에 작성해야 한다.

먼저 버퍼에 어떤 item이 있는지 판단하기 위해 OSSemPend()를 사용하여 Sem\_full의 값을 확인하여 0이 아닌 경우 버퍼에 소비할 데이터가 들어있음을 파악한다. 이때, 위의 TaskProducer에서 post(Sem\_full,0,&err2)을 하였기에 Sem\_full의 값이 1증가하였으므로 소비할 데이터가 들어있다. 만약 0이라면 기다린다.

또한, 공유자원인 screen을 다른 task의 사용으로부터 상호배제 시키기위한 목적으로 Sem을 pend하는 과정을 진행하였다. 이처럼 상호배제된 상태에서 RemoveFromBuffer을 사용하여 버퍼에서 데이터를 제거한 후, item에 저장하였다

그 후 화면 위치 (0, 6)에 이 태스크의 이름, consumer\_delay tick 수 및 제거한 item을 출력하기 위해 UARTputgoto(0,6)을 호출하여 커서를 옮긴 뒤 UARTprintf()를 이용하여 출력했다.

이와 같은 TaskConsumer1의 작업을 모두 진행하였으므로 Sem\_empty를 post하는 작업을 진행함으로써 Sem\_empty의 값을 1 증가시켜 버퍼에 하나의 빈 자리가 추가되었다는 것을 알려준다. 또한, 출력을 마쳤기 때문에 Sem을 post하는 작업을 진행하였다.

마지막으로, TaskConsumer1이 consumer \_delay만큼의 tick동안 wait 하도록 OSTimeDly()을 사용했다. TaskConsumer1에서 사용한 Sem\_empty와 Sem\_full은 유한한 버퍼를 사용하기 위해 동기화하는 기능을 수행한다. TaskProducer1과 TaskProducer2는 버퍼에 데이터가 입력되는 것을 세마포인 Sem\_full을 통해 TaskConsumer에게 알려주는 작업을 했다면 TaskConsumer1은 버퍼에 빈 공간이 생성된 것을 Sem\_empty를 통해 TaskProducer에게 알려주는 기능을 한다.

**void TaskConsumer2(void \*pdata)**

**{**

**INT32U item;**

**INT8U err1, err2;**

**for(;;){**

**OSSemPend(Sem\_full,0,&err2);**

**OSSemPend(Sem,0,&err1);**

**item=RemoveFromBuffer();**

**UARTputgoto(0,7);**

**UARTprintf("consumer2 dly=%d: %d",consumer\_delay,item);**

**OSSemPost(Sem\_empty);**

**OSSemPost(Sem);**

**OSTimeDly(consumer\_delay);**

**}**

**}**

TaskConsumer2(): 이 태스크는 버퍼에 어떤 item이 있으면 버퍼에서 item을 제거하고, 화면 위치 (0, 7)에 이 태스크의 이름, consumer\_delay tick 수 및 제거한 item을 출력한 후 consumer\_delay tick 시간 동안 wait하는 작업을 반복한다.

모든 태스크는 반복적으로 일을 하기 때문에 모두 for(;;) 무한루프 안에 작성해야 한다.

먼저 버퍼에 어떤 item이 있는지 판단하기 위해 OSSemPend()를 사용하여 Sem\_full의 값을 확인하여 0이 아닌 경우 버퍼에 소비할 데이터가 들어있음을 파악한다. 이때, 위의 TaskProducer에서 post(Sem\_full,0,&err2)을 하였기에 Sem\_full의 값이 1증가하였으므로 소비할 데이터가 들어있다. 만약 0이라면 기다린다.

또한, 공유자원인 screen을 다른 task의 사용으로부터 상호배제 시키기위한 목적으로 Sem을 pend하는 과정을 진행하였다. 이처럼 상호배제된 상태에서 RemoveFromBuffer을 사용하여 버퍼에서 데이터를 제거한 후, item에 저장하였다

그 후 화면 위치 (0, 7)에 이 태스크의 이름, consumer\_delay tick 수 및 제거한 item을 출력하기 위해 UARTputgoto(0,7)을 호출하여 커서를 옮긴 뒤 UARTprintf()를 이용하여 출력했다.

