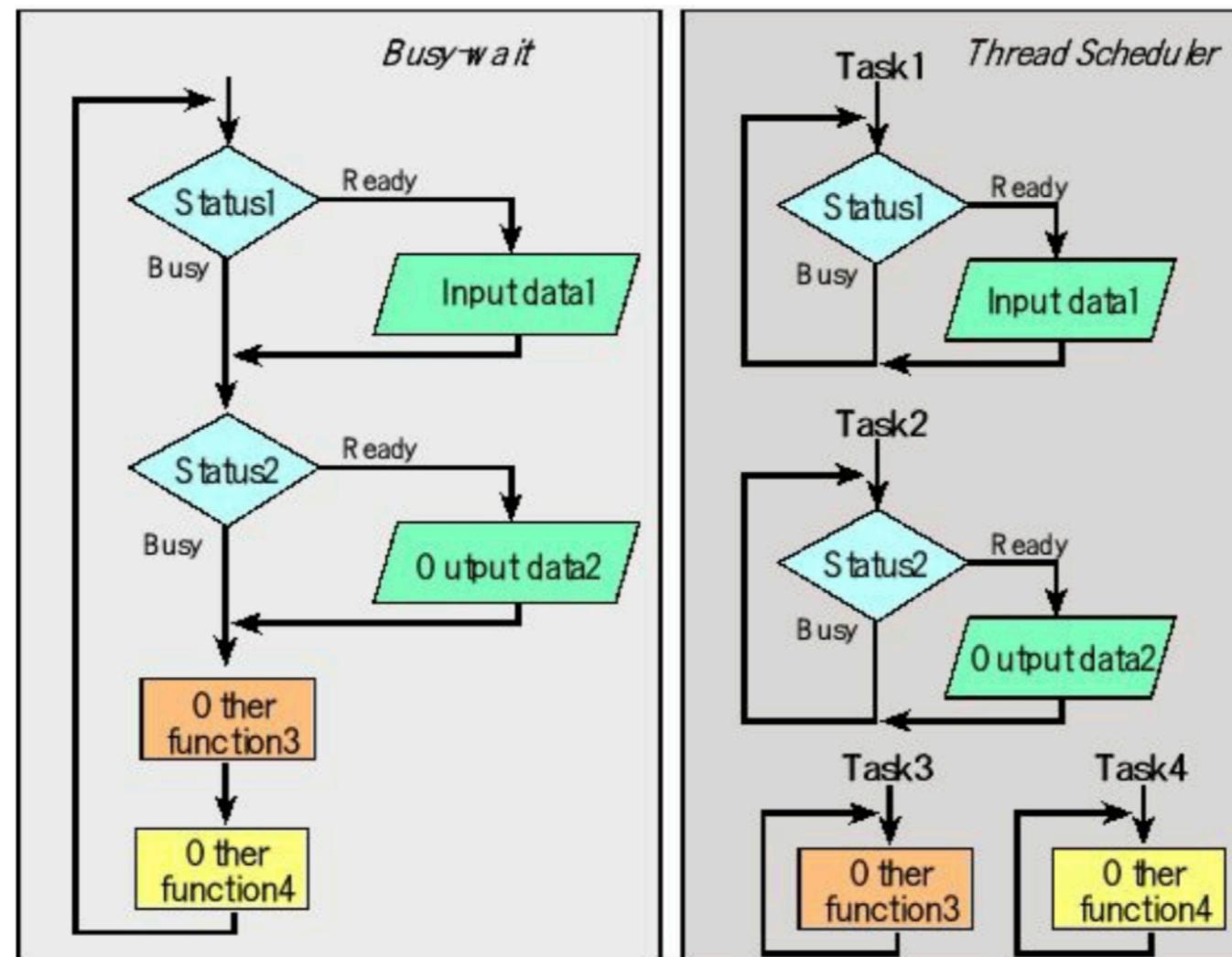


Thread Management

Lesson 07

Thread

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

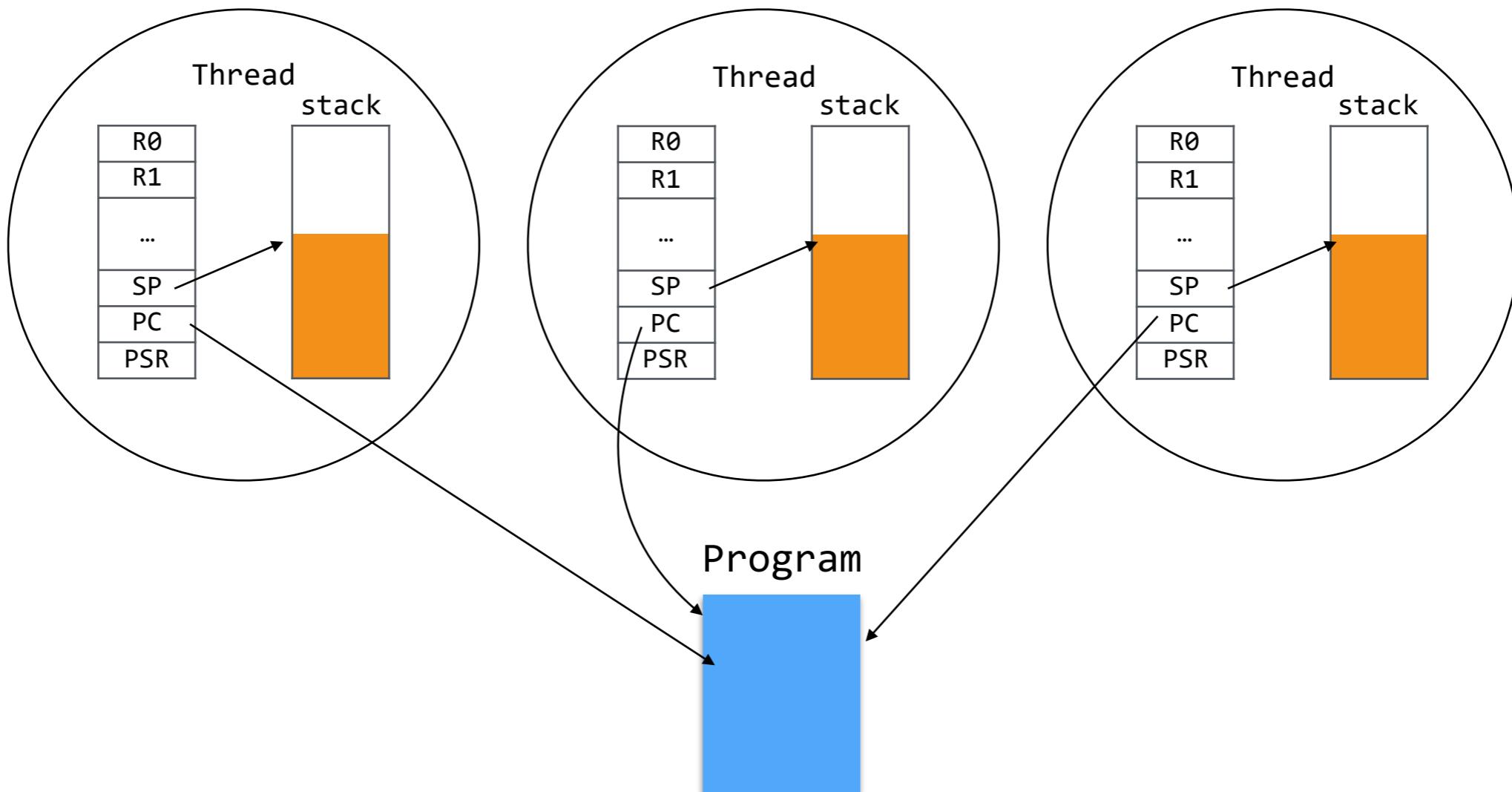


Busy-wait solution

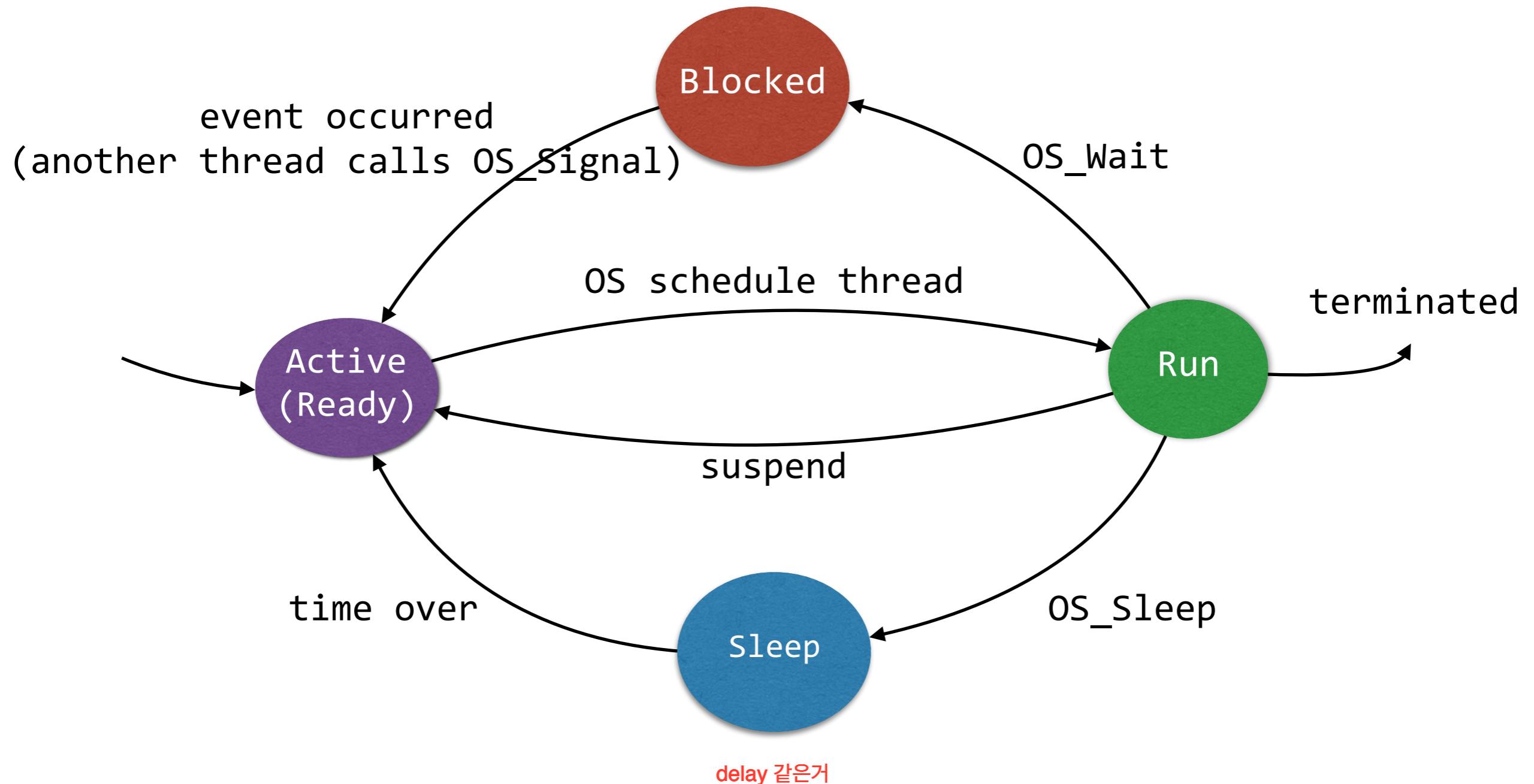
Multithread

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

우선 순위

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
```

use thread

```
void Task1(void){  
    Count1 = 0;  
    for(;;){  
        Count1++;  
        GPIOD->DATA ^= 0x02; // toggle PD1  
    }  
}  
    GPIOD는 외부에서 LED를 사용할때.  
    GPIOF는 내부 LED  
  
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){
    부팅되는 과정
    OS_Init();           // initialize, disable interrupts, 50 MHz

    SYSCTL->RCGCGPIO |= 0x20  

    SYSCTL->RCGCGPIO |= 0x08;      // activate clock for Port D
    while((SYSCTL->RCGCGPIO & 0x08) == 0){}; // allow time to stabilize
패치될때까지 기다려라

    GPIOD->DIR |= 0x0E;
    GPIOD->AFSEL &= ~0x0E;          GPIO가 아닌 다른 기능으로 사용할때 하는 해주는 것
    GPIOD->DEN |= 0x0E;          // enable digital I/O on PD3-1

