Network Manias White Paper

An Introduction to the Real-Time OS

2000년 3월 17일

Written by 유창모

URL http://user.chollian.net/~son6971

An Introduction to the Real-Time OS

유창모 (cmyoo@iae.re.kr)

최근의 embedded system 분야에서 Real-Time OS(이하 RTOS)를 탑재하여 개발된 제품들이 점차 늘어나고 있는 추세이며, 네트웍이나 멀티미디어 장비와 같이 시스템에서 처리해 주어야 하는 일의 양이 점점 많아지는 상황에서 RTOS 탑재는 너무나 당연한 것으로 인식되고 있다. 본 문서에서는 RTOS 이해를 위해 필요한 기본적인 개념에 대해서 설명하도록 하겠다.

1. RTOS & Embedded system

RTOS 설명에 앞서 Embedded system의 정의가 필요할 듯 싶다. **Embedded system**이란 PC나 Workstation과 같이 매우 많은 기능을 수행할 수 있도록 설계된 범용적 시스템(General purpose system)과는 달리, 특정한 작업/기능(specific function)만을 하도록 설계되어 있는 Computer system(CPU가 탑재된 하드웨어에 소프트웨어가 탑재된 시스템)을 말한다. 다른 시각에서 바라보면, PC나 Workstation을 제조하는 사람은 사용자가 이 시스템을 어떤 용도로(Network server, Game, Internet, Word Processor등) 사용할 것이지 알 수 없지만, Embedded system 제조업자는 이 시스템이 어디에 사용될 것이라는 것을 분명히 알 수 있다.(Cisco에서 GSR12000(고속 라우터)을 만들면서, 이 장비가 어디에 쓰일지 모를 리가 있겠는가?)

RTOS는 바로 이 Embedded system에 탑재되어 지는 운영체제이다. 처리해야 할 작업이 많아지면서 복잡해진 Embedded system에서 가장 필요했던 기능은 Multitasking이었다. 처리할 여러 개의 작업들을 task로 나누어 처리해야 했기 때문이다. 따라서 예전에 Computer system에서만 쓰이던 Multitasking OS가 Embedded system에서도 필요하게 된 것이다. 그러나 일반 Computer system에서 쓰이는 Multitasking OS와는 달리 대부분의 Embedded system은 Real-time이라는 특성을 만족시켜야 했기 때문에 나오게 된 것이 바로 Real-Time OS이다. RTOS의 Real-Time(실시간)이라는 단어 때문에, 모든 작업을 실시간으로 처리할 수 있는 환경을 제공하는, 매우 빠른 운영체제라고 생각할 수도 있지만, 그렇지는 않다. 여기서의 Real-Time이란, 임의의 정보가 시스템에 입력 됐을 때, 주어진 시간 안에 작업이 완료되어 결과가 주어져야 하는 것을 의미한다. 즉, 단지 빨리 처리해야 하는 것이 아니고 정해진 시간을 넘겨서는 안된다는 것이다. **RTOS**는 주어진 작업을 정해진 시간 안에 수행 할 수 있는 환경을 제공하는 운영체제이다.(예측 가능하고 일정한 응답 시간을 요구하는 응용 프로그램의 지원을 위한 운영체제)

Real-time System

- provides deterministic response to an external events
- has the ability to process real-word data at its rate of occurrence
- is deterministic in its functional & timing behavior

- whose timing is analyzed in the worst cases, not in the typical, normal cases to guarantee a timing response in any circumstances

RTOS should be deterministic and have guaranteed worst-case interrupt latency and context switch times. An Operating System is said to be deterministic if the worst-cast execution time of each of the system call is calculable.

그러면, RTOS는 Windows나 UNIX와 같은 Multitasking OS와는 어떤 차이가 있을까? Multitasking 환경을 제공한다는 면에서는 동일하지만, General purpose system에 탑재되는 OS의 경우에는 하드웨어 자원(메모리, I/O 디바이스, 하드디스크 등)을 얼마나 효율적으로 사용하고, 사용자들에게 얼마나 공평하게 이 자원을 분배할 것인지에 초점을 맞추고 있는 반면, RTOS는 정해진 시간 제약을 해결하는데 초점을 맞추고 있다. 그런 이유로 RTOS에서는 하드웨어 자원을 좀 낭비하더라도 작업의 시간 제한을 맞추려 하고, 공평성의 개념보다는 우선 순위가 높은 task가 많은 시간 동안 수행할 수 있도록 한다.

2. RTOS의 기본 개념

Soft Real-Time vs. Hard Real-Time

Real-time system 중에는 어떤 작업을 일정 시간 안에 반드시 처리해야 하며, 그 시간이 지난 후의 결과 값은 정확해도 소용이 없는 경우가 있다. 이런 시스템을 Hard Real-Time system이라고 한다.(예: 군사장비, 비행기) 이에 반해서 어떤 시간 안에 처리하면 좋지만 그렇지 못한 경우에 그 시간에서 약간 경과한 후의 값도 인정하는 경우를 Soft Real-Time system이라고 한다.(예: cellular phone, router)

Task & Multitasking: Basic concepts

Windows95/98 시스템의 경우, winamp를 통해 mp3 음악을 들으며 아래 한글 문서를 만들고, 그리고 ftp 프로그램으로 파일을 다운로드 받을 수 있다. 이 경우 winamp, 아래 한글 그리고 ftp client 프로그램을 각각의 task로 정의할 수 있다. 이처럼 여러 개의 task를 동시에 실행 할 수 있는 운영체제를 **Multitasking OS**라 한다. 이와 반대되는 것이 Singletasking이며 DOS가 그 예이다.

하나의 processor(CPU)는 어느 한 시점에 한 가지 일(task)만을 할 수 있으며, 이를 긴 시간(수 msec) 동안 보았을 때, 마치 여러 개의 task가 동시에 수행되는 것처럼 보이는 것이 multitasking이다. multiprocessor system의 경우처럼 어느 한 시점에 여러 가지의 일을 하는 것은 아니다.

위 예의 경우에는 3개의 task가 각각 무관한 프로그램들이지만, Embedded system에서의 task는 하나의 문제(목적)을 풀어나가기 위해서 하나의 큰 응용 프로그램을 논리적으로 나눈 개념으로 볼 수 있다. 어떤 목적을 수행하기 위해서 여러 기능들이 동시에 수행될 필요가 있고 이를 순차적으로 프로그램 하기 어렵게 때문에, 기능 블록의 모듈화 및 CPU의 효율적인 사용이라는 목적에서 Multitasking이라는 개념이 나온 것이다.

adsl router를 예로 들자면, 이 장비는 사용자 단말들을 ethernet으로 연결하여 adsl line을 통해 internet을 액세스하는 것이 목적이다. 이 하나의 기능 수행을 위해 다음과 같이 여러 개의 task로 나누어 각각의 임무를 수행한다. (FlowPoint 장비 manual에 나와있는 task 리스트의 일부)

- IDLE TASK
- PPP(Point-to-Point Protocol) TASK
- IP(Internet Protocol) TASK
- UDP(User Datagram Protocol) TASK
- TCP(Transmission Control Protocol) TASK
- RIP(Routing Information Protocol) TASK
- ATM(Asynchronous Transfer Mode) TASK
- SNMP(Simple Network Management Protocol) TASK

이처럼 여러 개의 task가 하나의 목적을 위해서 맡은 바 임무를 수행하기 위해서는 각 task 간에 유기적인 관계가 필요하며, 뒤에서 설명할 task synchronization, task communication등이 이를 위한 개념들이다.

Task & Multitasking: Advanced Concepts

각 task는 기본적으로 priority, status 정보, 그리고 stack 영역을 가지고 있으며, 이 정보들은 TCB(Task Control Block)라는 곳에 저장된다.(즉, task마다 독립된 TCB를 가짐)

◆ Task priority

Task priority는 이름 그대로 각 task들의 우선 순위를 나타내는 것으로, priority가 높은(높은 숫자 값이 높은 priority인 경우도 있고, 그 반대일 수도 있다. 이는 RTOS 구현에 따라 달라짐) task가 항상 CPU를 점유하여 실행 할 수 있다.(Preemptive kernel)

Priority는 task간의 상대적 개념이지 절대적인 값이 의미를 갖지는 않는다. 즉, 2개의 task가 있다고 했을 때, 둘 중에 누가 높은지가 중요한 것이지, 얼마나 높냐는 의미가 없다. RTOS 환경 하에서 응용 프로그램을 개발하는 경우에, task를 어떻게 나눌 것이며 각 task에 얼마의 priority를 줄 것인지는, 개발자가 결정해야 하는 매우 중요한 문제이다.