이와 같은 TaskConsumer2의 작업을 모두 진행하였으므로 Sem\_empty를 post하는 작업을 진행함으로써 Sem\_empty의 값을 1 증가시켜 버퍼에 하나의 빈 자리가 추가되었다는 것을 알려준다. 또한, 출력을 마쳤기 때문에 Sem을 post하는 작업을 진행하였다.

마지막으로, TaskConsumer2이 consumer \_delay만큼의 tick동안 wait 하도록 OSTimeDly()을 사용했다. TaskConsumer2에서 사용한 Sem\_empty와 Sem\_full은 유한한 버퍼를 사용하기 위해 동기화하는 기능을 수행한다. TaskProducer1과 TaskProducer2는 버퍼에 데이터가 입력되는 것을 세마포인 Sem\_full을 통해 TaskConsumer에게 알려주는 작업을 했다면 TaskConsumer2은 버퍼에 빈 공간이 생성된 것을 Sem\_empty를 통해 TaskProducer에게 알려주는 기능을 한다.

**void TaskPrinter(void \*pdata)**

**{**

**INT32S key;**

**INT32U tick, task, cpu;**

**INT8U err;**

**for(;;){**

**key = 1;**

**tick = OSTimeGet();**

**task = OSTaskCtr;**

**cpu = OSCPUUsage;**

**OSSemPend(Sem,0,&err);**

**UARTputgoto(0,1);**

**UARTprintf("Ticks=%d, Tasks=%d, CPU=%d%%\n", tick, task, cpu);**

**UARTputgoto(0,5);**

**UARTwrite("^[[K",6);**

**PrintBuffer();**

**while(key != -1) {**

**key = UARTgetc\_nb();**

**if(key == '+') {**

**if(consumer\_delay < 250)**

**consumer\_delay+=10;**

**}**

**else if(key == '-') {**

**if(consumer\_delay > 150)**

**consumer\_delay-=10;**

**}**

**}**

**OSSemPost(Sem);**

**OSTimeDly(printer\_delay);**

**}**

**}**

TaskPrinter(): 이 태스크는 화면 위치 (0,1)에 현재 tick 시간, 태스크 수, CPU 사용도를 출력하고, 화면 위치 (0,5)에 이 태스크의 이름, 버퍼 내 item의 개수 및 모든 item을 출력한 후 printer\_delay tick 시간 wait하는 작업을 반복한다. 또한 이 태스크는 키보드로부터 입력 받은 문자 '+' 및 '-'에 대하여 consumer\_delay를 10 증가 및 10 감소 작업을 수행한다. 이때 consumer\_delay는 150 이상이며 250 이하의 값으로만 수정할 수 있다

모든 태스크는 반복적으로 일을 하기 때문에 모두 for(;;) 무한루프 안에 작성해야 한다.

공유자원인 screen을 여러 task가 사용하기 때문에 이를 상호배제해주기 위한 목적으로 Sem을 pend해주었다. 화면 위치 (0,1)에 현재 tick 시간, 태스크 수, CPU 사용도를 출력하기 위해 UARTputgoto(0,1)을 호출하여 커서를 옮긴 뒤 UARTprintf()를 사용하여 미리 OSTimeGet(), OSTaskCtr(), OSCPUUsage()함수를 통해 저장한 현재 tick 시간, 태스크 수, CPU 사용도를 출력했다.

또한 화면 위치 (0,5)에 이 태스크의 이름, 버퍼 내 item의 개수 및 모든 item을 출력 하기 위해UARTputgoto(0,5)을 호출하여 커서를 옮긴 뒤 PrintBuffer()를 호출했다. 이때, 새로운 값의 출력이 깨끗하게 출력될 수 있도록 ANSI ESCAPE code를 사용하여 한 줄을 clear하도록 작성하였다.

