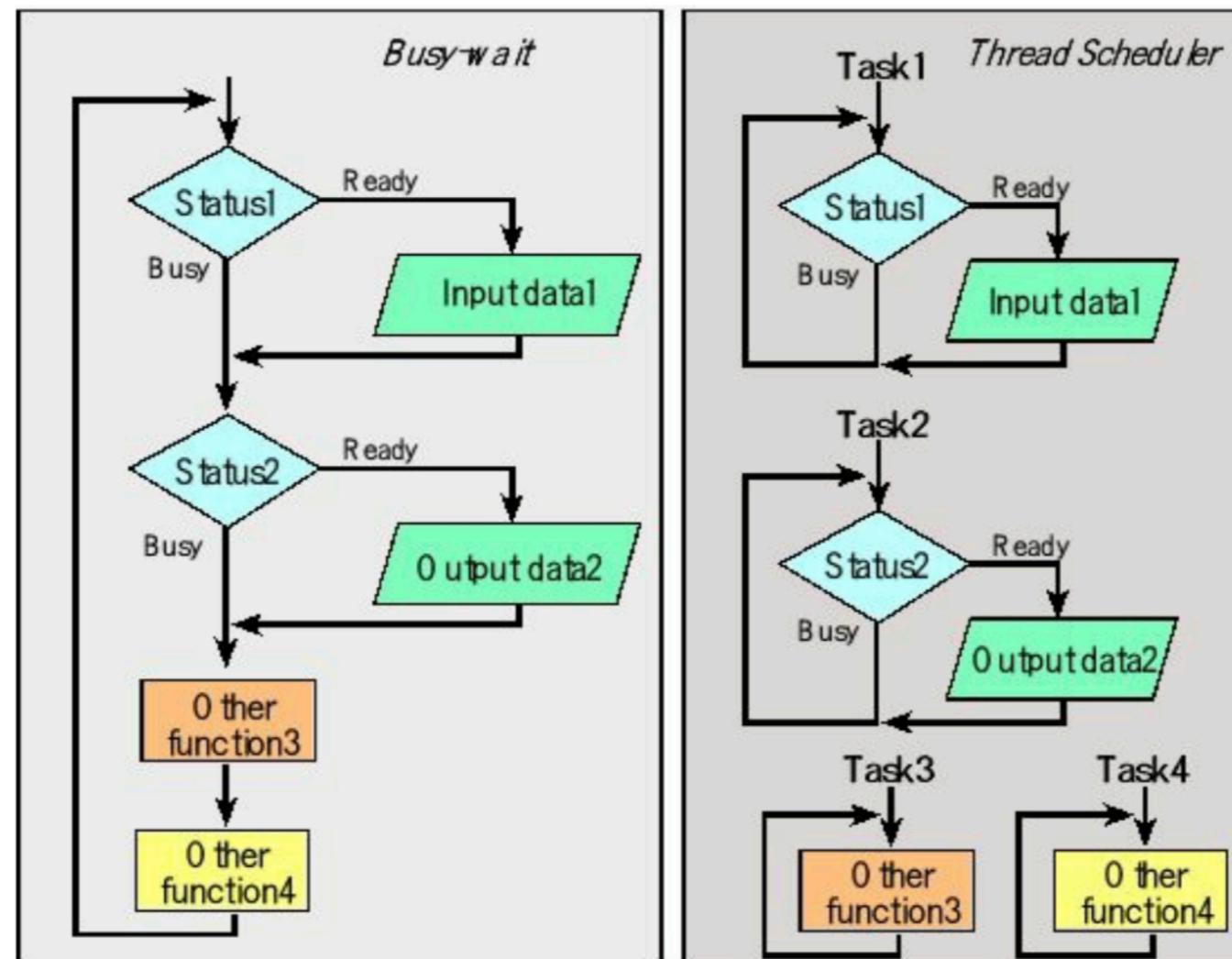


Thread Management

Lesson 07

Thread

- 예를 들어 하나의 입력 task, 하나의 출력 task, 그리고 2개의 non I/O task 가 있다고 가정해보자.