◆ Task status

Task status는 아래 그림과 같이 5개의 상태로 나눌 수 있다.(RTOS 구현에 따라서 달라질 수 있지만, 기본 개념은 모두 동일함) 각 task는 Scheduler에 의해서, 혹은 외부의 task, 혹은 자신의 힘으로 상태가 변화된다.

- DORMANT: memory에는 존재하나(그 task에서 사용하는 instruction/data들이 메모리에 존재한다는 의미), 아직 실행할 수는 없는 상태. task를 생성하면 본 상태로 되지만, 어떤 RTOS들은 이 상태가 존재하지 않고, 생성 시에 바로 READY상태가 되는 것도 있다.
- RUNNING: CPU를 사용(점유)하고 있는 상태.(task의 코드가 동작하고 있는 상태)
- READY: 현재 CPU를 사용하고 있는 task(RUNNING상태의 task)보다 priority가 낮아서, CPU 사용의 기회를 기다리고 있는 상태
- WAITING: 어떤 이벤트(serial로부터의 데이터 수신, 세마포어등)를 기다리고 있는 상태. 만약 기다리고 있던 이벤트가 발생하면, READY상태로 되어 CPU 사용기회를 기다리게 된다.

ISR(Interrupt Service Routine)은 task 상태에 포함되지 않는다. 인터럽트가 발생하면, RUNNING상태에 있던 task가 CPU 제어권을 빼앗기고, ISR이 수행된다.

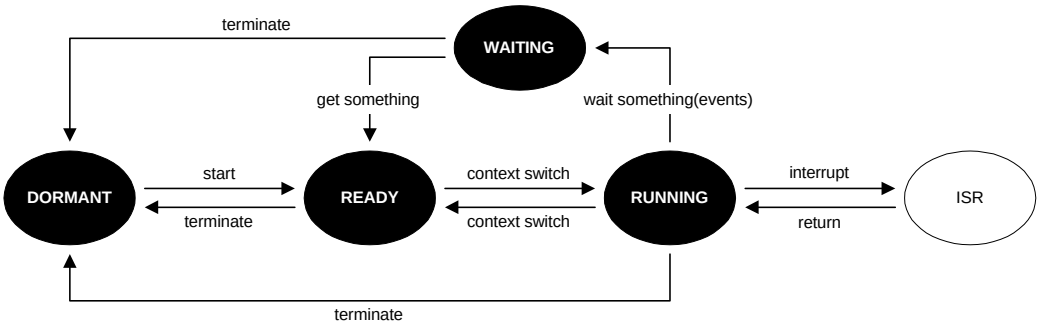

Figure : Task states

[예] High priority taskA는 serial로부터 데이터를 수신하는 일을 하고, Low priority taskB는 주기적으로 LED를 on/off하는 일을 한다고 가정하면, 2개의 task는 다음과 같은 상태변화가 발생한다.

❶ taskA가 taskB보다 priority가 높기 때문에 RUNNING상태가 되고, task B는 CPU 사용의 기회를 기다리는 READY상태가 된다.

❷ taskA는 serial로부터 데이터가 수신되지 않았기 때문에 이를 기다리기 위해서 WAITING상태가 되고, Scheduler에 의해서(Context Switch가 발생하여) taskB가 RUNNING상태가 된다. 이제 taskB는 루프를 돌면서 주기적으로 LED를 on/off한다.

❸ serial로부터 데이터가 수신되면, 인터럽트가 발생하고 이에 의해서 ISR이 수행된다. ISR에서는 인터럽트에 대한 처리를 수행(인터럽트를 처리하면서, taskA가 serial 데이터를 기다리고 있음을 알고서, 이 task를 WAITING에서 READY상태로 만든다)하고, Schedule(Scheduler기능을 수행하는 함수)r를 호출한다.

❹ Scheduler에 의해서(Context Switch가 발생하여) ISR 종료와 동시에 WAITING상태에 있던 taskA가 RUNNING상태가 되고, taskB는 READY상태로 된다.

◆ Task stack

task는 우리가 일반적으로 생각하는 하나의 프로그램(함수들의 모임)으로, 어떤 프로그램이 수행될 때 함수를 호출할 것이며, 그 과정에서 stack이 쓰이게 된다.(함수 안에서 선언한 지역변수, 함수의 아규먼트, 함수 수행 후 돌아갈 리턴 어드레스 등이 stack에 저장된다) task마다 메모리에 독립된 stack 영역을 가지게 되고, 다른 task들은 이 영역을 액세스할 수 없다.

각 task의 stack size는 task 생성 시에 결정되어지며, 적절한 stack 크기를 할당하는 것은 구현상에 중요한 문제이다. stack이 overflow날 경우에 시스템이 엉뚱하게 동작 하다고 다운되기 때문에 디버깅하는데(stack overflow 문제라고 찾아내는데 까지) 꽤 고생하게 된다.

앞에서 task의 priority, status 그리고 stack 영역의 위치 등이 TCB에 저장된다고 설명했다.(물론 이보다 훨씬 많은 정보들이 TCB에 저장됨) 이를 그림으로 나타내면 아래와 같다.

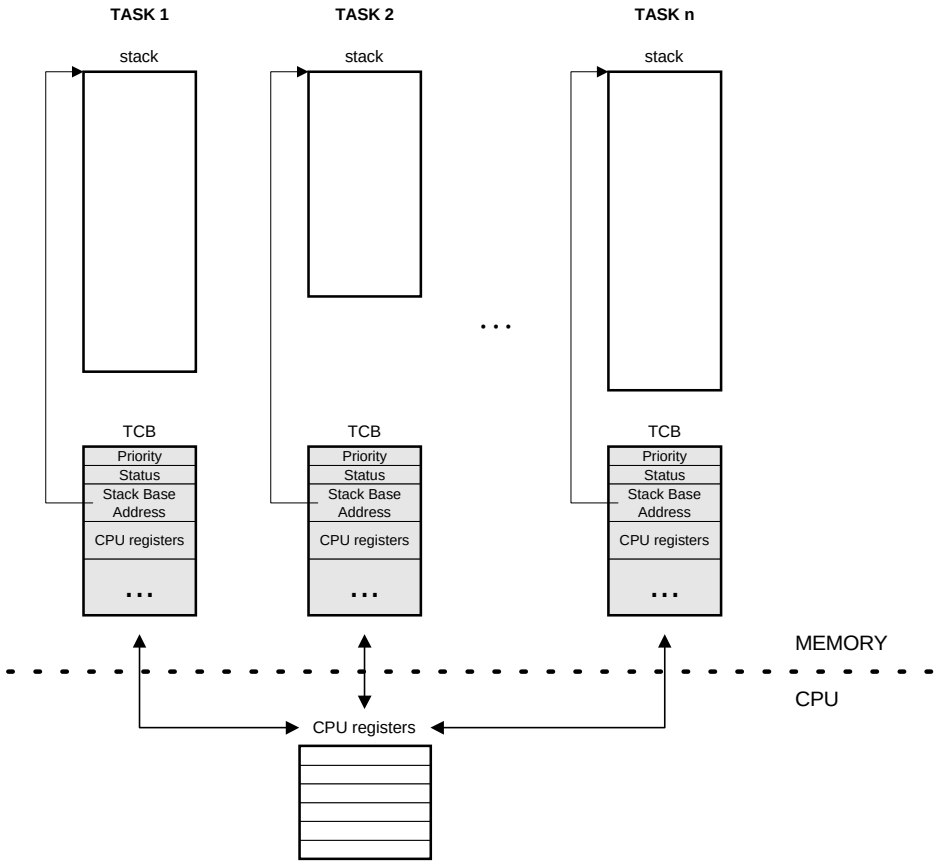

Figure : Multiple tasks

Scheduler(Dispatcher)

Scheduler는 READY상태에 있는 여러 개의 task 중에, 다음에 어떤 task를 수행시킬 것인지(RUNNING상태로 만들 것인지)를 결정하는 책임을 맡는다. 일반적으로 가장 높은 priority의 task를 수행시키는 "priority-based scheduling" 방법을 사용한다.