또한 이 태스크는 키보드로부터 입력 받은 문자 '+' 및 '-'에 대하여 consumer\_delay를 10 증가 및 10 감소 작업을 수행해야 되기 때문에 UARTgetc\_nb()를 호출하여 key변수에 키보드로부터 입력 받은 문자를 저장하고 그 입력 받은 키 값을 다 뽑아 처리하기 위해 while문을 사용했다. 주어진 조건(consumer\_delay는 150 이상이며 250 이하의 값)안에서 입력 받은 문자 '+' 및 '-'에 대하여 consumer\_delay를 10 증가 및 10 감소 작업을 수행하도록 했다. 이때 +를 하면 consumer\_delay가 증가하기 때문에 소비자가 버퍼에 접근하는 주기가 늘어나고, -를 하면 consumer\_delay가 감소하기 때문에 소비자가 버퍼에 접근하는 주기가 줄어든다.

출력을 모두 마친 뒤 공유자원(screen)을 모두 사용하였기에 Sem을 post 했고, printer\_delay tick 시간 wait하기 위해 OSTimeDly()을 호출했다.

**2.4**

텍스트이(가) 표시된 사진

자동 생성된 설명

초기 수행 화면으로 sme-prod-cons-demo.bin과 동일한 형태로 출력되어 프로그램이 잘 수행됨을 확인했다.

**2.5**

텍스트이(가) 표시된 사진

자동 생성된 설명

+를 하면 consumer\_delay가 증가하기 때문에 소비자가 버퍼에 접근하는 주기가 늘어난다. 이로 인해 producer가 생산한 데이터가 버퍼에 가득 쌓이고 소비자는 이 속도에 맞추지 못하기 때문에 데이터를 빨리 소비할 수 없어 버퍼가 가득 차게 된다. 또한, 버퍼가 full이 되었기 때문에 생산자는 더 많은 데이터를 생산할 수 없다.

**2.6**

텍스트이(가) 표시된 사진

자동 생성된 설명

-를 하면 consumer\_delay가 감소하기 때문에 소비자가 버퍼에 접근하는 주기가 줄어든다. 이로 인해 cousumer가 버퍼에서 데이터를 빨리 소비하게 되고 생산자는 이 속도에 맞추지 못해 버퍼가 empty상태인 경우가 생기게 된다. 이로 인해 consumer는 empty 버퍼에서 데이터를 소비하지 못하는 경우도 발생하게 된다.

**3. 식사하는 철학자 문제**

**3.1**

식사하는 철학자 문제는 5명의 철학자가 식사를 하고 있는 상황을 가정한 문제이다. 철학자들은왼쪽, 오른쪽 포크를 동시에 가지고 있어야만 식사를 할 수 있다. 하지만 철학자들이 원형의 테이블에 앉아 있는데 철학자들 사이에는 포크 한 쌍이 아닌 하나씩만 놓여있다.

따라서, 포크를 세마포로 설정하여 세마포(=포크)의 값이 1이면 철학자가 그 포크를 들고 세마포(=포크)의 값이 0이라면 값이 1로 바뀔 때까지 계속해서 기다린다. 이때, 모든 철학자가 왼쪽/오른쪽 포크를 들 경우 모든 철학자들이 자신의 오른쪽/왼쪽 포크를 들 수 있을 때까지 기다려야 한다. 이는 DeadLock 현상이 발생한 것으로 표현할 수 있다.

따라서 DeadLock 현상을 방지하기 위해서는 한 철학자가 왼/오른쪽의 포크를 동시에 들어야한다.

이를 위해 왼쪽의 세마포(=포크)와 오른쪽의 세마포(=포크)를 동시에 사용하여야 한다.

즉, 하나의 Task는 2개의 세마포를 사용함으로써 다음 Task는 post 될 때까지 기다리게 되어 DeadLock 현상을 방지할 수 있는데 이는 다음과 같이 처리하였다.