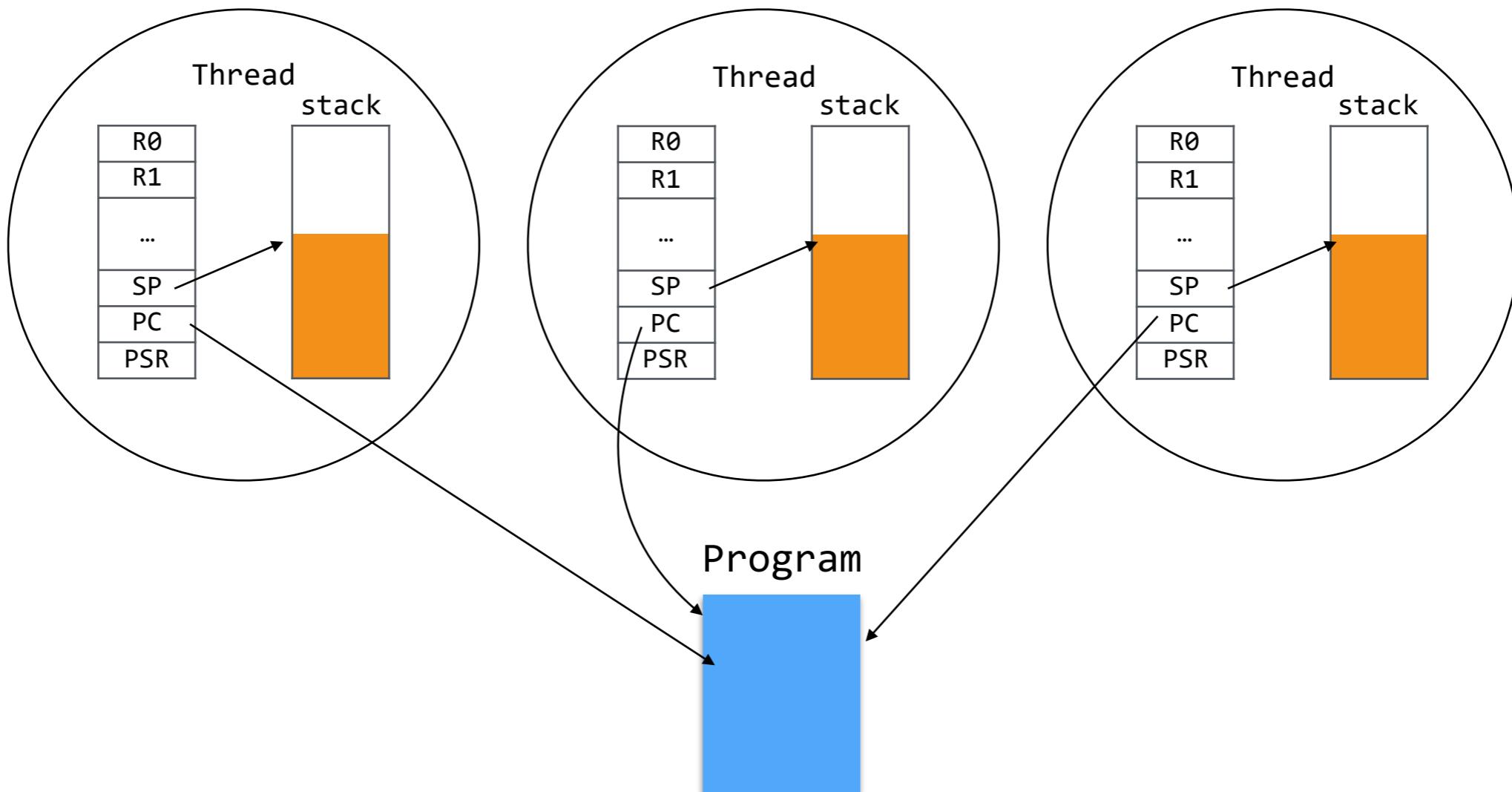


Busy-wait solution

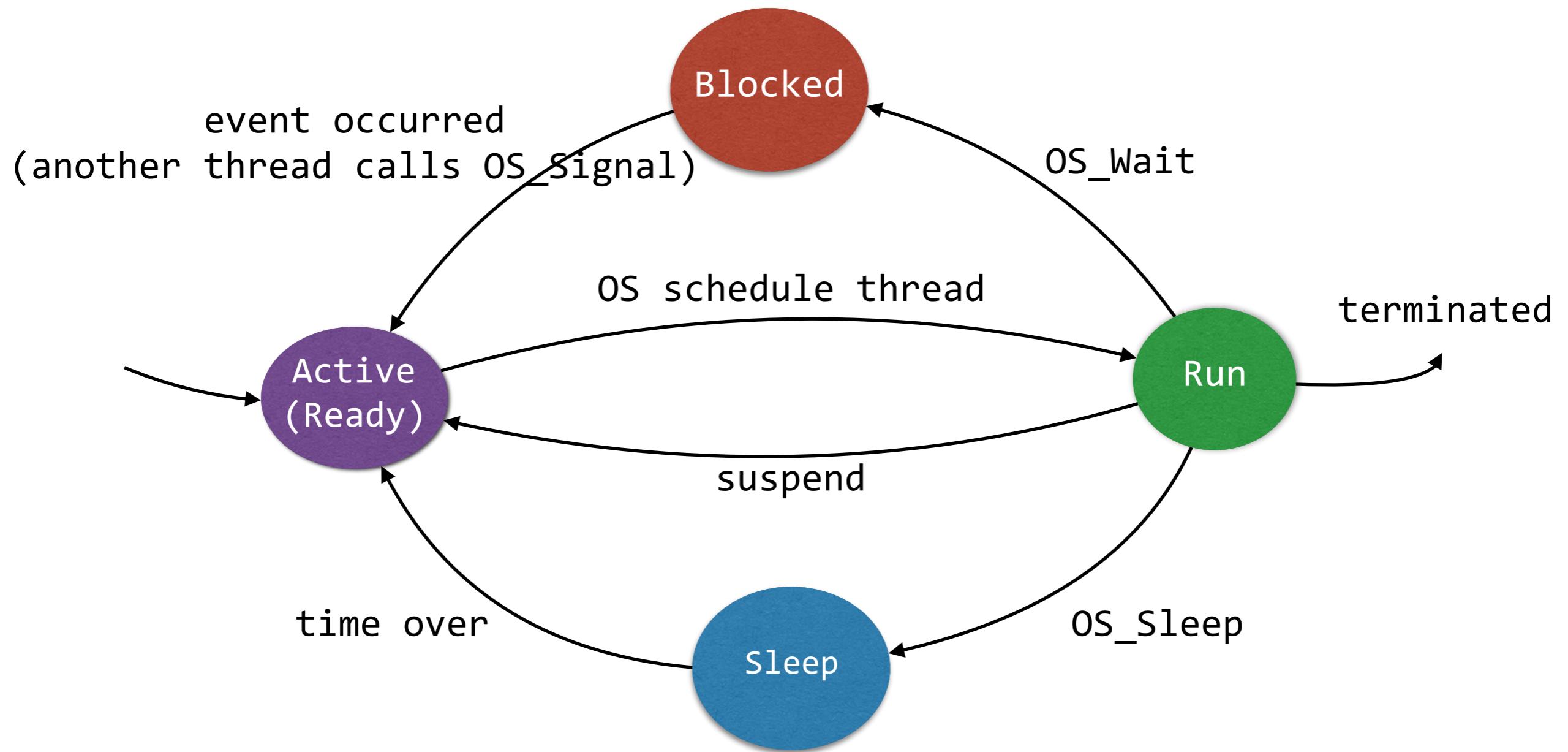
Multithread

Thread

- Thread의 상태는 CPU context (레지스터들의 현재 값)와 지역변수들의 값에 의해 정의된다.
- 이 두가지 정보는 모두 스택에 저장된다.



Thread State



Thread Scheduling

- ⦿ **Round robin scheduler**
- ⦿ **Weighted round robin scheduler**
- ⦿ **Priority scheduler**

An Extremely Simple RTOS

(<http://users.ece.utexas.edu/~valvano/arm/index.htm>)

main.c

```
#include "TM4C123GH6PM.h"
#include <stdint.h>
#include "os.h"

#define TIMESLICE TIME_2MS      // thread switch time in system time units

uint32_t Count1;    // number of times thread1 loops
uint32_t Count2;    // number of times thread2 loops
uint32_t Count3;    // number of times thread3 loops
```

main.c

```
void Task1(void){  
    Count1 = 0;  
    for(;;){  
        Count1++;  
        GPIOD->DATA ^= 0x02;          // toggle PD1  
    }  
}  
  
void Task2(void){  
    Count2 = 0;  
    for(;;){  
        Count2++;  
        GPIOD->DATA ^= 0x04;          // toggle PD2  
    }  
}  
  
void Task3(void){  
    Count3 = 0;  
    for(;;){  
        Count3++;  
        GPIOD->DATA ^= 0x08;          // toggle PD3  
    }  
}
```

```
int main(void){  
  
    OS_Init();          // initialize, disable interrupts, 50 MHz  
  
    SYSCTL->RCGCGPIO |= 0x08;      // activate clock for Port D  
    while((SYSCTL->RCGCGPIO & 0x08) == 0){}; // allow time to stabilize  
  
    GPIOD->DIR |= 0x0E;  
    GPIOD->AFSEL &= ~0x0E;        // disable alt funct on PD3-1  
    GPIOD->DEN |= 0x0E;          // enable digital I/O on PD3-1  
  
    GPIOD->PCTL &= 0xFFFF000F;    // configure PD3-1 as GPIO  
    GPIOD->AMSEL &= ~0x0E;        // disable analog functionality on PD3-1  
  
    OS_AddThreads(&Task1, &Task2, &Task3);  
  
    OS_Launch(TIMESLICE); // doesn't return, interrupts enabled in here  
  
    return 0;                // this never executes  
}
```

```
#ifndef __OS_H
#define __OS_H 1

#define TIME_1MS 50000
#define TIME_2MS 2*TIME_1MS

void OS_Init(void);

int OS_AddThreads(void(*task0)(void),
                  void(*task1)(void),
                  void(*task2)(void));

void OS_Launch(uint32_t theTimeSlice);

#endif
```

```
#include "TM4C123GH6PM.h"
#include <stdint.h>
#include "os.h"

#define NVIC_SYS_PRI3_R          (*((volatile uint32_t *)0xE000ED20))

// function definitions in osasm.s
void DisableInterrupts(void);      // Disable interrupts
void EnableInterrupts(void);       // Enable interrupts
int32_t StartCritical(void);
void EndCritical(int32_t primask);
void StartOS(void);
```

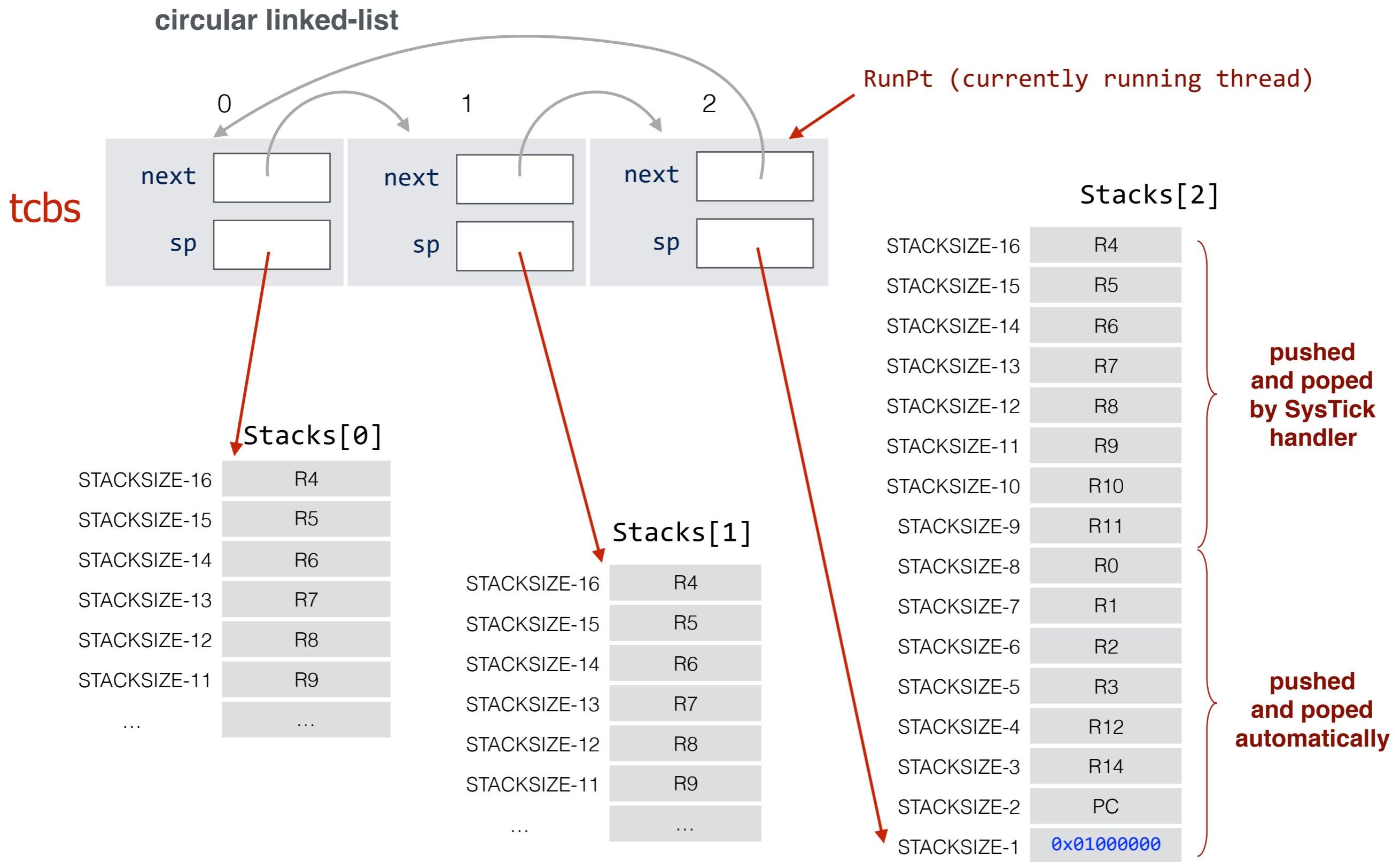
```
#define NUMTHREADS 3           // maximum number of threads
#define STACKSIZE 100          // number of 32-bit words in stack
```

```
struct tcb{
    int32_t *sp;            // pointer to stack
    struct tcb *next;       // link to the tcb of the next thread
};
typedef struct tcb tcbType;
```

```
tcbType tcbs[NUMTHREADS];
tcbType *RunPt;
```

```
int32_t Stacks[NUMTHREADS][STACKSIZE];
```

