An Introduction to the Real-time OS & Nucleus PLUS Training Guide

리 얼 타 임 코 리 아

REAL TIME KOREA

Accelerated Technology Inc.

Korea Representative

대표전화: 02-522-2267

Index

Preface

최근 multimedia 와 network 분야의 발달로 embedded system 에서도 multimedia 정보를 처리하거나 network 을 접속하는 기능들이 포함되어가면서 시스템이 더욱 복잡해지고 있다. 예전처럼 시스템이 단순했을 때는 순차적인 설계로 가능했지만 시스템에 다양한 기능을 추가하면서 시스템이 복잡해 지면서 기존의 방법과는 다른 새로운 접근 방법이 필요하게 되었다. 처리해야 할 작업이 많아지면서 복잡해진 Embedded system 에서 가장 필요했던 기능은 multitasking 기능이었다. 처리할 여러 개의 작업들을 task 로 나누어 처리해야 했기 때문이다. 따라서 예전에 computer system 에서만 쓰이던 OS 가 embedded system 에서도 필요하게 된 것이다.

그러나 일반 computer system 에서 쓰이는 OS 와는 달리 대부분의 embedded system 은 real-time 이라는 특성을 만족시켜야 했기 때문에 나오게 된 것이 바로 real-time OS 이다. Real-time OS 는 기존의 OS 와 처리하는 작업의 내용은 같지만 real-time 요소를 만족시키기 위해서 구현상에서 몇 가지 차이를 지니고 있다.

여기에서는 real-time OS 의 소개에서부터 real-time system 을 설계하는 단계까지를 설명하도록 한다.

PART 1 은 real-time OS 에서 알아야 할 기본 개념에 대해 설명한다. 1 장은 real-time OS 가 나오게 된 배경, 쓰이는 이유, 표준화의 진행 상황에 대해서 알아본다. 다음에 2 장에서는 real-time OS 를 이해하기 위해서 필요한 개념에 대해서 설명한다. 2 장은 이후에 나오는 내용을 이해하는데 기본이 되는 것들이다.

PART 2 는 real-time OS 를 쓰는 방법에 대해서 알아본다. Real-time OS 에서 제공하는 함수에 대해서 3 장에서 설명될 것이다.

PART 3 는 real-time 시스템을 설계하는 방법에 대해 알아본다. 4 장에서는 embedded System 을 설계할 때 real-time OS 를 선정하는 기준들에 대해 알아본다. 5 장에서는 실제로 real-time OS 를 써서 설계하는 방법에 대해 간략히 설명한다.

Part 1: Real-time OS Basic

1 Introduction to the Real-time OS?

최근의 embedded system 분야에서 real-time OS 를 채택하여 개발된 제품들이 점점 늘어나고 있다. 네트웍이나 멀티미디어가 시스템에 기본으로 자리잡으면서 이제는 많은 시스템에서 채용하고 있는 실정이다.
예전에는 system 이 비교적 간단해서 OS 의 개념을 적용하지 않고 순차적인 program 을 작성해 왔기 때문에 embedded system 에서 OS 가 왜 필요한지, 어떻게 적용해서 어떻게 설계해야 할지 잘 이해가 되지 않을 것이다.
이 장에서는 우선 real-time OS 에 대해 알아보기 전에 일반적인 OS 의 개념에 대해 간략히 알아보고 real-time OS 가 이와 무엇이 다르며 왜 필요한지, 둘 사이에 차이점은 무엇인지 등에 대해 알아보고 마지막으로 표준화 작업에 대해 간략히 소개하기로 하겠다.