먼저 왼쪽 포크를 pending하는 과정을 진행한다. 왼쪽 포크를 pend하면 세마포 값이 0 이된다. 이때, 만약 왼쪽 포크와 오른쪽 포크의 값이 같다면 세마포의 값이 둘다 0이므로 DeadLock 현상이 된다. 이럴 경우, 왼쪽 포크 세마포를 post한 후 OSTimeDly를 통해 랜덤시간 동안 wait에 갔다가 다시 시도해보도록 설정하였다. (비교를 통해 세마포 값이 같지 않다면 다음 과정인 오른쪽 포크를 드는 것으로 수행할 수 있도록 설정)

**3.2**

식사하는 철학자 문제를 해결하기 위해서는 총 6개의 세마포가 필요하다.

5개의 세마포는 철학자들 사이에 있는 포크를 나타내는 용도이며 위의 3.1에서 설명했듯이 세마포의 값이 1이면 철학자가 포크를 들 수 있는 것을 나타내고 세마포의 값이 0 이면 1이 될 때까지 기다려야 한다. 이때 문제에서 포크가 총 5개였기에 이러한 포크들의 상태를 나타내는 세마포는 5개가 필요하다.

이때, task가 2개의 세마포를 가져야 작업이 수행된다. 이때 초기에는 모든 철학자가 포크를 들을 수 있으므로 포크를 나타내는 용도의 세마포 5개의 초기값을 모두 1로 설정하였다.

또한, 공유자원인 screen의 상호배제를 위한 세마포가 추가적으로 필요하다. 이때, 화면 출력을 위해 세마포의 초기값을 1로 설정하였다. 이와 같이 3번 식사하는 철학자 문제를 해결하기 위해서는 총 6개의 세마포가 필요하다.

**3.3**

**void TaskPrint(void \*pdata)**

**{**

**INT32S key;**

**INT32U tick, task, cpu;**

**INT8U err;**

**for(;;)**

**{**

**tick = OSTimeGet();**

**task = OSTaskCtr;**

**cpu = OSCPUUsage;**

**key = UARTgetc\_nb();**

**OSSemPend(Sem, 0, &err);**

**UARTputgoto(0,0);**

**UARTputcolor(30);**

**UARTprintf("Real-time kernel: dining philosophers");**

**UARTputgoto(0,1);**

**UARTputcolor(30);**

**UARTprintf("Ticks=%d, Tasks=%d, CPU=%d%%\n", tick,task,cpu);**

**UARTputgoto(0,9);**

**UARTputcolor(30);**

**UARTprintf("Type any char to refresh the screen.");**

**if(key!=-1)**

**UARTputclear();**

**OSSemPost(Sem);**

**OSTimeDly(100);**

**}**

**}**

TaskPrint(): 이 태스크는 100 tick 시간 마다 화면 위치 (1,0)에 현재 시간(tick), 태스크 수, CPU 사용도(%)를 출력한다. 또한 어떤 키보드 입력이 들어오면 화면을 지우고 다시 출력하는 역할을 한다.

TaskPrint함수에서 사용되는 세마포는 공유자원인 screen을 상호배제하기 위함이다.

따라서, 무한 루프 속에서 OSSemPend(Sem, 0, &err)을 처음에 호출함으로써 screen의 상호배제를 형성한 후 작업을 진행한다.

모범 출력 화면과 같이 출력되기 위해 먼저 UARTputgoto(0,0)을 호출하여 커서를 옮긴 뒤 UARTprintf()를 사용하여 “Real-time kernel: dining philosophers”를 출력한다. 이 때 검은 색으로 출력하기 위해 UARTputcolor(30)을 호출한다.

그 다음 화면 위치 (1,0)에 출력하기 위해 먼저 UARTputgoto(1,0)을 호출하여 커서를 옮긴 뒤 처음 선언한 변수 tick, task, cpu에 각각 OSTimeGet(), OSTaskCtr(), OSCPUUsage()함수를 호출하여 반환 값을 저장한 뒤 UARTprintf()를 사용하여 현재 시간(tick), 태스크 수, CPU 사용도(%)를 출력한다. 이 때 검은 색으로 출력하기 위해 UARTputcolor(30)을 호출한다. 그 후 모범 수행 화면과 같이 출력되기 위해 UARTputgoto(0,9)을 호출하여 커서를 옮긴 뒤 UARTprintf()를 사용하여 “Type any char to refresh the screen”를 출력한다. 이 때 검은 색으로 출력하기 위해 UARTputcolor(30)을 호출한다. 또한 어떤 키보드 입력이 들어오면 화면을 지우고 다시 출력하기 위해 미리 선언한 key변수에 UARTgetc\_nb()함수를 호출하여 키보드 입력을 받아 만약 -1이 아니라면(키보드 입력이 있다면) UARTputclear()함수를 호출하여 화면을 지우고 다시 출력하도록 했다.