앞에서 설명한 Real-time의 조건(주어진 시간 안에 작업이 완료)을 만족시키기 위해서는, "Earliest Deadline First"를 사용하는 것이 바람직하지만, 구현상의 어려움(복잡함-이로 인한 kernel의 오버헤드)과 task간 통신/세마포어 사용의 문제 때문에 상용 RTOS에서 이 방법을 사용하지 않는다. RTOS kernel에서 제공하는 시스템 콜에 대해서는 deterministic time을 보장하므로, 사용자 코드의 수행 소요 시간이 계산된다면, Real-time 조건을 만족하는 시스템을 만들 수 있게 되는 것이다.

Earliest Deadline First : 각 task가 해야 하는 일의 종료 시간(Deadline)을 정해놓고, Deadline까지 가장 시간이 남지 않은 task를 수행시키는 방법이다.

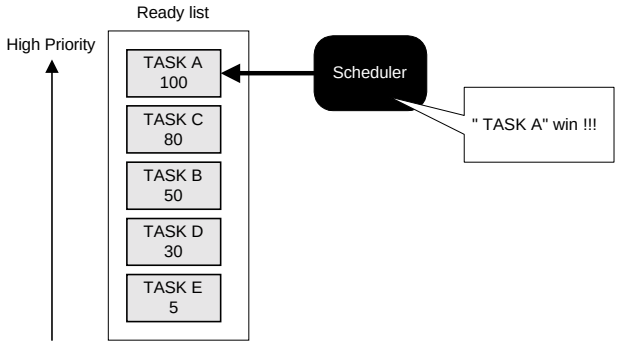

Figure : Priority based Scheduler

Context Switch(Task Switch)

다음 문장들은 모두 같은 의미로 사용된다.

- task가 RUNNING상태에 있다.
- task가 CPU를 사용(점유)하고 있다.
- task가 CPU의 레지스터를 사용하고 있다.

여기서 task가 사용하는(또는 사용하던) CPU 레지스터들의 값을 **Context**라고 한다.

사실 (Task) Context는 위에서 설명한 것 보다 좀더 포괄적인 의미를 가지고 있다. task가 유지해야 하는 정보들의 모임이라고나 할까?

Scheduler에 의해서 새로이 RUNNING상태가 될 task가 결정되면, 현재 RUNNING상태의 task가 사용하던 Context를 메모리 특정 영역(이는 아래 그림처럼 TCB가 될 수도 있고, 아니면 스택이 될 수도 있음)에 저장한 후, 새로이 수행될 task의 Context를 TCB 또는 스택에서 CPU의 레지스터 영역으로 복사하여 새로운 task가 수행되도록 하는 작업을 **Context Switch** 또는 Task Switch라고 한다.

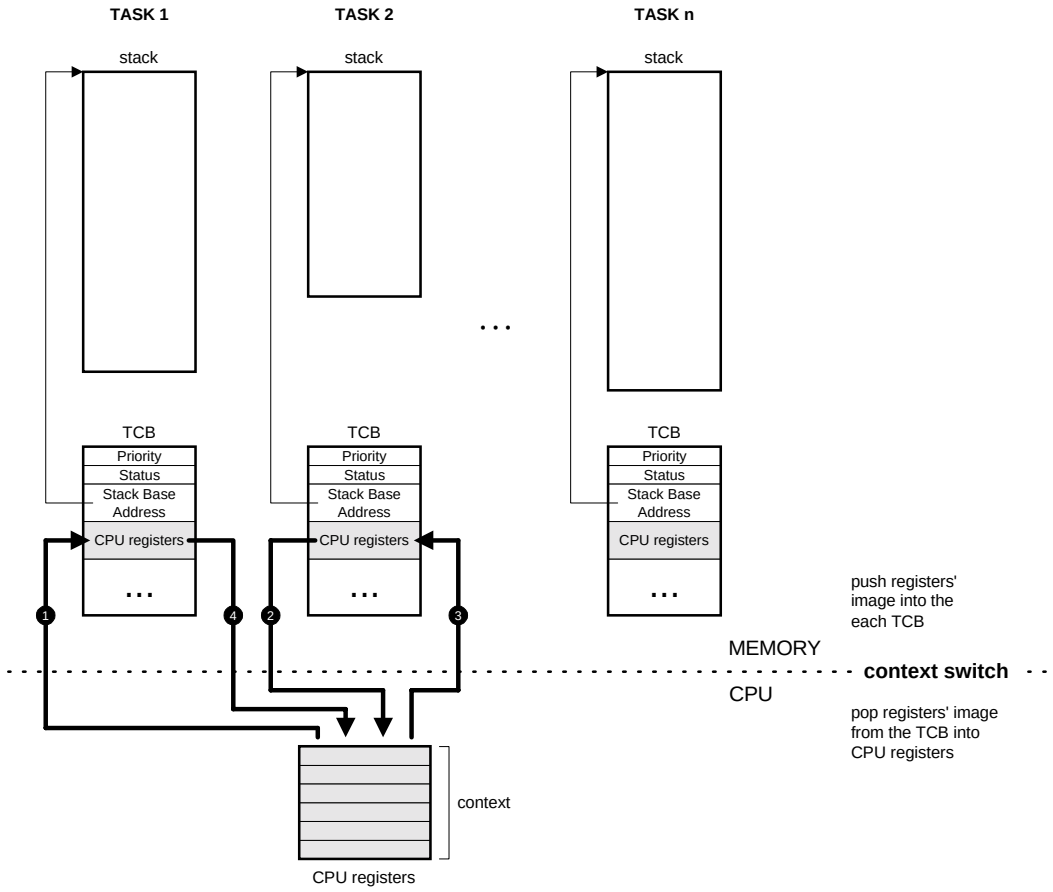

Figure : Context Switch

위의 그림에서는 Context를 TCB안에 - TCB는 여러개의 정보(필드)들을 가지는 C-언어의 스트럭쳐 모양으로 구성될 것이고, 이 스트럭쳐 필드 중에 Context를 저장할 수 있는 필드가 있음 - 저장하는 모양으로 되어 있지만, 이 Context는 TCB가 아닌 stack에 저장될 수도 있다. 이는 RTOS 구현상에 문제이다.

[예] 현재 task1이 수행(RUNNING 상태)되고 있고, Scheduler에 의해서 READY상태에 있던 task2가 수행되어야 할 경우에, 첫번째 Context Switch(❶, ❷)가 발생한다.

❶ task1의 Context(task1이 사용하던 CPU 레지스터들의 값)를 task1의 TCB에 저장.

❷ task2의 TCB에 저장되어 있던 Context를 CPU 레지스터로 복사. 이 시점부터 task2의 코드가 수행된다.

Scheduler에 의해서 task1을 다시 RUNNING상태로 만든다면, 두번째 Context Switch(❸, ❹)가 발생한다.

❸ task2가 사용하던 CPU 레지스터 값들(Context)을 task2의 TCB에 저장

❹ task1의 TCB에 저장되어 있던 Context를 CPU 레지스터 값으로 복사.

Context Switch가 이루어지는 동안은 실제 응용 프로그램에서 원하는 작업이 이루어지지 않기 때문에 오버헤드라고 볼 수 있다. 그 오버헤드 시간은 저장(save) 및 복귀(restore)할 레지스터의 개수에 따라서 달라지게 된다.

Scheduler와 Context Switch를 다시 정리해 보도록 한다.