Thread Control Blocks



```
void OS_Init(void){  
    DisableInterrupts();  
  
    SysTick->CTRL = 0; // disable SysTick during setup  
    SysTick->VAL = 0;  
}
```

```
void SetInitialStack(int i){  
    tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer  
    Stacks[i][STACKSIZE-1] = 0x01000000; // thumb bit, PSR  
    Stacks[i][STACKSIZE-3] = 0x14141414; // R14  
    Stacks[i][STACKSIZE-4] = 0x12121212; // R12  
    Stacks[i][STACKSIZE-5] = 0x03030303; // R3  
    Stacks[i][STACKSIZE-6] = 0x02020202; // R2  
    Stacks[i][STACKSIZE-7] = 0x01010101; // R1  
    Stacks[i][STACKSIZE-8] = 0x00000000; // R0  
    Stacks[i][STACKSIZE-9] = 0x11111111; // R11  
    Stacks[i][STACKSIZE-10] = 0x10101010; // R10  
    Stacks[i][STACKSIZE-11] = 0x09090909; // R9  
    Stacks[i][STACKSIZE-12] = 0x08080808; // R8  
    Stacks[i][STACKSIZE-13] = 0x07070707; // R7  
    Stacks[i][STACKSIZE-14] = 0x06060606; // R6  
    Stacks[i][STACKSIZE-15] = 0x05050505; // R5  
    Stacks[i][STACKSIZE-16] = 0x04040404; // R4  
}
```

```
int OS_AddThreads(void(*task0)(void),
                  void(*task1)(void),
                  void(*task2)(void)){
    int32_t status;
    status = StartCritical();
    tcbs[0].next = &tcbs[1]; // 0 points to 1
    tcbs[1].next = &tcbs[2]; // 1 points to 2
    tcbs[2].next = &tcbs[0]; // 2 points to 0

    SetInitialStack(0);
    Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
    SetInitialStack(1);
    Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
    SetInitialStack(2);
    Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC

    RunPt = &tcbs[0];           // thread 0 will run first
    EndCritical(status);
    return 1;
}
```

```
void OS_Launch(uint32_t theTimeSlice){  
    SysTick->LOAD = theTimeSlice - 1; // reload value  
    SysTick->CTRL = 3; // enable, core clock and interrupt arm  
  
    StartOS(); // start on the first task  
}
```

osasm.s



download
this file

```
EXTERN RunPt           ; currently running thread
EXPORT DisableInterrupts
EXPORT EnableInterrupts
EXPORT StartOS
EXPORT SysTick_Handler
EXPORT StartCritical
EXPORT EndCritical
```

```
SECTION .text : CODE (2)
```

DisableInterrupts

```
CPSID I           ; __disable_irq();
BX    LR          ; returns
```

EnableInterrupts

```
CPSIE I           ; __enable_irq();
BX    LR
```

SysTick_Handler

```

CPSID    I          ; 1) Saves R0-R3,R12,LR,PC,PSR
PUSH    {R4-R11}     ; 2) Prevent interrupt during switch
LDR     R0, =RunPt   ; 3) Save remaining regs r4-11
LDR     R1, [R0]      ; 4) R0=pointer to RunPt, old thread
                      ;     R1 = *RunPt i.e., R1 = &(RunPt->sp);

STR     SP, [R1]      ; 5) Save SP into TCB; SP = RunPt->sp;

LDR     R1, [R1,#4]    ; 6) R1 = RunPt->next
STR     R1, [R0]      ;     RunPt = R1

LDR     SP, [R1]      ; 7) new thread SP; SP = RunPt->sp;

POP    {R4-R11}     ; 8) restore regs r4-11
CPSIE   I          ; 9) tasks run with interrupts enabled
BX      LR         ; 10) restore R0-R3,R12,LR,PC,PSR

```

address of RunPt

StartOS

```
LDR    R0, =RunPt      ; R0 = &RunPt;
LDR    R2, [R0]        ; R2 = *RunPt, i.e., R2 = &(RunPt->sp);
```

```
LDR    SP, [R2]        ; first thread SP; SP = RunPt->sp;
```

```
POP   {R4-R11}        ; restore regs r4-11
POP   {R0-R3}          ; restore regs r0-3
POP   {R12}
POP   {LR}             ; discard LR from initial stack
```

```
stacksize-2 POP   {LR}          ; start location
```

function pointer
of first thread
(return address로 사용됨)

```
stacksize-1 POP   {R1}          ; discard PSR
```

```
CPSIE I               ; Enable interrupts at processor level
BX    LR              ; start first thread
```

StartCritical

```
MRS    R0, PRIMASK ; save old status  
CPSID I           ; mask all (except faults)  
BX    LR
```

EndCritical

```
MSR    PRIMASK, R0  
BX    LR
```

```
END
```

MRS:

Move the contents of a special register to a general-purpose register.

MSR:

Move the contents of a general-purpose register into the specified special register.

PRIMASK:

The PRIMASK register prevents activation of all exceptions with configurable priority.

Semaphore

Semaphore

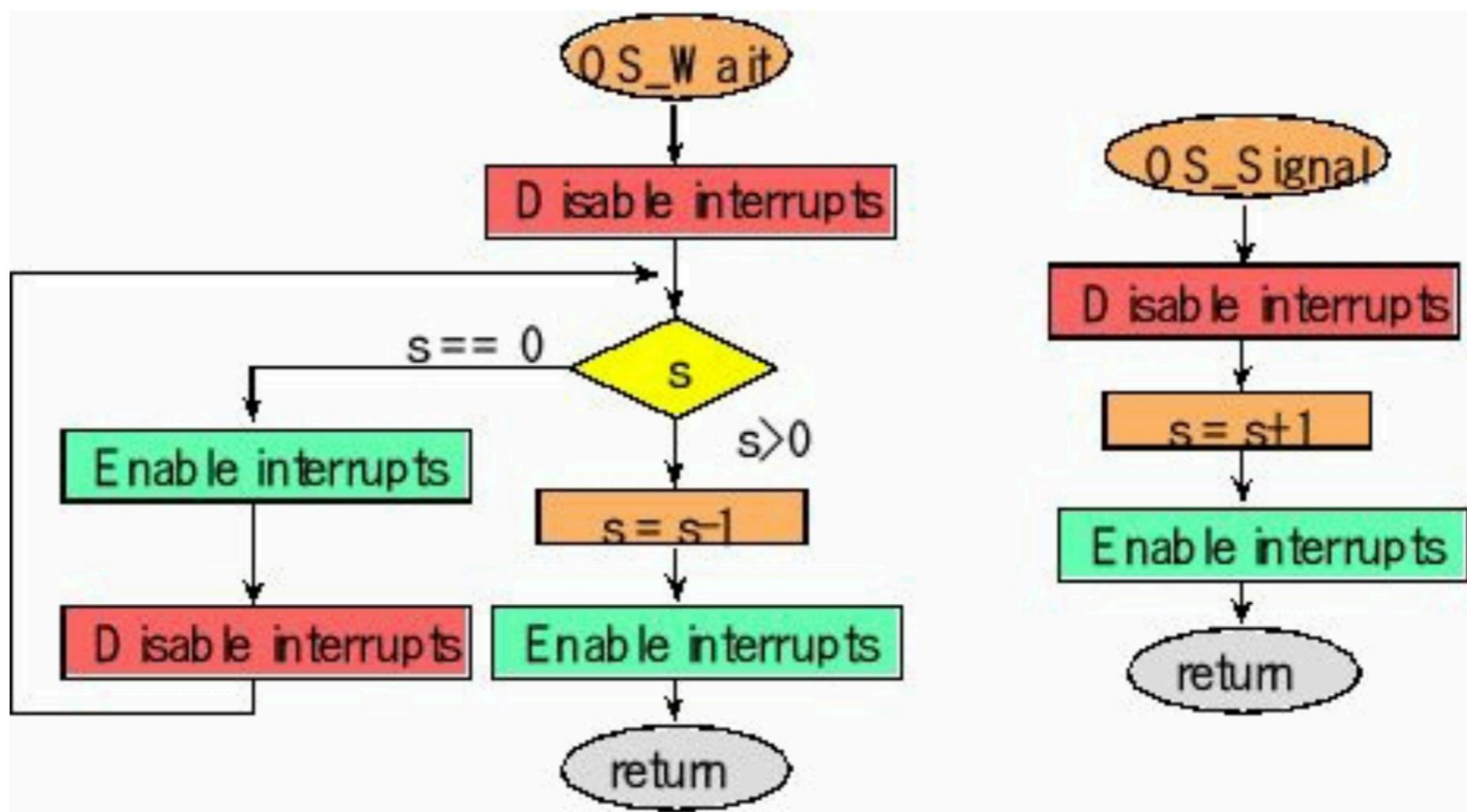
- ☞ Thread들이 서로 상호작용하기 위한 수단이 필요하다.
 - ☞ 쓰레드 동기화(synchronization)
 - ☞ 자원 공유
 - ☞ 통신(communication)
- ☞ semaphore는 thread간의 상호작용을 위한 기본 수단이다.