마지막으로 모든 출력이 끝나면 다른 task가 screen에 출력할 수 있도록 OSSemPost()를 사용하여 세마포 Sem을 post하고 이 모든 과정을 100 tick 시간 마다 반복하도록 OSTimeDly(100)을 작성하였다.

**void TaskPhilosopher(void \*pdata)**

**{**

**INT8U err;**

**int Left\_fork= (\*((int\*)pdata));**

**int Right\_fork= (\*((int\*)pdata)+1)%5;**

**for(;;){**

**OSSemPend(Sem,0,&err);**

**UARTputgoto(0, Left\_fork+3);**

**UARTputcolor(34);**

**UARTprintf("Philosopher %d is thinking...", Left\_fork);**

**OSSemPost(Sem);**

**OSTimeDly(200+OS\_GetRandom(50));**

**OSSemPend(Fork[Left\_fork],0,&err);**

**if(Fork[Right\_fork]==Fork[Left\_fork]){**

**OSSemPost(Fork[Left\_fork]);**

**OSTimeDly(OS\_GetRandom(30));**

**continue;**

**}**

**OSSemPend(Fork[Right\_fork],0,&err);**

**OSSemPend(Sem,0,&err);**

**UARTputgoto(0,Left\_fork+3);**

**UARTputcolor(31);**

**UARTwrite("^[[K",6);**

**UARTprintf("Philosopher %d is eating...", Left\_fork);**

**OSSemPost(Sem);**

**OSTimeDly(200+OS\_GetRandom(50));**

**OSSemPost(Fork[Right\_fork]);**

**OSSemPost(Fork[Left\_fork]);**

**}**

**}**

5개의 TaskPhilosopher(): 이 태스크들은 (200 + OS\_GetRandom(50)) tick 시간 동안 생 각하고, (200 + OS\_GetRandom(50)) tick 시간 동안 식사하는 일을 반복한다. 여기서 철 학자들은 둥근 테이블에 앉아 있고, 철학자는 자신의 왼쪽 및 오른쪽 포크를 둘 다 사 용할 수 있어야 식사할 수 있다. 이 태스크는 생각 혹은 식사하는 동안 각 철학자의 상태 출력 위치에 그 철학자가 수행하는 상태(생각 중은 푸른색, 식사 중은 붉은 색)를 출력한다.

전역변수로 설정한 int Philo[NO\_OF\_PHILOSOPHERS]와 TaskStart 함수 속에서 for문을 이용하여 Philo[0] ~ Philo[4] 총 5개의 pdata값을 생성하였다. 이 pdata값을 통해 OSSemCreate로 5개의 TaskPhilospher를 생성한 상태이다. 따라서, void TaskPhilosopher(void \*pdata)함수에서는 TaskPhilosopher0 ~ TaskPhilosopher4를 수행시켜야 한다.

TaskPhilosopher 함수에서는 pdata 값을 int형 변수로 설정하여 0~4의 값으로 표현해주었다. Left\_fork는 왼쪽에 위치한 포크(=세마포)를 의미하며 Right\_fork는 오른쪽에 위치한 포크(=세마포)를 의미한다. 왼쪽 포크인 Left\_fork에 int형 변수로 바꾼 pdata 값 0~4의 값을 할당, 오른쪽 포크인 Right\_fork에는 왼쪽포크에서 +1 된 값을 할당해주어야 한다 이때, 왼쪽포크가 4인 경우 오른쪽 포크가 +1되어 5가 될 수 있으므로 이를 방지 해주기 위해 %5로 값을 설정하였다.