- Scheduler는 어느 task를 RUNNING상태로 만들지를 결정하는 일을 담당한다.
- RUNNING상태에 있던 task가 priority가 가장 높을 경우에는 Scheduler는 그냥 그 task를 계속 RUNNING상태로 놔두게 되고, 다른 높은 priority의 task가 READY상태에 있을 경우에는, 현재 RUNNING상태의 task를 READY상태로 (TCB의 status값을 RUNNING에서 READY로 변경), 그리고 READY상태에 있던 가장 높은 priority의 task를 RUNNING상태로 만든다(TCB의 status값을 READY에서 RUNNING으로 변경).
- Context Switch는 Scheduler의 결정에 따라서 다음 기능을 수행한다.
  ❶ RUNNING상태에 있던 task가 사용하던 Context를 TCB에 저장
  ❷ 새로 RUNNING상태가 될 task의 TCB에 저장되어 있던 Context를 CPU 레지스터로 복사
- RUNNING상태에 있던 task가 Scheduler의 결정 및 Context Switching에 의해서 CPU 제어권을 빼앗기게 되는 상황을 그 task가 preemption되었다고 말한다.
- Scheduling이 일어난다고(Scheduler가 수행된다고) 해서 반드시 Context Switch가 발생하는 것은 아니다.(RUNNING상태에 있던 task가 가장 priority가 높으면, Scheduler는 그 task를 계속 RUNNING상태로 놔두게 되므로, Context Switch가 발생하지 않는 것이다.)
- 보통 Scheduler 안에서 Context Switch가 호출되며, 이를 pseudo code로 표현하면 다음과 같다.

```
void Scheduler(void)
{
    t1_pri = current task(RUNNING state)'s priority;
    t2_pri = highest priority in the Ready list;
    if (t1_pri < t2_pri)
        call Context_Switch();
    else
        ; /* do nothing */
}
```

Non-preemptive kernel

비선점형 커널은 어떤 한 task가 수행하고 있을 때, kernel이 그 task의 수행을 강제로 중지시키고 다른 task를 수행시킬 수 있는 능력이 없으며, 다른 말로 cooperative multitasking이라고도 한다. 이 구조에서는 priority가 낮은 task의 수행에 의해서 높은 priority의 task가 무한정 기다릴 수 있는 상황이 발생 할 수 있기 때문에(이를 response time이 nondeterministic하다고 함), Real-time system에서는 사용될 수 없는 구조이다. Windows 3.1이 Non-preemptive kernel의 예이다.

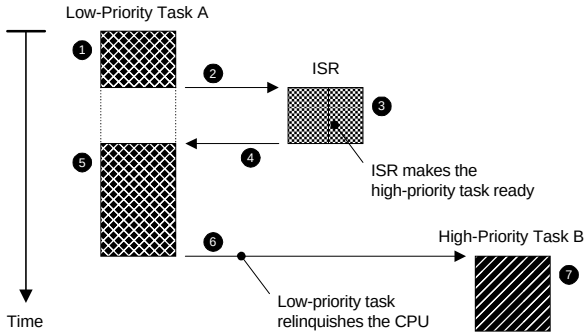

Figure : Non-preemptive kernel

[예]

❶ Low priority taskA가 수행 중
❷ 인터럽트 발생
❸ ISR에서 인터럽트 처리를 하고 Scheduler를 호출하면, 현 task 보다 높은 priority를 가진 task를 READY상태로 만듬(예를 들어 serial로부터 데이터가 수신되었다는 인터럽트가 발생하였고, High priority taskB가 이를 처리해야 할 경우)
❹ ISR의 수행이 종료되고, ISR 이전에 수행하던 Low priority taskA로 돌아감
❺ 인터럽트 발생 직전의 코드부터 다시 수행
❻ taskA가 kernel system call을 호출하여, CPU 사용권(제어권)을 포기
❼ kernel에 의해서 taskB가 수행(❸에서 예를 든 serial 데이터에 대해서 처리 할 것임)

Preemptive kernel

선점형 커널은 어떤 한 task가 수행하고 있는 도중에도 kernel이 그 task의 수행을 중지시키고 다른 task(중지되는 task 보다 priority가 높은)를 수행시킬 수 있는 능력을 가지고 있다. 이는 CPU를 사용할 준비가 된 가장 높은 priority의 task가 항상 CPU를 점유하여 수행될 수 있다는 deterministic한 특성을 가지고 있기 때문에 RTOS에서는 이 구조를 사용한다. 이 구조의 예로는 Windows 95/98/NT, UNIX가 있다.

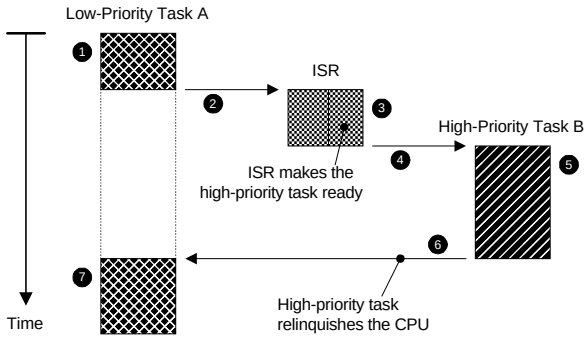

Figure : Preemptive kernel

[예]

❶ Low priority taskA가 수행 중

❷ 인터럽트 발생

❸ ISR에서 인터럽트 처리를 하고 Scheduler를 호출하여 현 task 보다 높은 priority를 가진 task를 READY 상태로 만듬(예를 들어 serial로부터 데이터가 수신되었다는 인터럽트가 발생하였고, High priority taskB가 이를 처리해야 할 경우)

❹ ISR 종료와 동시에 Context Switch에 의해서 ISR 이전에 수행하던 taskA (interrupted task)가 아닌 High priority taskB가 CPU 제어권을 가짐

❺ taskB 수행(❸에서 예를 든 serial 데이터에 대해서 처리 할 것임)

❻ taskB가 kernel system call을 호출하여, CPU 제어권을 포기하고 kernel에 의해서 taskA가 선택됨

❼ taskA의 인터럽트 발생 직전의 코드부터 다시 수행

Critical Section(Region)

다른 task에 의해서 중단되어서는 안 되는 일련의 코드 블록을 **Critical Section**이라고 한다. 즉, 이 블록의 코드를 시작하게 되면, 코드 블록의 종료까지 Context Switch 발생에 의한 다른 task의 수행이 없어야 한다.

Mutual Exclusion

예를 들어서 두개의 task가 동시에 프린터를 쓰려고 한다고 가정하자. 만약 task1이 프린트하는 도중에 Context Switching이 발생하여 task2가 실행되고, 다시 Context Switching이 발생하여, task2가 실행하는 도중에 다시 task1이 실행된다면, 결과는 task1과 task2가 출력하려는 문자들이 마구 뒤섞이게 될 것이다. 이와 같이 복수 개의 task가 공유하여 사용하는 **공유자원**(shared resource: 전역 변수(global variables)일수도 있고, 위의 예와 같은 I/O device일 수도 있음)에 대해서 Mutual Exclusion(상호 배제)이 보장되어야 한다. **Mutual Exclusion**이란, 하나의 task가 공유자원을 사용하고 있는 동안은 다른 task가 이 자원을 사용하지 않는다는 것을 보장하는 개념이다. Mutual Exclusion을 보장하기 위해서는 다음과 같은 방법이 사용될 수 있다.

- 공유자원을 사용하는 동안은 인터럽트를 금지시킨다.

```
Disable interrupts;
Access(read/write) the shared resource;
Enable interrupts;
```

  인터럽트를 금지시키고 있는 동안은 task가 스스로 CPU 사용권을 포기하지 않는 이상, preemption당하지 않기 때문에, 공유자원을 사용할 수 있다. 매우 간단한 방법이기는 하지만, 인터럽트 금지 시간이 길어지면 문제가 발생(이후에 발생하는 인터럽트를 잊어버리고, 이에 의해서 RTOS의 deterministic 특성의 파괴)하므로, 되도록 빠른 시간 안에 다시 인터럽트를 enable해야 한다.

- 공유자원을 사용하는 동안은 Scheduling을 금지시킨다.

```
Disable scheduling;
Access(read/write) the shared resource; (interrupts are recognized)
Enable scheduling;
```

  Scheduling만을 금지하였기 때문에, 공유자원을 액세스하는 동안 인터럽트가 발생할 수 있다. 만약 ISR에서 공유자원을 액세스하게 될 경우에는 Mutual Exclusion을 보장할 수 없기 때문에, task 간에 공유되는 자원에 대해서만 사용할 수 있는 방법이다. 응용 프로그램에서 이 방법을 많이 사용하게 될 경우에, 앞에서 설명한 높은 priority task가 CPU를 점유할 수 있는 시점이 지연되기 때문에 deterministic의 특성이 파괴되는 문제가 있다.

- Semaphore를 사용한다.
  가장 좋은 방법으로 아래에서 자세히 설명하도록 하겠다. 단, 전역 변수 하나의 액세스와 같이 공유자원 access time이 짧은 경우에는, 오히려 semaphore 코드 수행 시간이 더 길어질 경우가 있으며, 이 경우에는 위의 2가지 방법 중 하나를 사용하는 것이 더

낳을 것이다.