Semaphore 다음의 세 가지 함수를 제공하는 카운터

- ⦿ **OS_InitSemaphore**: 초기화, 처음에 한 번 실행
- ⦿ **OS_Wait (pend 혹은 P라고도 부름)**
- ⦿ **OS_Signal (post 혹은 V라고도 부름)**
- ⦿ 각각은 **atomic** 연산이어야 함

Spin-Lock Semaphore

• 가장 단순한 semaphore 구현



Spinlock semaphore

빙빙 돈다

```
void OS_Wait(int32_t *s){  
    DisableInterrupts();  
    while(*s == 0){ s>0를때까지 돈다  
        EnableInterrupts(); // <- interrupts can occur here  
        DisableInterrupts();  
    }  
    *s = *s - 1;  
    EnableInterrupts();  
}
```

일종의 busy waiting. 비효율적

```
void OS_Signal(int32_t *s){  
    DisableInterrupts();  
    *s = *s + 1;  
    EnableInterrupts();  
}
```

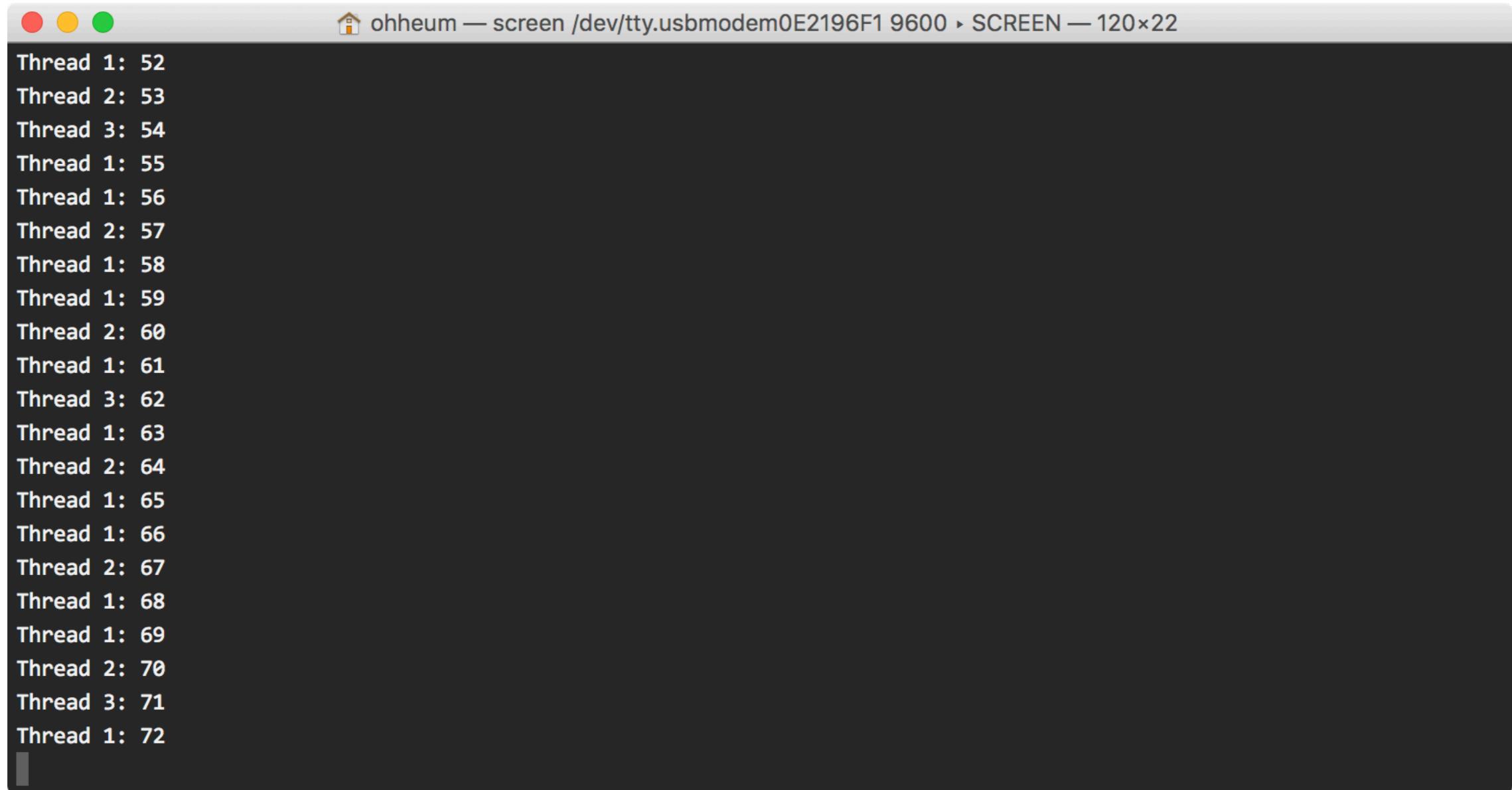
semaphore는 처음에 1로 초기화

Mutex

- ❸ semaphore를 이용하여 하나의 자원을 mutually exclusive하게 공유하기

```
void Thread1(void){  
    Init1();  
    while (1){  
        OS_Wait(&x);  
        // exclusive access  
        OS_Signal(&x);  
        // other processing  
    }  
}
```

```
void Thread2(void){  
    Init2();  
    while (1){  
        OS_Wait(&x);  
        // exclusive access  
        OS_Signal(&x);  
        // other processing  
    }  
}
```



```
ohheum — screen /dev/tty.usbmodem0E2196F1 9600 ▶ SCREEN — 120x22
```

```
Thread 1: 52
Thread 2: 53
Thread 3: 54
Thread 1: 55
Thread 1: 56
Thread 2: 57
Thread 1: 58
Thread 1: 59
Thread 2: 60
Thread 1: 61
Thread 3: 62
Thread 1: 63
Thread 2: 64
Thread 1: 65
Thread 1: 66
Thread 2: 67
Thread 1: 68
Thread 1: 69
Thread 2: 70
Thread 3: 71
Thread 1: 72
```

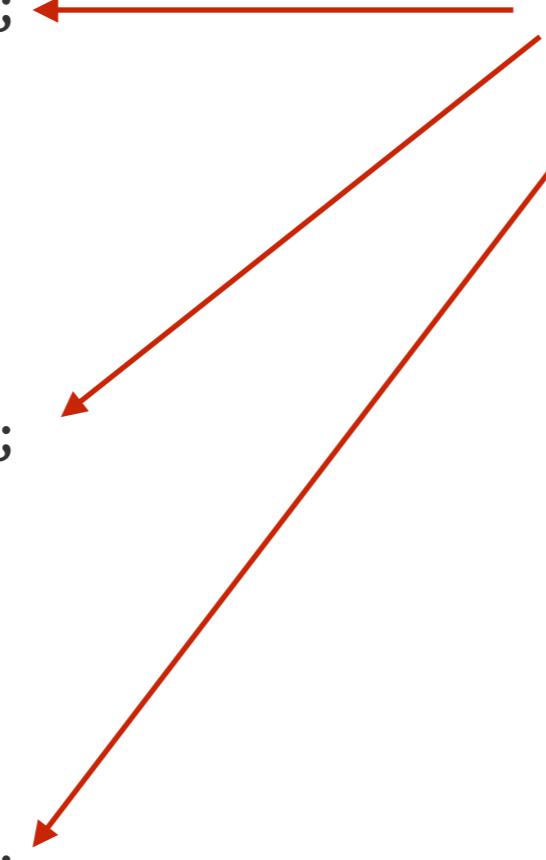
이런 식으로 3개의 thread가 **함께** counting하도록 만들어보자. **race condition**이 발생하지 않아야 한다.

```
void Task1(void){  
    while(1) {  
        counting_op(1);  
        doSomethingElse(10);  
    }  
}
```

```
void Task2(void){  
    while(1) {  
        counting_op(2);  
        doSomethingElse(20);  
    }  
}
```

```
void Task3(void){  
    while(1) {  
        counting_op(3);  
        doSomethingElse(30);  
    }  
}
```

각 thread마다 다른 일을 하는데
걸리는 시간은 다르다.