    GPIOD->PCTL &= 0xFFFF000F;      GPIO가 아닌 다른 기능으로 사용할때 하는 해주는 것
    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);
```

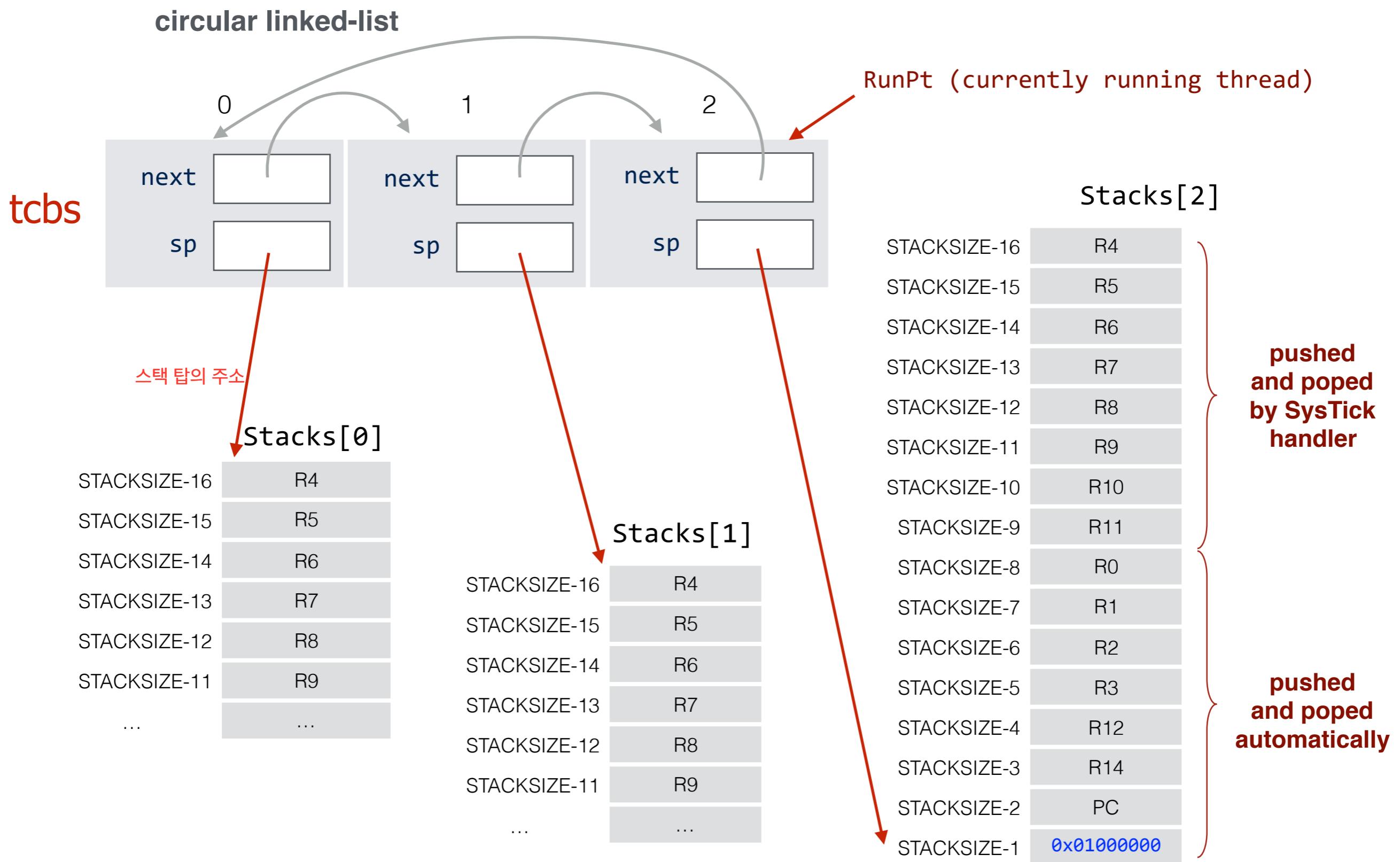
```
#define NUMTHREADS 3           // maximum number of threads
#define STACKSIZE 100          // number of 32-bit words in stack
```

```
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(); // 크리티컬한 작업이라서 방해받으면 안된다.  
  