Semaphore

1960년대 중반에 Edgser Dijkstra에 의해서 고안된 것으로 대부분의 RTOS에서 본 mechanism을 지원한다. Semaphore는 아래 그림과 같이 열쇠(Key)라고 할 수 있다. 공유자원을 액세스하기 위해서는 Key를 먼저 얻어와야 하고, 액세스가 끝나면 다른 task가 사용할 수 있도록 그 Key를 다시 반환하는 것(mechanism)이다. 만약, 다른 task가 이미 Key를 사용하고 있는 중이라면, 그 Key를 얻으려는 task는 WAITING상태가 되고, 그 Key가 사용가능 할 때(Key를 가지고 있는 task가 그 Key를 돌려줄 때) 다시 RUNNING상태가 되어, 그 Key를 받아와 공유자원을 사용할 수 있게 되는 것이다.

이 Semaphore는 아래 그림과 같이 Binary Semaphore와 Counting Semaphore로 나누어진다. Binary Semaphore는 하나의 Key만을 가지고 있는 경우이고, Counting Semaphore는 N개의 Key가 존재하는 경우이다.

Key라고 표현된 Semaphore를 구현할 때, counter를 사용하는데 이 counter는 Semaphore가 생성될 때 초기값이 정해지고, 어느 task가 Semaphore를 얻어 가면 counter가 1 감소하여 그 값이 0보다 크거나 같으면 그냥 리턴하고, 음수일 경우에는 현 task를 WAITING상태로 바꾸고 Scheduling이 발생한다. 그리고 Semaphore를 가지고 있던 task가 이를 반환하면, counter가 1 증가한다. Binary Semaphore는 counter가 1과 0 둘 중에 한 값이 되고, N counting Semaphore는 counter가 0에서 N 중에 한 값이 된다.

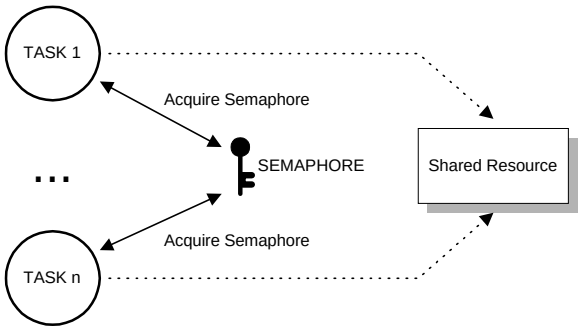

Figure : Binary Semaphore

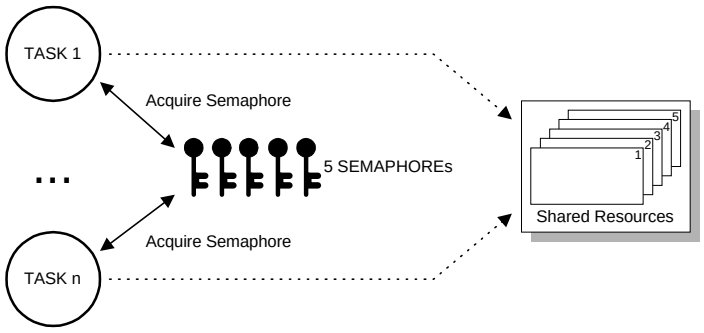

Figure : Counting Semaphore

복수개의 task가 하나의 Semaphore를 기다리고 있다고 가정하자. Semaphore를 가지고

있던 task가 Semaphore를 반환했을 경우에 복수개의 task 중에 어떤 task가 그 Semaphore를 가질 수 있을까? 두 가지 경우를 생각할 수 있다. 첫번째는 priority based 방법으로, priority가 가장 높은 task가 가지는 것이고, 두번째는 FIFO based 방법으로, 가장 먼저 Semaphore를 가지려 한 task(가장 오래 기다린 task)가 가지는 것이다. 이는 RTOS 구현에 따라 두 가지 중 하나를 지원할 수도 있고, 또는 두 가지 모두를 지원할 수도 있다.

Task Synchronization

Semaphore는 Mutual Exclusion 지원 외에도 두개의 task간 또는 task와 ISR간의 동기화(Synchronization)를 지원한다.

[예] 아래와 같은 예를 들어, 동기화의 의미를 살펴보도록 한다. taskA는 전역변수 "N"을 1부터 2000까지 증가시키는 일을 하고, taskB는 2000번의 루프를 돌면서 "N" 값을 출력하는 일을 한다. 두 task의 priority가 같다고 했을 때 어떤 결과가 나올까?

```
int    N = 0;

/* taskA */
void taskA(void)
{
    int    i
    for (i = 1; i <= 2000; i++)
        N++;
}

/* taskB */
void taskB(void)
{
    int    i
    for (i = 1; i <= 2000; i++)
        printf("N is %d\n", N);
}
```

한번 정도는 "N is 1"을 출력하고(경우에 따라서는 안 그럴 수도 있음), 그 다음에는 거의 2000에 가까운 수나 2000을 출력할 것이다. 그리고는 2000번의 루프가 종료될 때까지 2000이라는 값만을 출력한다. 두개의 task priority가 같을 경우에, 두개의 task는 서로 공평하게 CPU를 사용하는데(priority-based scheduling을 하니까), taskA에서의 "N++" 명령은 taskB의 "printf("N is %d\n", N)" 함수의 수행시간에 비해서 굉장히 짧기 때문에(assemble 코드 양을 비교하면, 수백 배의 차이가 있을 것임), taskA는 매우 빠른 시간에 N을 2000까지 증가시키고 종료하며, 그 이후에 2000이라는 수를 taskB가 출력하는 것이다.

1부터 2000까지의 수를 순차적으로 출력하기 위해서는 동기화 방법을 사용해야 한다. 이 두개의 task가 동기 되어 동작하려면, 한번씩 교대로 루프를 돌면 될 것이다. 이를 위해서 2개의 Semaphore를 사용한다.

```
int    N = 0;

/*
    Initially,
    semaphoreX count = 0
    semaphoreY count = 1
*/

/* taskA */
void taskA(void)
{
    int    i
    for (i = 1; i <= 2000; i++)
    {
        Take semaphoreX;     /* attempt to get semaphoreX */
        N++;
        Give semaphoreY;     /* release semaphoreY */
    }
}

/* taskB */
void taskB(void)
{
    int    i
    for (i = 1; i <= 2000; i++)
    {
        Take semaphoreY;     /* attempt to get semaphoreY */
        printf("N is %d\n", N);
        Give semaphoreX;     /* release semaphore X*/
    }
}
```

❶ semaphoreX count 초기값은 0(Key가 하나도 없는 상태), semaphoreY count 초기값은 1(Key가 하나 있는 상태)로 정한다.

❷ semaphoreX가 없기 때문에, semaphoreX를 얻으려는 taskA는 WAITING상태가 되고, taskB가 수행된다.

❸ taskB는 semaphoreY를 가지고 오고 printf()를 수행한다. 그리고 semaphoreX를 반환한다.

❹ taskB가 semaphoreX를 반환했기 때문에, 이를 기다리던 taskA가 수행될 수 있는 기회를 가진다. taskA로 Context Switch가 일어난다면, taskA는 semaphoreX를 가지고 오고, N++ 코드를 수행한다. 그리고 semaphoreY를 반환한다

❺ taskB는 semaphoreY를 얻어오고, printf()를 통하여 N(=2)을 출력한다.

이와 같이 두 task가 서로 협력하며 2000번의 루프를 돌게 되는 것이다.

Reentrancy

함수 안에서 선언된 변수나 함수간에 전달되는 아규먼트, 리턴 값들은 stack에 위치하는 반면, 함수 밖에서 선언한 전역변수나 함수 안에서 “static”이라고 선언한 변수들은 stack이 아닌 다른 메모리 공간(data/bss section)에 위치하게 된다. 앞에서도 설명하였듯이 stack은 각 task 마다 따로 가지고 있기 때문에, 어느 한 task가 사용하는 지역변수를 다른 task가 액세스하지 못한다. 하지만 전역/static 변수들은 공유자원의 성격을 가지고 있어서 모든

task들이 공유하게 된다.