지난 시간의 code에서 task들을
이렇게 수정한다.

```

int32_t count = 0;           ← 3개의 thread가 공유하는 counter
int32_t mutex = 1;           ← semaphore variable, 1로 초기화
char buffer[20];

void counting_op(int thread_id) {
    OS_Wait(&mutex);      ← 이 2라인을 지우고 실행해보자.

    count++;
    UART0TxString("Thread ");
    itoa(thread_id, buffer);
    UART0TxString(buffer);
    UART0TxString(": ");
    itoa(count, buffer);
    UART0TxString(buffer);
    UART0TxString("\n\r");
}

OS_Signal(&mutex);

```

critical section
 (오직 하나의 thread 만이
 이 부분을 실행중일 수 있다).

UART0TxString은
 serial.c 파일에 구현되어 있다.
 main 함수의 앞부분에서
 initUART0()를 호출해준다.

```
char* itoa(int32_t i, char b[]){
    char const digit[] = "0123456789";
    char* p = b;
    if(i<0){
        *p++ = '-';
        i *= -1;
    }
    int shifter = i;
    do{ //Move to where representation ends
        ++p;
        shifter = shifter/10;
    } while(shifter);
    *p = '\0';
    do{ //Move back, inserting digits as u go
        *--p = digit[i%10];
        i = i/10;
    } while(i);
    return b;
}
```

int를 char []로 변환하는 함수

```
/* delay n milliseconds (16 MHz CPU clock) */
void doSomethingElse(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}
```

Blocking semaphores

Suspend

- spinLock 세마포어는 세마포어가 release될때까지 busy waiting한다. 이 것은 비효율적이다.
- OS 혹은 실행중인 thread 자신이 (어떤 특정한 조건이 만족될 때까지는) 더 이상 의미있는 진도를 나갈 수 없다고 판단할 수 있다.
- 이때 실행 중인 thread를 중단하고 다른 thread를 실행하기 위해서 OS_Suspend를 호출한다.
실행 중인 쓰레드를 블락시키고 새로운 쓰레드를 실행한다.
- Interrupt Control State 레지스터의 26번 비트에 1을 쓰면 SysTick 인터럽트가 발생된다.
 - CMSIS/Include/core_cm4.h 파일의 struct SCB_Type 참조
 - SCB->ICSR = 0x04000000

OS_Suspend

```
void OS_Suspend(void){  
    SysTick->VAL = 0;           // reset counter  
  
    SCB->ICSR = 0x04000000;    // 강제로 SysTick 인터럽트를 발생시킨다.  
    }                           // 26번째 값
```

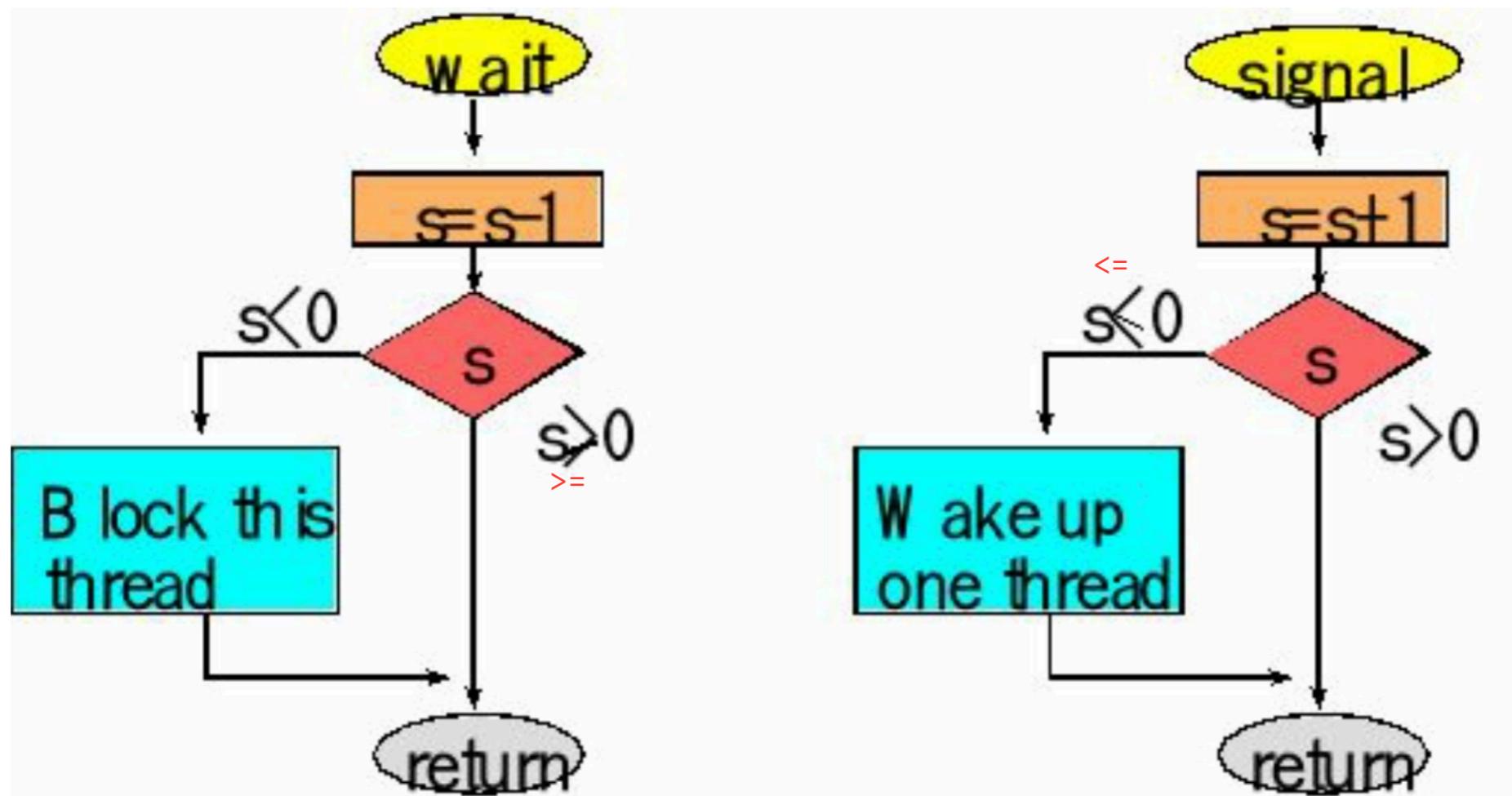
SysTick 카운터를 0으로 reset하지 않으면
다음에 실행될 thread는 더 짧은 기간 동안만 실행된다.

```
typedef struct  
{  
    __IOM uint32_t CTRL;        /* SysTick Control and Status Register */  
    __IOM uint32_t LOAD;        /* SysTick Reload Value Register */  
    __IOM uint32_t VAL;         /* SysTick Current Value Register */  
    __IM  uint32_t CALIB;       /* SysTick Calibration Register */  
} SysTick_Type;  
  
#define SysTick ((SysTick_Type *) SysTick_BASE )
```

core_cm4.h파일에서

Blocking Counting Semaphore

s는 처음에 1로 초기화한다.