    SysTick->CTRL = 0; // disable SysTick during setup  
                        systick timer를 disable  
    SysTick->VAL = 0;  
}
```

```

void SetInitialStack(int i){                                cpu context를 제외한 그위에 쌓이는 레지스터의 값들이 16개이다
    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
}

```

stacksize-2의 자리는 PC 값이 있어
없는 것이다. 중요해서 따로 저장한다,

의미없는 값이나 일부러 디버거에서 알기 쉬우라고 한 것이다.

```
int OS_AddThreads(void(*task0)(void),
                  void(*task1)(void),
                  void(*task2)(void)){
    int32_t status;
    status = StartCritical();//인터럽트를 disable해줌
    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

PCB를 관리하는 포인터
    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  
}
```

0부터 시작해서 그값에 도착하면 인터럽트함

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)
```

어셈블리 코드

DisableInterrupts

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

함수 호출에서 리턴 어드레스를 저장함

EnableInterrupts

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

타이머가 되면 포인터를 옮겨서 다음 thread를 수행한다.

인터럽트가 발생한 순간 중요한 것들은 자동으로 스택에 저장된다.

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
```

```
POP   {LR}              ; start location
```

function pointer
of first thread
(return address로 사용됨)

```
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

공동의 전역변수를 사용하면 경쟁상태가 될 수도 있다. 쓰기전에 인터럽트가 발생할 수 있다.

Semaphore

할꺼면 다하고 안할꺼면 포기하게 만들어야 한다.

랑데뷰 => 각자 수행하다가 일정 시점에서 같이 동작하고 다시 각자 수행하는 것..

- ☞ 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

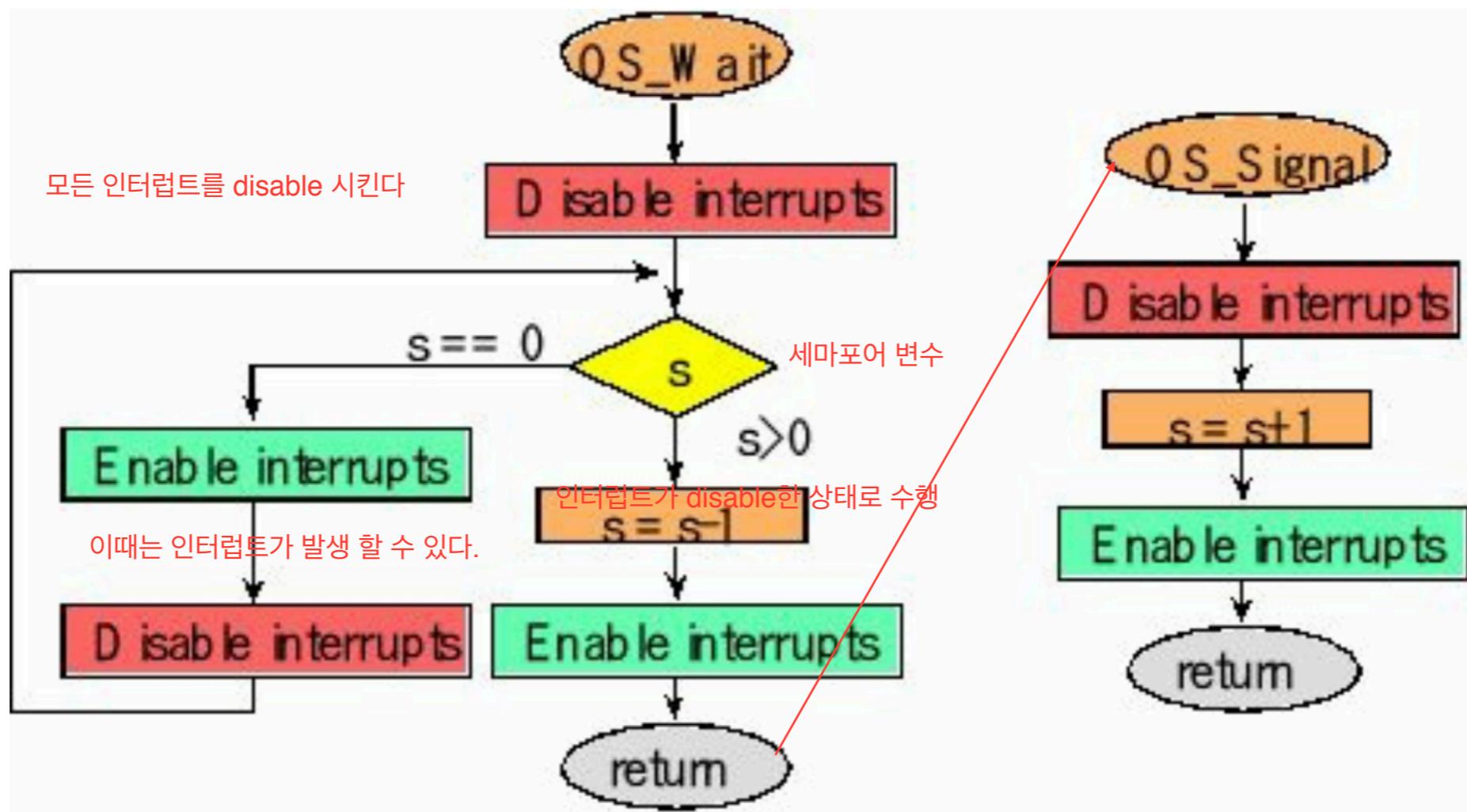
- ⦿ **OS_InitSemaphore**: 초기화, 처음에 한 번 실행
- ⦿ **OS_Wait (pend 혹은 P라고도 부름)**
- ⦿ **OS_Signal (post 혹은 V라고도 부름)**
- ⦿ 각각은 **atomic** 연산이어야 함