어떤 함수가 **Reentrant**하다는 말은 코드가 재진입이 가능하다는 것이다. 즉, task1에서 reentrant한 functionA를 수행하다가 도중에 Context Switch가 일어나 task2로 제어권이 넘어가고, 이 task2에서 이 functionA를 호출해도 이 함수가 제대로 동작한다는 것이다.

[예] 다음은 reentrant하지 않은 함수에 대한 오동작의 예이다.

```
/* Non-reentrant function */
int  Temp;

void swap(int *x, int *y)
{
    Temp   = *x;
    *x     = *y;
    *y     = Temp;
}
```

```
/* taskA: Low priority */
void taskA(void)
{
    int    x = 1;
    int    y = 2;
    for (;;)
    {
        swap(&x, &y);
        sleep(1);        /* WAITING state during 1 secs */
    }
}

/* taskB: High priority */
void taskB(void)
{
    int    Z = 3;
    int    t = 4;
    for (;;)
    {
        swap(&z, &t);
        sleep(1);        /* WAITING state during 1 secs */
    }
}
```

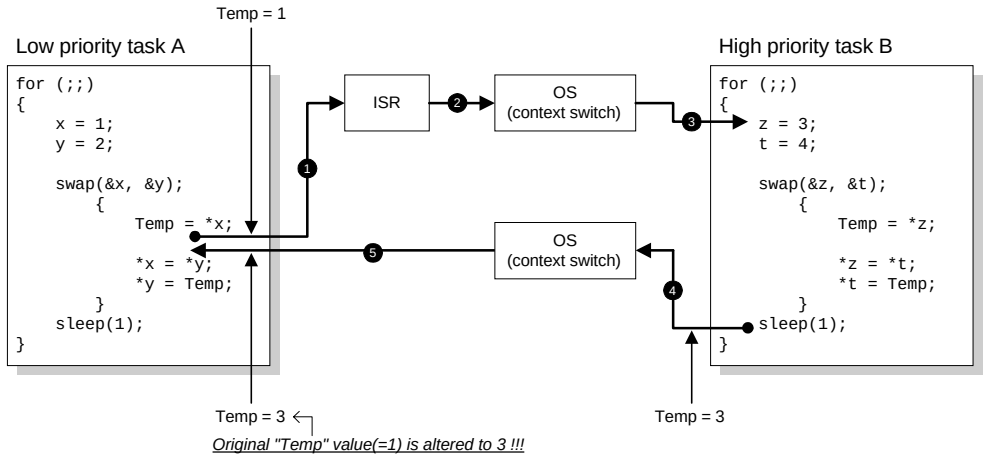

Figure : Non-reentrant function example

❶ taskA가 swap() 함수를 호출하고, 이 함수의 Temp 변수에는 1의 값이 저장되어 있는 상태에서 인터럽트가 발생한다.
❷ ISR에서 인터럽트 처리를 하고, Scheduler/Context Switch를 호출한다.
❸ ISR 종료와 더불어 Context Switch가 수행되고, taskB가 동작(RUNNING상태가 됨)한다. taskB에서 swap()을 호출하고, 이에 의해서 Temp 변수에는 3의 값이 들어간다.
❹ sleep(1)에 의해서 다시 Context Switch가 일어난다.
❺ taskA가 RUNNING상태가 되고, 중단되었던 코드를 수행한다. 이 시점에서 이전에 taskA가 저장했던 Temp 값 1이 아닌, taskB에 의해서 수정된 3이라는 값이 Temp에 저장되는 문제가 발생한다.

예제와 같은 문제를 해결하기 위해서는 다음과 같은 방법들을 사용할 수 있다.
- swap()에서 사용하는 Temp 변수를 지역변수로 선언한다.(지역변수는 stack에 저장되고, stack은 task마다 독립적이므로)
- swap() 함수 안의 코드를 실행하는 동안에는 Scheduling 또는 Interrupt를 금지시킨다.
- Semaphore를 사용한다.

Priority Inversion & Priority Inheritance

Priority Inversion은 높은 priority의 task가 낮은 priority task의 수행이 끝날 때까지 (indefinite period 동안) 기다리는 상황을 일컫는다.
[예] 아래 그림은 task1, 2, 3이 각각 high, medium, low priority를 가진다고 가정하였을 때, Priority Inversion이 발생할 수 있는 예이다.
❶ task3가 공유자원을 액세스하기 위해 binary semaphore를 가지고(take/own) 수행된다.
❷ Scheduler에 의해서 task1이 수행된다.
❸ task1은 task3가 먼저 가져버린 Semaphore를 얻으려(take) 하고, task3가 그 Semaphore를 반환할 때까지(give), WAITING상태가 된다.
❹ Scheduler에 의해서 task3가 수행된다.
❺ Scheduler에 의해서 task2가 수행된다. 이 시점에서 보면, task3의 priority가 task2보다 높음에도 불구하고 task2가 먼저 수행되는 Priority Inversion 문제가 발생한다.
❻ task2의 수행이 종료되면, 다시 task3가 수행된다.
❼ task3가 Semaphore를 반환한다.
❽ 비로소 task1이 수행된다.

Priority Inheritance는 Priority Inversion을 해결 할 수 있는 방법 중에 한가지로, 높은 priority의 task가 WAITING상태인 동안, 그 task를 기다리도록 만든 task의 priority를 높은 task priority 레벨로 올리는 방법이다.
[예]
❶ task3가 공유자원을 액세스하기 위해 binary semaphore를 가졌다.
❷ Scheduler에 의해서 task1이 수행된다.
❸ task1은 task3가 먼저 가져버린 Semaphore를 얻으려 하고, task3가 그 Semaphore를 놓을 때까지, WAITING상태가 된다.

❹ Scheduler에 의해서 task3가 수행되는데, task3의 priority를 task1이 WAITING 상태인 동안, task1의 레벨로 높인다(Priority Inheritance). 이제는 task3의 priority가 task2보다 높기 때문에 preemption 없이 수행된다.
❺ task3가 Semaphore를 반환한다.
❻ task1이 Semaphore를 얻어서 수행됨과 동시에, task3의 priority가 다시 예전의 값으로 돌아간다.
❼ task1이 수행된다.
❽ task2가 수행된다.

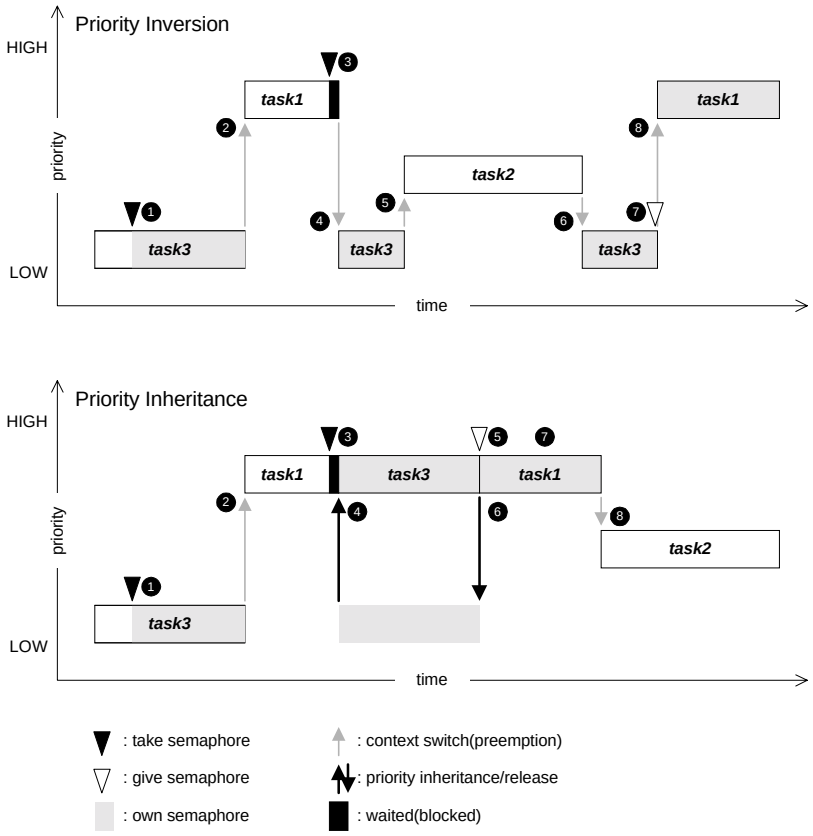

Figure : Priority Inversion & Priority Inheritance

(Inter)Task Communication: Message Queues, Message Mailboxes

task간에 통신하는 방법은 공유자원의 성격을 가진 전역변수를 쓰는 방법(linked list, circular queue등)과 task간의 message passing 방법이 있다. 전역변수를 사용하는 경우는 Mutual Exclusion을 보장해 주어야 하며, 이는 앞에서 설명하였다.