s가 음수이면 |s|개의 thread가 wait 상태임을 표현한다.
개수

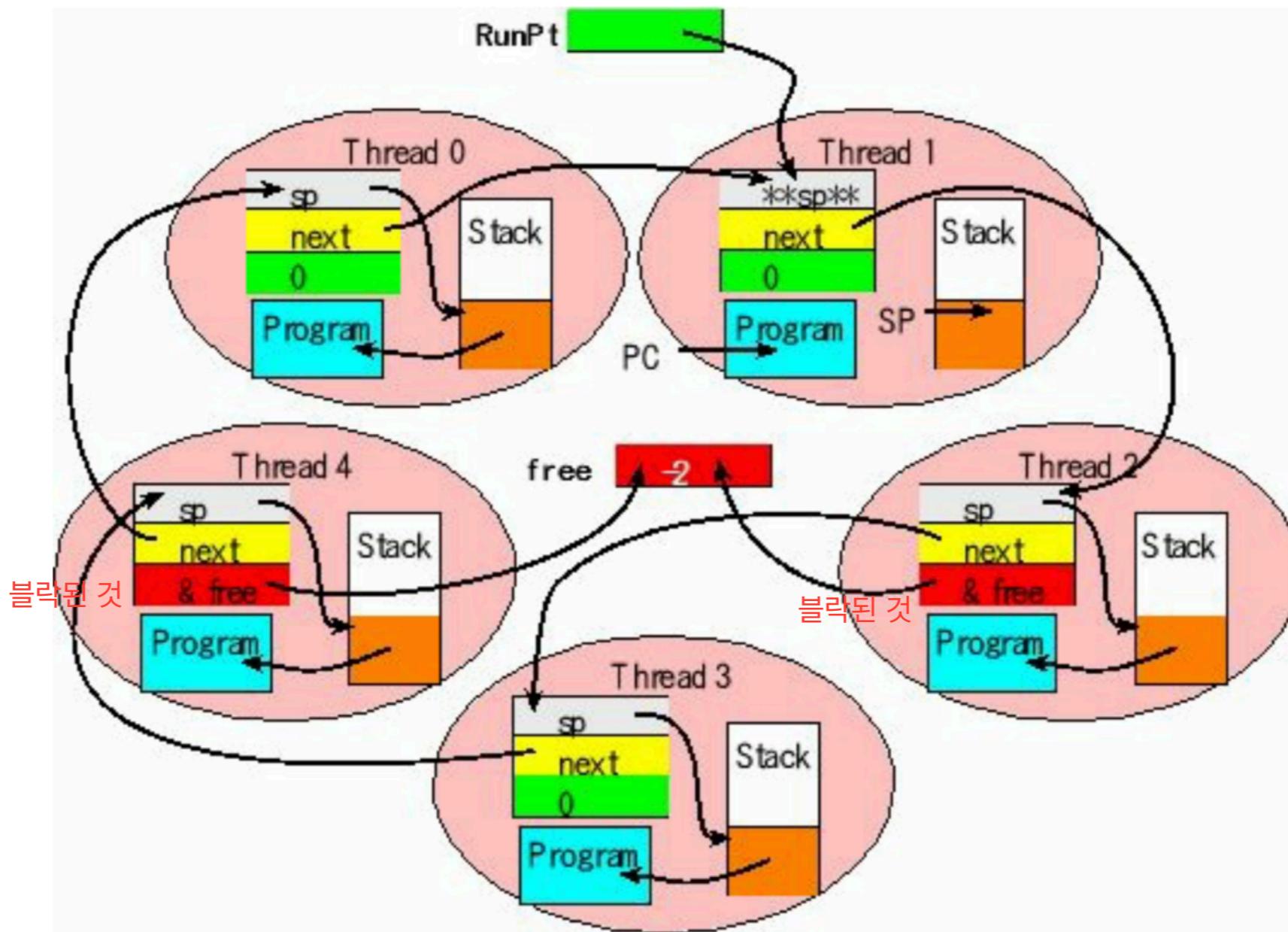
block된 상태의 thread를 관리하기 위해서
TCB의 구조를 수정한다.

block wait을
구분해야한다.

```
struct tcb{  
    int32_t *sp;          // pointer to stack (valid for threads not running  
    struct tcb *next;    // linked-list pointer  
  
    int32_t *blocked;    ← 이 thread가 block될 때 waiting하고 있는 semaphore  
                        변수의 주소를 저장한다.  
};  
typedef struct tcb tcbType;
```

c에선 순서가 상관없지만 어셈블리어에서는 순서가 중요하다

Blocking Counting Semaphore



block 상태의 thread들 (이 그림에서는 Thread 2, 4)은 TCB의 blocked 필드가
자신이 waiting하고 있는 semaphore의 주소를 저장한다.

OS_Wait

```
void OS_Wait(int32_t *s){  
    DisableInterrupts();  
    *s = *s - 1;  
    if(*s < 0){  
        실행중인 TCB RunPt->blocked = s;           // reason it is blocked  
        EnableInterrupts();  
        OS_Suspend();                                // run thread switcher 블락 상태로 들어감  
    }  
    EnableInterrupts();  
}
```

OS_Signal

```
void OS_Signal(int32_t *s){
    tcbType *pt;
    DisableInterrupts();
    *s = *s + 1;
    if((*s) <= 0){
        pt=RunPt->next;      //search for one blocked on this
        while(pt->blocked != s){
            pt = pt->next;      나때문에 잠 든 애를 찾는다.
        }
        pt->blocked = 0;      // wakeup this one
                               깨워주고 실행 시키지는 않는다.
    }
    EnableInterrupts();
}
```

scheduler: os.c

```
void Scheduler(void) {  
    RunPt = RunPt->next;  
    while(RunPt->blocked)  
        RunPt = RunPt->next;  
}
```

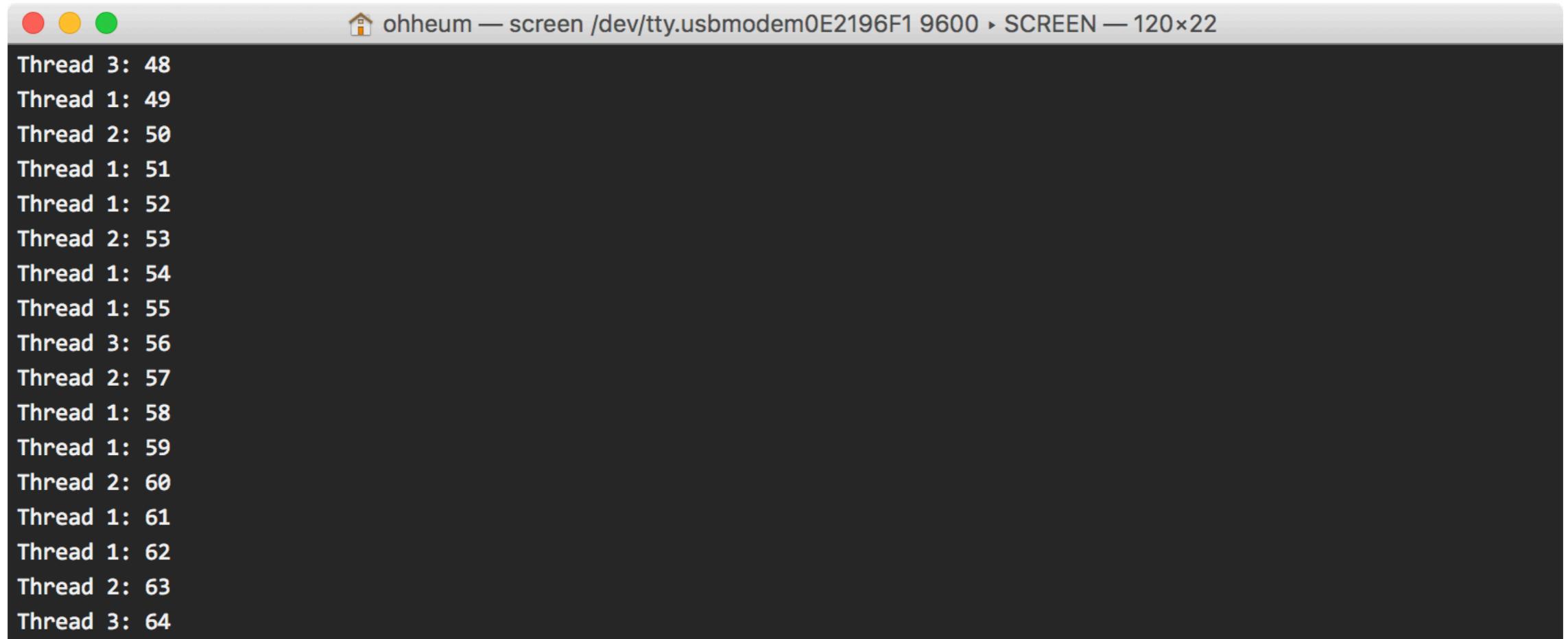
Scheduler는 단순 round robin이 아닌 block되지 않은 첫 번째 thread를 찾아서 dispatch한다.