all and nothing

몇개의 기계어 단계로 나뉜다

Spin-Lock Semaphore

가장 단순한 semaphore 구현



동시에 어려명이 시도하면 대기번호를 뽑고 기다림
한번에 하나의 thread만 수행한다.

Spinlock semaphore

```
void OS_Wait(int32_t *s){  
    DisableInterrupts();  
    while((*s) == 0){  
        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();  
}
```

1로 초기화 해주어야 한다,

- 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  
    }  
}
```

Blocking semaphores

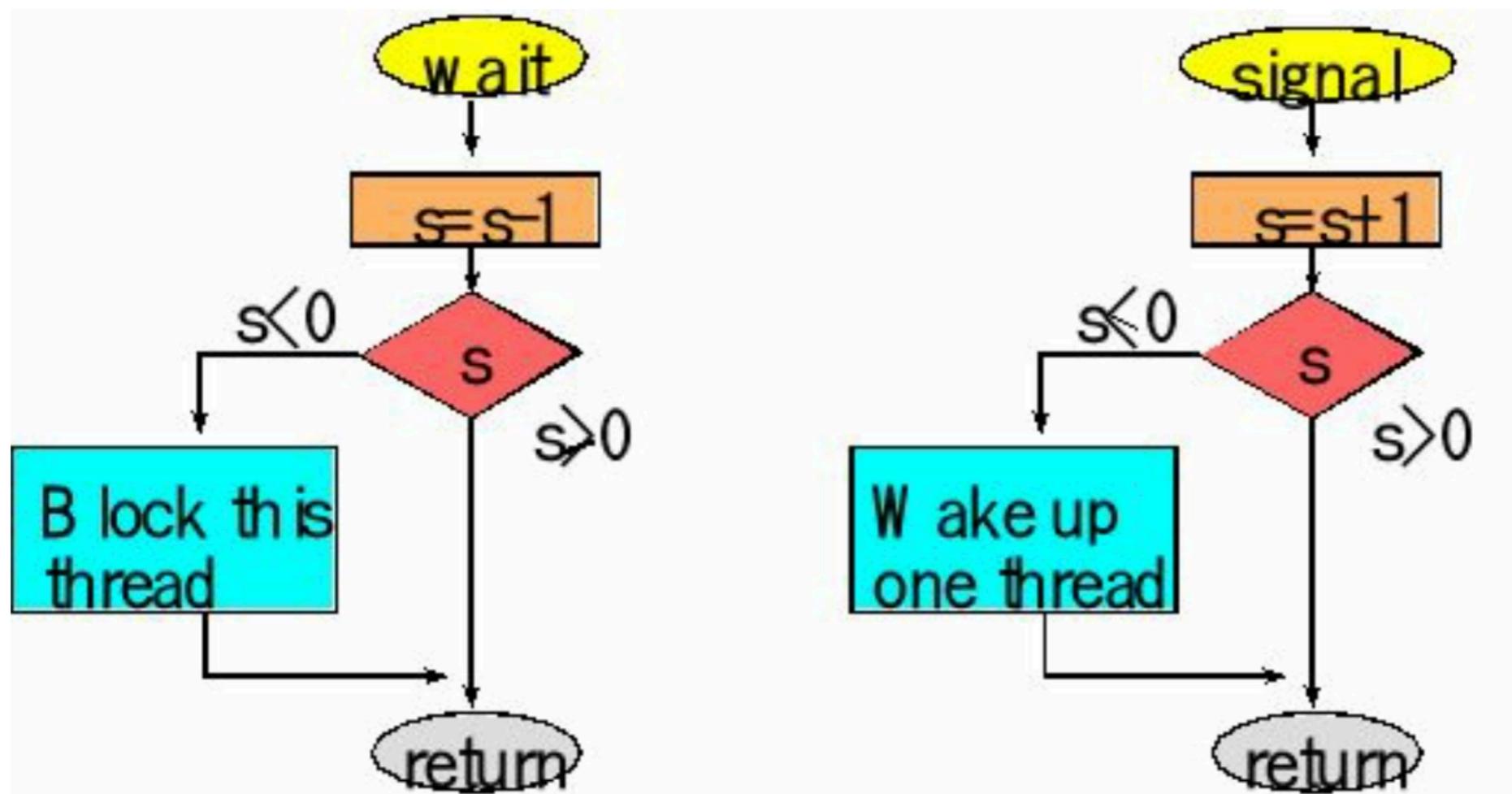
Suspend

- OS 혹은 실행중인 **thread** 자신이 (어떤 특정한 조건이 만족될 때까지는) 더 이상 의미있는 진도를 나갈 수 없다고 판단할 수 있다.
- 이때 실행 중인 **thread**를 중단하고 다른 **thread**를 실행하기 위해서 **OS_Suspend**를 호출한다.
- **INTCTRL** 레지스터의 **26번** 비트에 **1**을 쓰면 **SysTick** 인터럽트가 발생된다.
(이 레지스터의 다른 비트들에 **0**이 써지지만 이것은 아무런 효과도 없다.)

OS_Suspend