Message passing에는 Message Queue를 사용하는 방식과 Message Mailbox를 사용하는 방법이 있다. 기본적으로, kernel이 관리하는 Queue/Mailbox라는 메모리 영역이 있어서, 메시지를 보내는 측(task나 ISR)에서 kernel의 시스템 콜을 사용하여 이 Queue/Mailbox에 보낼 메시지를 쓰고, 받는 측(task)에서 kernel 시스템 콜을 사용하여 메시지를 받는 구조이다. 어느 task가 메시지 수신을 하려 했는데, Queue/Mailbox에 메시지가 없을 경우에는

WAITING상태가 된다. (메시지 수신 시스템 콜에 의해서 WAITING상태가 될 수 있기 때문에 ISR에서는 메시지를 수신할 수 없다)

Message Queue와 Message Mailbox의 차이는 kernel이 복수개의 메시지를 저장하고 있을 것인지(Queue), 하나의 메시지만을 가지고 있을 것인지(Mailbox: 보통 하나의 메시지를 저장하는 공간은 word 크기로 함)로 구분된다.

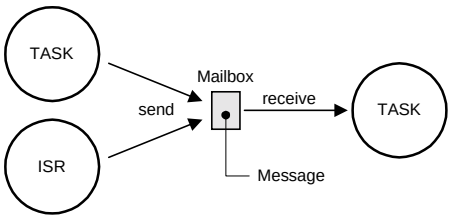

Figure : Message Queue

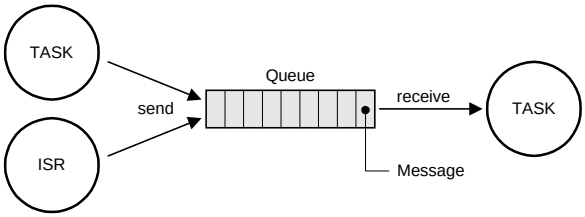

Figure : Message Mailbox

세마포어와 마찬가지로, 복수개의 task가 메시지를 기다리는 WAITING상태에 있을 경우에는, priority base로 메시지를 수신할 수도 있고, FIFO base로 수신할 수도 있다.

위에서 설명한 두 가지 이외에 UNIX에서 사용되는 pipe(file descriptor를 이용한 IPC(InterProcessCommunication))를 지원하는 상용 RTOS도 있다.

Interrupts

인터럽트는 하드웨어 mechanism으로서 CPU에게 비동기적 사건(asynchronous events)을 알리는데 사용된다. 인터럽트 발생 후, ISR이 수행 및 종료되는 mechanism은 CPU 마다 다르게 되어 있다.

PowerPC의 경우, 외부 인터럽트가 발생하면 CPU는 0x500번지로 점프하여 그 곳에 있는 instruction - 물론 사용자가 그 메모리 영역에 적당한 instruction을 넣어 주어야 함 - 을 수행한다. 인터럽트 발생과 동시에 proessor에 의해서 자동적으로 LR(Link Register), MSR(Machine Status Register) 값이 각각 SRR0, SRR1에 저장되고, "rfi"라는 명령에 의해서 ISR이 종료됨과 동시에 SRR0, SRR1의 값이 다시 LR, MSR로 복귀된다.

인터럽트가 종료되는 시점에서 Non-preemptive kernel과 Preemptive kernel 동작 mechanism이 달라진다.

- Non-preemptive kernel은 ISR의 종료되면, ISR전에 수행하던 task(interrupted task)가 다시 수행되는 반면,
- Preemptive kernel의 경우, ISR의 종료 시점에서, Scheduling이 일어나고, 이에 의해서

ISR전에 수행되던 task보다 높은 priority의 task가 READY상태에 존재하면, Context Switch가 일어나서 더 높은 priority task가 수행된다. 물론 더 높은 priority task가 없을 경우에는 ISR 수행 전에 실행되던 task가 다시 수행되어진다.

인터럽트의 발생과 이에 의한 ISR 수행 및 종료 과정에서, 다음 3개의 단어(Interrupt Latency/Response/Recovery)를 정의할 수 있다.

- Interrupt Latency: 인터럽트가 발생한 직후부터 ISR의 첫번째 instruction이 실행되기까지의 시간

  Maximum amount of time interrupts are disabled
  + Time to start executing the first instruction in the ISR

- Interrupt Response: 인터럽트가 발생한 직후부터 ISR의 사용자 코드(instruction)가 실행되기까지의 시간

  Interrupt latency
  + Time to save the CPU's context
  + Execution time of the kernel ISR entry function(preemptive kernel에서만 해당)

- Interrupt Recovery: ISR의 사용자 코드가 종료된 시점에서 task의 코드가 수행되기까지(인터럽트가 종료되기까지)의 시간

  Time to determine if a higher priority task is ready(preemptive kernel에서만)
  + Time to restore the CPU's context of the highest priority task
  + Time to execute the return from interrupt instruction

다음 그림은 Non-preemptive와 Preemptive kernel에서의 Interrupt latency, Interrupt Response, 그리고 Interrupt Recovery를 나타낸 것이다.

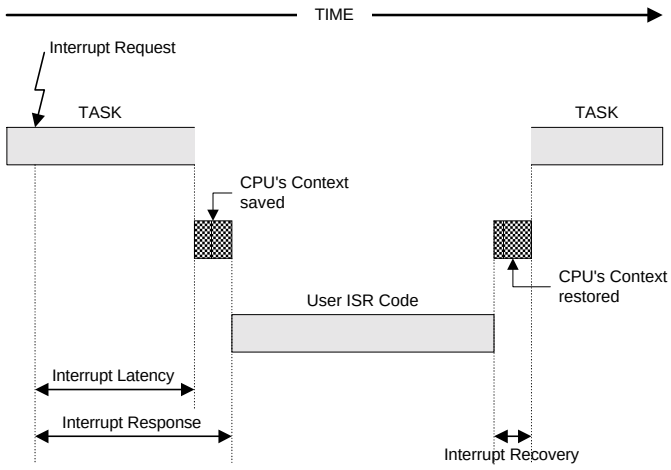

Figure : Interrupt latency, response, and recovery (non-preemptive kernel)

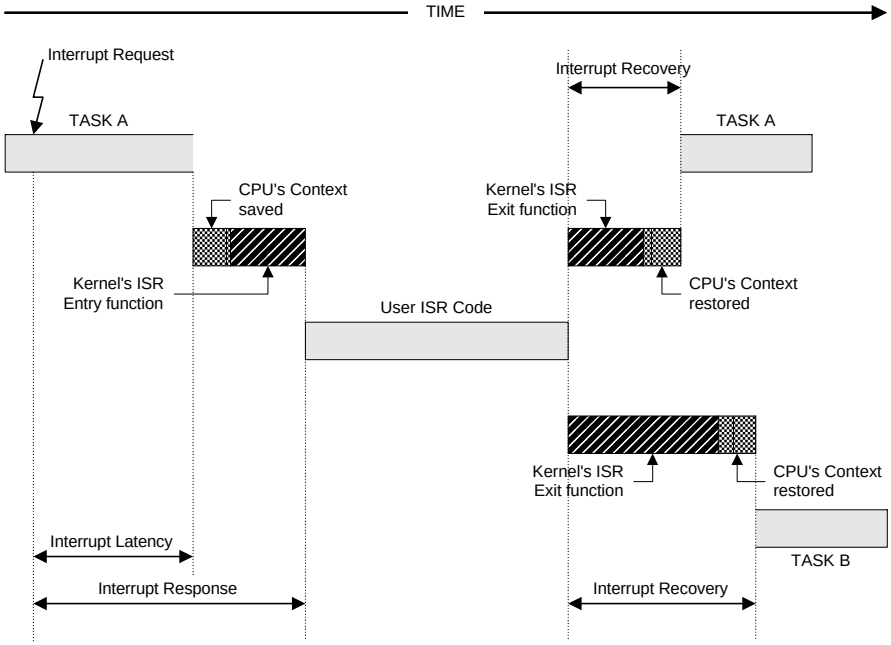

Figure : Interrupt latency, response, and recovery (preemptive kernel)

앞에서 설명하였듯이, Preemptive kernel의 경우 ISR 종료 시점(ISR Exit function)에서 Scheduler에 의해, 이전에 중단되었던 task로 돌아갈 것인지, 아니면 Scheduler에 의해서 READY 상태가 된 task를 수행 할 것인지를 결정하게 된다.