SysTickHandler

```
FUNCTION      SysTick_Handler    // 1) Saves R0-R3,R12,LR,PC,PSR
CPSID        I                  // 2) Prevent interrupt during switch
PUSH         {R4-R11}           // 3) Save remaining regs r4-11
LDR          R0, =RunPt         // 4) R0=pointer to RunPt, old thread
LDR          R1, [R0]           //      R1 = RunPt
STR          SP, [R1]           // 5) Save SP into TCB
PUSH         {R0, LR}
BL          Scheduler
POP          {R0, LR}
LDR          R1, [R0]
LDR          SP, [R1]           // 7) new thread SP; SP = RunPt->sp;
POP          {R4-R11}           // 8) restore regs r4-11
CPSIE       I                  // 9) tasks run with interrupts enabled
BX          LR                 // 10) restore R0-R3,R12,LR,PC,PSR
ENDFUNC SysTick_Handler
```

←———— os.c의 Scheduler 함수를 호출하도록 수정한다.

실행 예



The screenshot shows a terminal window titled "ohheum — screen /dev/tty.usbmodem0E2196F1 9600 ▶ SCREEN — 120×22". The window contains the following text output:

```
Thread 3: 48
Thread 1: 49
Thread 2: 50
Thread 1: 51
Thread 1: 52
Thread 2: 53
Thread 1: 54
Thread 1: 55
Thread 3: 56
Thread 2: 57
Thread 1: 58
Thread 1: 59
Thread 2: 60
Thread 1: 61
Thread 1: 62
Thread 2: 63
Thread 3: 64
```

눈에 보이는 결과에는 별 차이가 없다.

Thread rendezvous

주로 베리어, 조인을 쓴다.

```
void Task1(void){ // Thread 1
    Init1();
    while (1){
        Unrelated1();

        OS_Signal(&S1);
        OS_Wait(&S2);  ← 둘다 0으로 초기화 시켜줌
    }
}
```

```
void Task2(void){ // Thread2
    Init2();
    while (1){
        Unrelated2();

        OS_Signal(&S2);
        OS_Wait(&S1);  ← 동기화를 시켜준다.
    }
}
```

두개 이상의 프로세스 혹은 thread가 코드의 어떤 부분에서 서로 동기화하는 것

Thread rendezvous

```
int32_t rend1 = 0;
int32_t rend2 = 0;
int32_t rend3 = 0;

void Task1(void){
    while(1) {
        counting_op(1);
        doSomethingElse(1);
        OS_Signal(&rend2);
        OS_Wait(&rend1);
    }
}

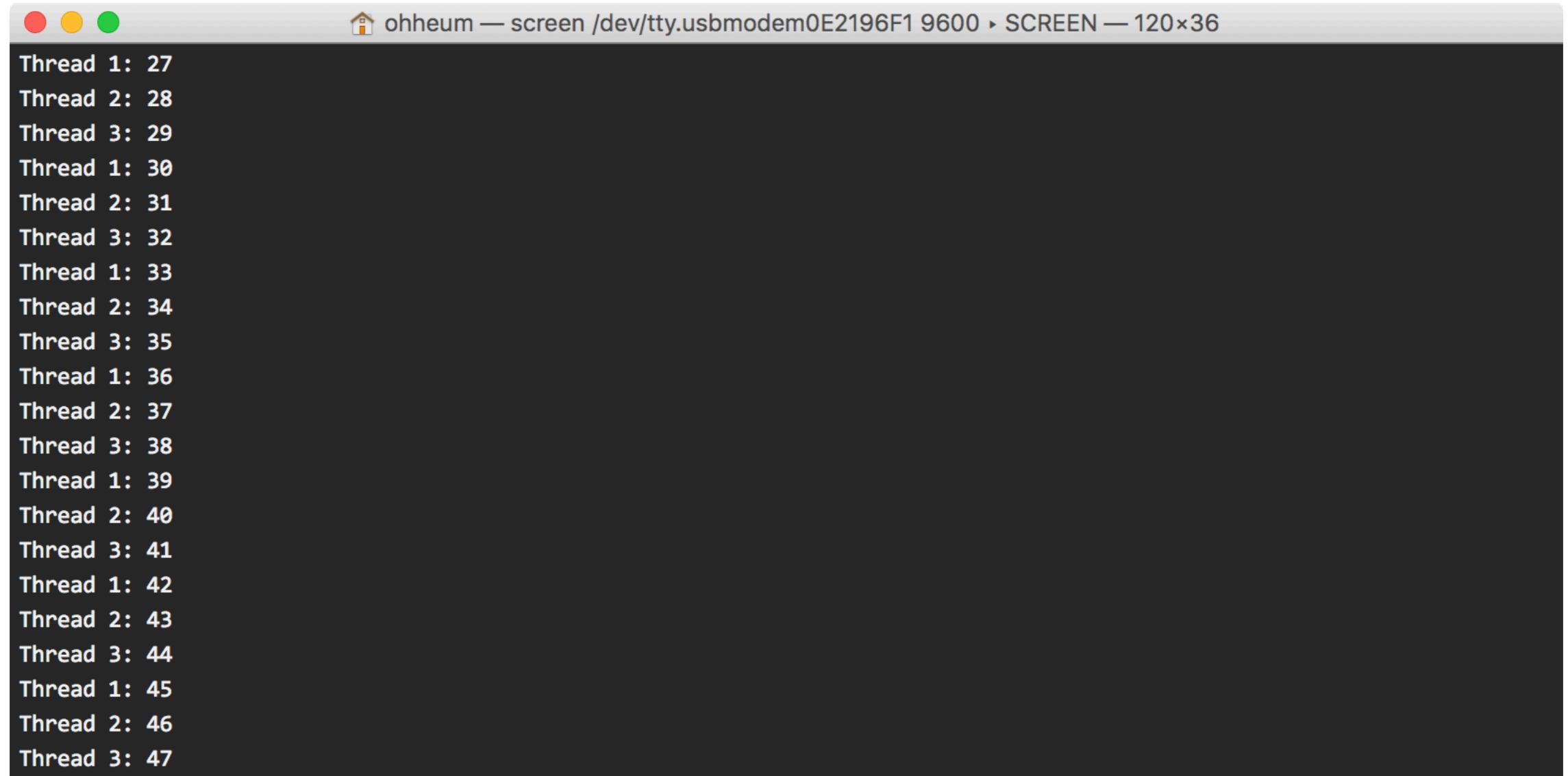
void Task2(void){
    while(1) {
        counting_op(2);
        doSomethingElse(10);
        OS_Signal(&rend3);
        OS_Wait(&rend2);
    }
}

void Task3(void){
    while(1) {
        counting_op(3);
        doSomethingElse(50);
        OS_Signal(&rend1);
        OS_Wait(&rend3);
    }
}
```

동기화는 되지 않아도 돌아가면서 사용한다.

실수 예제

실행 예



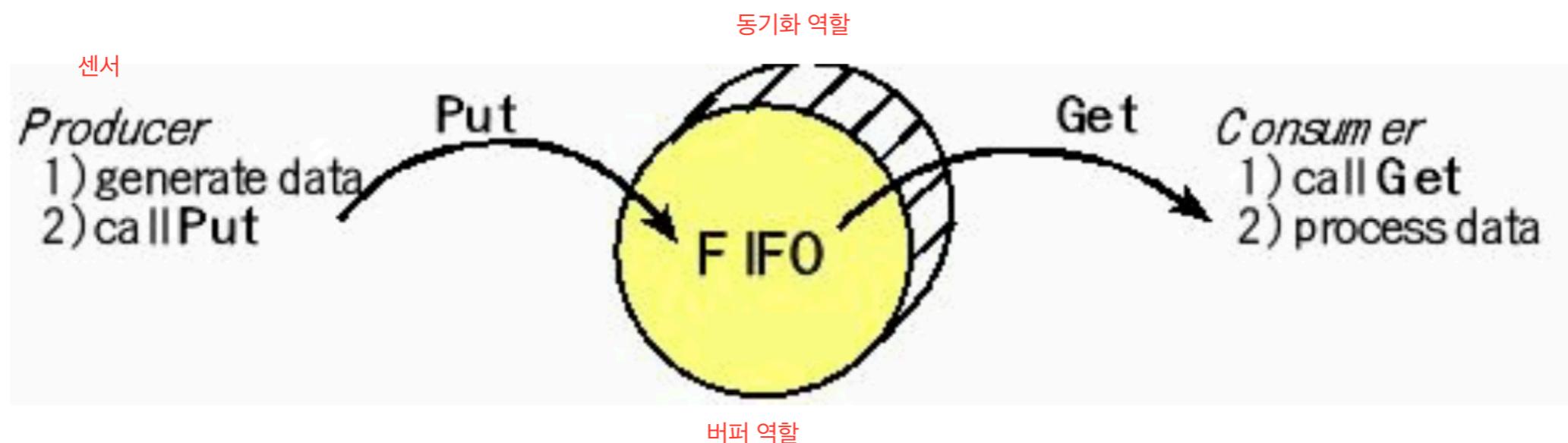
```
ohheum — screen /dev/tty.usbmodem0E2196F1 9600 ▶ SCREEN — 120×36
```

```
Thread 1: 27
Thread 2: 28
Thread 3: 29
Thread 1: 30
Thread 2: 31
Thread 3: 32
Thread 1: 33
Thread 2: 34
Thread 3: 35
Thread 1: 36
Thread 2: 37
Thread 3: 38
Thread 1: 39
Thread 2: 40
Thread 3: 41
Thread 1: 42
Thread 2: 43
Thread 3: 44
Thread 1: 45
Thread 2: 46
Thread 3: 47
```

Thread들이 서로 돌아가면서 1번씩 counting한다.