철학자는 생각을 먼저 해야 하므로(세마포=포크를 기다리고 있다가 왼쪽/오른쪽 포크를 모두 들 수 있을 때 식사) “philosopher[0~4] is thinking”을 출력하는 과정부터 진행했다. UARTprintf문에서 Left\_fork는 철학자의 번호와 같으므로 몇 번째 철학자가 생각 중인지 나타낼 수 있다. 먼저, 공유자원인 screen을 상호배제하기 위해 Sem을 Pend하였으며 생각 중은 파랑색으로 출력해야 하므로UARTputcolor(34)와 함께 UARTputgoto()를 이용하여 철학자N이라면 N+3의 위치((0,0), (0,1)는 출력물이 이미 있음)에 UARTprintf()문을 통해 “philosopher[0~4] is thinking…”을 출력하였다.

이렇게 출력이 끝나면 공유자원인 screen을 모두 사용하였기 때문에 Sem을 post한 뒤, OSTimeDly()을 호출하여 (200 + OS\_GetRandom(50)) tick 시간 동안 생각하도록 했다.

생각이 끝나면, 먼저 왼쪽 포크인 Fork[Left\_fork](=세마포)를 pending하는 과정을 진행한다. 이때, DeadLock 방지를 위한 작업을 추가하였다. Fork[Left\_fork] (=세마포)를 pend하면 세마포 값이 0이 된다. 이때, 만약 Fork[Left\_fork] (=세마포)와 Fork[Right\_fork] (=세마포)의 값이 같다면 왼/오 포크의 세마포의 값이 둘 다 0이므로 DeadLock 현상이 된다. 이럴 경우, 세마포인 Fork[Left\_fork]를 post한 후 OSTimeDly()를 통해 랜덤시간 동안 wait에 갔다가 continue하도록 설정하였다. (비교를 통해 세마포 값이 같지 않다면 다음 과정으로 수행할 수 있도록 설정)

위의 과정이 끝나면 오른쪽 포크인 Fork[Right\_fork]를 pend하는 작업을 진행한다. 현재 왼쪽 포크를 들고 있고 오른쪽 포크도 들 수 있다면 식사를 할 수 있다. 먼저, 공유자원인 screen을 상호배제하기 위해 Sem을 Pend하였으며 식사 중은 붉은색으로 출력해야 하므로 UARTputcolor(31)과 함께 UARTputgoto()를 이용하여 철학자 N이라면 N+3의 위치((0,0), (0,1)는 출력물이 이미 있음)에 UARTprintf()문을 통해 “philosopher[0~4] is eating…”을 출력하였다.

이때, UARTprintf문에서 Left\_fork는 철학자의 번호와 같으므로 몇 번째 철학자가 먹고 있는지를 나타낼 수 있다. 또한 기존에 출력 중이던 “philosopher[0~4] is thinking”을 지우고 그 위치에 “philosopher[0~4] is eating…”이 깨끗하게 출력될 수 있도록 ANSI ESCAPE code를 사용하여 한 줄을 clear하도록 작성하였다.

출력이 모두 끝났으므로 공유자원인 screen을 모두 사용하였기에 Sem을 post했고, 마찬가지로 식사가 모두 끝났으므로 포크를 내려 놓아야 하기 때문에 오/왼 포크를 내려놓기 위해 오른쪽 포크와 왼쪽 포크의 세마포를 post하였다. 이렇게 post된 포크를 다시 for문이 무한 반복되면서 다른 철학자들이 사용할 수 있게 된다.

**3.4**

텍스트이(가) 표시된 사진

자동 생성된 설명

위 수행 화면은 철학자 1과 4가 식사 중이고 남은 철학자 0, 2, 3은 생각하는 중이다.

철학자 1과 4가 양손에 포크를 쥐고 식사를 하고 있을 경우 철학자 0, 2, 3은 식사를 할 수 없기 때문에 프로그램이 올바르게 잘 수행됨을 알 수 있다.