위의 그림에서, taskB가 수행되는 경우에 taskA보다 Kernel's ISR Exit function의 실행 시간이 좀더 소요되는 것으로 표시되어 있다. 이는 이전에 수행되던 taskA에서 taskB로의 Context Switch 수행 시간만큼이 더 걸리기 때문이다.

3. 상용 RTOS

VxWorks나 pSOS등 Embedded system을 위한 상용 RTOS는 지난 70년대부터 산업용 Embedded 시장에서 출발해 Network Protocol, GUI, JAVA등의 다양한 기술을 수용하면서 발전해 왔다. 초기에는 간단한 multitasking kernel만 제공하던 것에서 벗어나 다양한 미들웨어와 개발 환경을 추가하게 되었고 지금은 WebTV와 같은 정보가전과 PDA 같은 개인통신장비에서 필요로 하는 기능에 중점을 두고 있다. 기본적으로 Embedded system의 개발 환경은 Cross Development Environment이다. Visual C++을 이용하여 Windows 응용 프로그램을 개발하는 경우, 코딩과 컴파일, 실행 및 디버깅이 모두 Windows가 탑재된 PC라는 하나의 시스템에서 이루어진다. 이에 반면 Embedded system의 경우, 코딩은 Host(PC나 Workstation)에서 이루어지고, 개발된 코드의 실행은 Target Board에서 이루어진다. 그리고 디버깅은 Host와 Target이 함께 동작하면서 이루어진다. 상용 RTOS의 장점 중에 하나가 편리한 디버깅 환경을 제공한다는 것이다. 아래 그림은 Cross Development Environment의 전형적인 모습을 나타낸 것이다.

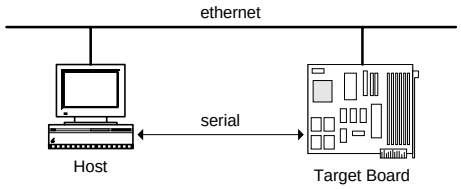

Figure : Development Environments

다음은 Cross Development Environment를 제공하기 위한 기본 요소들이다.

- Ethernet: 보통 host와 target 간에는 Etherent으로 연결되어, host에서 개발된 코드가 Ethernet을 통해서 target으로 다운로드되고, 또한 이 라인을 통해서 host에서 target의 코드를 디버깅한다.
- BSP: Board Support Package의 약자로, target의 Flash memory/ROM에 저장되어 있어, Boot Loader의 기능을 담당한다. 즉, target에 전원이 들어오면, BSP에 의해서 CPU 및 기본적인 디바이스(Ethernet 디바이스, 메모리, Timer등)를 초기화한 후, host로 부터 실행 이미지(OS & Application)를 다운로드 받는 기능을 담당한다. 다운로드 시에는 BOOTP/TFTP/FTP와 같은 표준 프로토콜 또는, RTOS 벤더만의 특별한 프로토콜이 사용될 수 있다.
- Cross Compiler/Linker: target에 탑재된 CPU가 실행할 수 있는 이미지를 생성해 내는 Compiler/Linker가 host에 인스톨되어 있어야 한다. 예를 들어 host는 solaris이고 target CPU는 PowerPC라면, solaris에서 돌아가는 PowerPC Compiler/Linker가 있어야 한다. (이를 Cross Compiler라고 부름)
- Host Shell: host에 설치되는 이 응용프로그램은 사용자가 만든 이미지를 target으로 다운로드 및 언로드(load & unload) 할 수 있는 기능 및 target 상태 변경 및 확인에 대한 포괄적인 기능을 command 레벨에서 지원한다.
- Source Level Debugger: target에 RTOS와 응용 프로그램을 적재 및 동작시킨 후, 오류가 생기는 부분을 찾을 수 있도록 host상에서 C/C++ 소스 레벨에서 응용 프로그램의 흐름을 따라 추적하고 원하는 지점에서 정지하거나 변수 값을 확인 및 수정할 수 있는 GUI 툴이 제공된다.
- Analysis Tool: target에서 동작하고 있는 각 task의 CPU 사용율 및 task 및 ISR간의 관계(어떤 task가 동작하다가 무슨 일(Message Queue/Semaphore등) 때문에 WAITING 상태가 되고, 또 다른 task가 동작하는 등)를 graphical하게 표시해준다.
- Browser: Task, Semaphore, Message queue의 상태 및 메모리 사용 정보 등을 보여주는 기능을 담당한다.
- Target Configuration Tool: OS image 생성 시에 포함되어질 모듈(TCP/IP 위에서 돌아가는 각 응용 프로토콜, File System등)중에 필요한 사양(기능)을 선택적으로 골라서 이미지를 생성할 수 있는 기능을 제공한다.
- Target Simulator: target이 존재하지 않는 환경에서, host의 CPU를 이용하여 target CPU 기능을 emulation하는 기능을 제공한다.

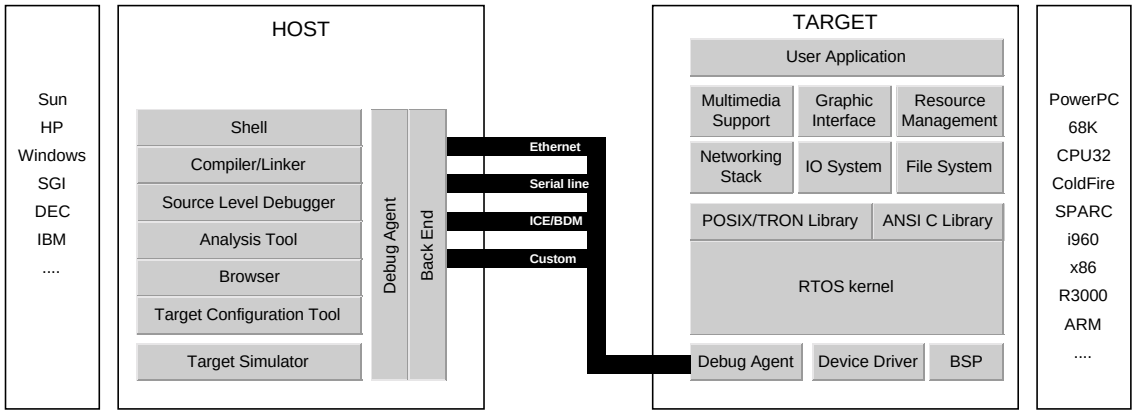

Figure : Development Architecture

RTOS에서 제공하는 함수들의 종류, 모양이 제품마다 각각 차이가 있어서, 하나의 RTOS에서 개발한 응용 프로그램을 다른 RTOS 환경에 이식시키기 위해서는 부가적인 작업이 필요로 하다. 이를 해결하고자, 미국과 일본에서는 각각 표준화 작업을 진행하고 있다. 미국의 경우, POSIX(Portable Operating System Interface) 1003.1b란 것을, 그리고 일본에서는 TRON(The Real-time Operation system Nucleus)이란 인터페이스 표준을 만들어, 응용 프로그램(source code)의 portability를 갖도록 노력하고 있다.

관련 자료

1. Jean J. Labrosse, “MicroC/OS-II The Real-Time Kernel”, 1999
2. Michael Barr, “Programming Embedded Systems in C and C++”, 1999
3. 정용길, 홍성수, “운영 체제 설계(Volume I The XINU Approach - PC Edition)″, 1992
4. 마이크로 소프트웨어, “임베디드 시스템, 그 성공적인 시작을 위하여”, 1999. 9
5. http://www.realtime-info.be
   RTOS에 대한 자료가 굉장히 잘 정리되어 있다. 강추천 !!!
6. http://www.oarcorp.com
   RTEMS(Real-Time Executive for Multiprocessor System) home page
7. http://www.redhat.com/services/ecos
   eCos(Embedded Configurable Operating System) home page