FIFO

Semaphore를 이용한 FIFO 구현



empty 시에는 consumer는 블락
full 시에는 문제가 된다. -> 새로운 데이터를 넣어버림, or 오래된 데이터에 덮어씌움

- 2개의 **thread**가 **counting_op**를 수행한 후 **count** 값을 자신의 **thread_id**와 함께 **queue**에 저장하면 3번째 **thread**가 **queue**에서 꺼내 노트북으로 전송하도록 프로그램을 수정해보자.

Semaphore를 이용한 FIFO 구현(queue.c)

```
#include "queue.h"

#define CAPACITY 10

uint32_t rear;
uint32_t front;

Item Fifo[CAPACITY];
```

사이즈를 보호해줌 ->> 사이즈를 세마포어로 쓰기위해

```
int32_t size;           // 0 means FIFO empty, FIFOSIZE means full
uint32_t lostData;      // number of lost pieces of data
```

```
void OS_FIFO_Init(void){
    rear = front = 0; // Empty
    OS_InitSemaphore(&size, 0); // CurrentSize = 0
    lostData = 0;
}
```

```
/* os.c와 os.h에 이 함수를 추가한다. */
void OS_InitSemaphore(int32_t *s, int32_t initVal) {
    *s = initVal;
}
```

Semaphore를 이용한 FIFO 구현(queue.c)

```
int OS_FIFO_Put(Item data) {
    if(size == CAPACITY) {
        lostData++;
        return -1; //full
    }
    else {
        Fifo[rear] = data; // Put
        rear = (rear+1)%CAPACITY;
        OS_Signal(&size );
        return 0; // success
    }
}

Item OS_FIFO_Get(void) {
    Item data;
    OS_Wait(&size);
    data = Fifo[front];
    front=(front+1)%CAPACITY; //place to get next
    return data;
}
```

새로운 데이터를 잃어버릴거다.. 그대신 개수를 센다.

큐에다가 넣을려고 하는데 다 차있으면 블락시키는 방법 --> 임베디드 시스템에서는 주기적으로 값을 측정하기 때문에 블록시키는 것은 원래의 의도를 잃어버릴 수 있다.
차라리 오래된 데이터에 덮어씌우는게 낫다. 특히 들어오는 값이 인터럽트 핸들러일때 문제는 더 커질 수 있다.

Semaphore를 이용한 FIFO 구현(queue.h)

```
#ifndef QUEUE_H_
#define QUEUE_H_

#include <stdint.h>
#include "os.h"

typedef struct item {
    int32_t thread_id;
    uint32_t count;
} Item;

void OS_FIFO_Init(void);
int OS_FIFO_Put(Item data);
Item OS_FIFO_Get(void);

#endif /* QUEUE_H_ */
```

main.c에서 counting_op함수의 수정

```
void counting_op(int32_t thread_id) {  
    OS_Wait(&mutex);  
  
    count++;  
  
    Item t;  
    t.thread_id = thread_id;  
    t.count = count;  
  
    OS_FIFO_Put(t);  
  
    OS_Signal(&mutex);  
}
```

main.c에서 함수 추가

3번 쓰레드만

```
void transmitMessage()
{
    Item t = OS_FIFO_Get();

    UART0TxString("Thread ");
    itoa(t.thread_id, buffer);
    UART0TxString(buffer);
    UART0TxString(": ");
    itoa(t.count, buffer);
    UART0TxString(buffer);
    UART0TxString("\n\r");
}
```

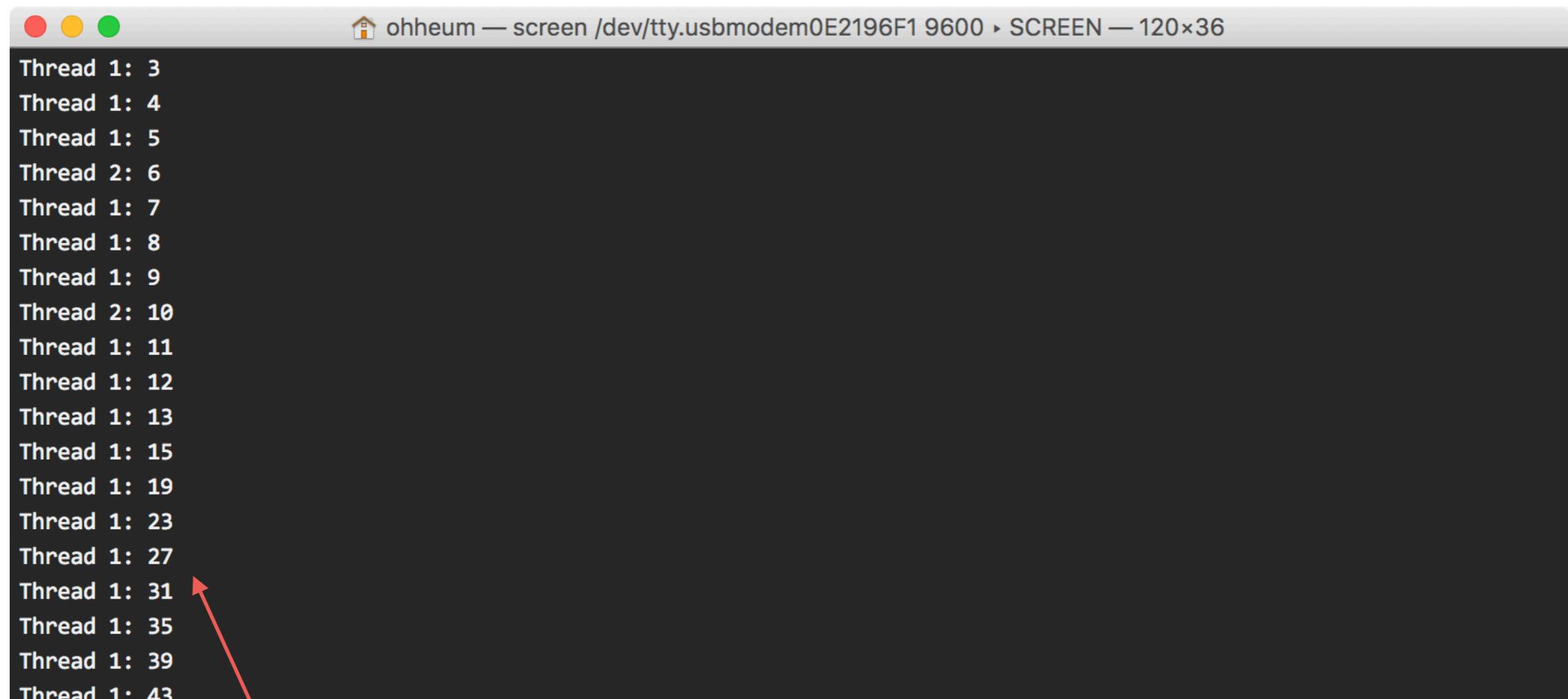
main.c에서 task 함수

```
void Task1(void){ producer
    while(1) {
        counting_op(1);
        doSomethingElse(10);
    }
}

void Task2(void){ producer
    while(1) {
        counting_op(2);
        doSomethingElse(30);
    }
}

void Task3(void){ consumer
    while(1) {
        transmitMessage();
        doSomethingElse(30);
    }
}
```

실행 결과



```
ohheum — screen /dev/tty.usbmodem0E2196F1 9600 ▶ SCREEN — 120×36
```

```
Thread 1: 3
Thread 1: 4
Thread 1: 5
Thread 2: 6
Thread 1: 7
Thread 1: 8
Thread 1: 9
Thread 2: 10
Thread 1: 11
Thread 1: 12
Thread 1: 13
Thread 1: 15
Thread 1: 19
Thread 1: 23
Thread 1: 27
Thread 1: 31
Thread 1: 35
Thread 1: 39
Thread 1: 43
```

A red arrow points from the text "lost data들이 발생한다." to the number "31" in the terminal output.

lost data들이 발생한다.
이것에 대한 대책은 상황마다 다를 수 있다.

인터럽트를 받는다는 것은 기본적으로 쓰레드가 두개라는 이야기다.
main문과 인터럽트

- 6장에서 다른 **UART0**과 **UART5**를 사용하여 노트북과 통신하기을
semaphore를 이용한 **FIFO**를 사용하도록 수정하라 (단, **spin lock**
semaphore를 사용하라.)