```
void OS_Suspend(void){  
    STCURRENT = 0;          // reset counter  
    INTCTRL = 0x04000000;    // trigger SysTick  
}
```

Blocking Counting Semaphore



Blocking Counting Semaphore

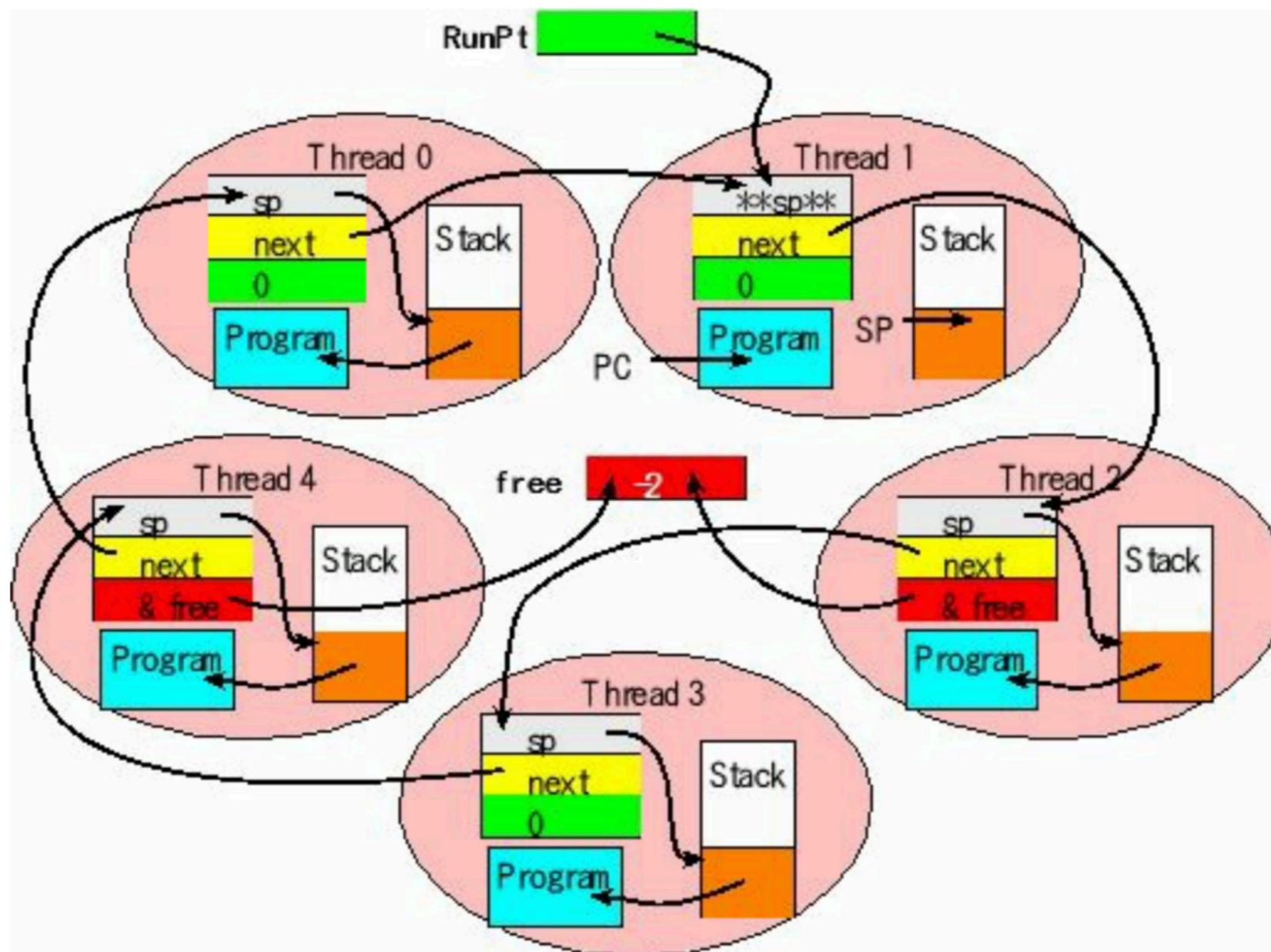


Figure 4.3. Threads 0, 1 and 3 are being run by the scheduler. Threads 2 and 4 are blocked on *free* and will not run until some thread signals *free*.

OS_Wait

```
void OS_Wait(int32_t *s){  
    DisableInterrupts();  
    (*s) = (*s) - 1;  
    if((*s) < 0){  
        RunPt->blocked = s;           // reason it is blocked  
        EnableInterrupts();           // run thread switcher  
        OS_Suspend();  
    }  
    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;
        }
        pt->blocked = 0; // wakeup this one
    }
    EnableInterrupts();
}
```

Thread rendezvous

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

        OS_Signal(&S1);
        OS_Wait(&S2);

        Stuff1();
    }
}
```

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

        OS_Signal(&S2);
        OS_Wait(&S1);

        Stuff2();
    }
}
```

```
#define FIFO_SIZE 10
uint32_t PutI;           // index of where to put next
uint32_t GetI;           // index of where to get next
uint32_t Fifo[FIFO_SIZE];
int32_t CurrentSize;    // 0 means FIFO empty, FIFO_SIZE means full
uint32_t LostData;       // number of lost pieces of data

void OS_FIFO_Init(void){
    PutI = GetI = 0;      // Empty
    OS_InitSemaphore(&CurrentSize, 0);
    LostData = 0;
}
```

FIFO

```
int OS_FIFO_Put(uint32_t data) {
    if(CurrentSize == FIFOSIZE). {
        LostData++;
        return -1;           //full
    }
    else {
        Fifo[PutI] = data;      // Put
        PutI = (PutI+1)%FIFOSIZE;
        OS_Signal(&CurrentSize );
        return 0;           // success
    }
}

uint32_t OS_FIFO_Get(void)      {
    uint32_t data;
    OS_Wait(&CurrentSize);      // block if empty
    data = Fifo[GetI];          // get
    GetI = (GetI+1)%FIFOSIZE;   // place to get next return data;
}
```