1.1 OS 란 무엇인가?

OS 란 Operating System 의 약자로서 우리말로 운영체제라고 번역한다. 우리가 쓰고 있는 모든 컴퓨터에는 OS 가 들어가 있다. PC 에서 흔히 볼 수 있는 DOS, Windows 95, Windows NT, OS/2 들이 모두 OS 들이고 Workstation 이나 그 이상의 컴퓨터에서는 UNIX 나 그의 변종인 OS 들이 수행되고 있다.
그렇다면 OS 는 왜 필요한 것인가? 결론부터 말하면 컴퓨터를 좀 더 쉽게 쓰기 위해서 만들어 졌고 컴퓨터에서 없어서는 안될 핵심 소프트웨어이다.
컴퓨터의 기본 요소로는 CPU, memory, I/O 를 꼽을 수 있다. 우리가 PC 를 쓰는 경우를 생각해보면 언제나 작업하려는 내용들이 memory 에 저장되어 CPU 에서 하나씩 읽어서 수행하며 때에 따라 I/O 에도 반응하게 되는 것이다.
예를 들어서 계산기 프로그램을 수행시켜 계산을 하려한다고 하자. 계산은 CPU 에서 이루어지지만 이에 앞서 CPU 가 처리할 명령들이 메모리에 적재되어 있어야 한다. 또한 결과를 얻어서 출력을 하기 위해서는 프린터로 결과를 보내야 한다. 즉, 여러 가지 자원이 있지만 이를 쓰기 위해서 무언가 이를 관리해주는 프로그램이 필요하다. 이런 자원을 이용하는 여러 개의 프로그램을 작성할 때 각각의 프로그램마다 별도로 자원을 관리하는 기능을 넣는다면 이는 매우 비효율적인 일이 될 것이다.
예를 들어 계산기 프로그램 내부에 load 하는 기능까지 넣을 수도 있겠지만 load 하는 프로

그램은 계산기뿐 아니라 다른 모든 프로그램에서 공통이 되기 때문에 따로 loader 라는 프로그램을 둔다. 이처럼 운영체제는 여러 프로그램에서 공통적인 부분을 묶어서 시스템 프로그램을 작성하면서부터 시작됐다.

또한, 예전에 컴퓨터라는 것은 매우 값비싼 장비였다. 따라서 컴퓨터를 효율적으로 쓰는 것이 매우 중요했는데, 가령 어떤 프로그램이 CPU 와 관련된 작업을 끝내고 I/O(예를 들어 프린터)와 관련된 작업을 시작했을 때 CPU 는 다른 일을 하지 않고 대기 상태에 있었다. 이를 해결하기 위해서 multiprogramming 이란 개념이 도입되었다.

그러나 여러 사람이 컴퓨터를 쓰는 경우에 다른 사람은 쓸 수 없었기 때문에 한번 사용하기 위해 많은 시간을 기다려야 했다. 이를 해결하는 방법은 시간을 나누어서 여러 사람들 모두를 만족시키는 것이었다. 즉, 여러 사람에게 조금씩 시간을 할당해 주었고 한 사람 한 사람은 마치 자신이 혼자 컴퓨터를 쓰는 것처럼 느끼게 했다. 이를 time-sharing 이라고 한다.

이처럼 여러 사람이 동시에(논리적으로) 서로 제 각각 프로그램들을 수행하게 되었고 서로 충돌이 나지 않도록 해야 하는 기능이 필요하게 되었다. 그래서 task 들이 충돌이 나지 않도록 하는 monitor program 이 생겨났고 이것이 점점 발전되어 운영체제의 일부가 되었다.

이와 같은 과정을 거치면서 multiprocessing(multitasking), task scheduling 등의 개념이 OS 에서는 일반화가 되었다.

요즘에 일반적인 OS 가 하는 일을 살펴보면 CPU 에서 수행되는 프로그램들을 관리하기 위한 task management(task scheduling, communication, synchronization 등을 포함) memory 를 관리하기 위한 memory management, I/O 를 관리하기 I/O management, file system management 들이 있다. 이런 작업들은 앞으로 계속 다룰 real-time OS 에서도 공통적으로 하는 일이다. 따라서 일반적인 OS 를 확실히 이해하고 있다면 real-time OS 에 대한 개념을 쉽게 이해할 수 있을 것이다. 일반적인 OS 에 대한 개념을 좀 더 알고 싶다면 [1]을 참고한다.

1.2 Real-time OS 는 왜 나오게 되었는가?

Real-time OS 가 왜 나오게 되었는지는 현재 어디에서 쓰이고 있는지 알면 될 것이다. Real-time OS 가 쓰이고 있는 경우는 크게 2 가지로서 embedded system 과 multiprocessor system 이다.

Multiprocessor system 은 여러 개(multi)의 processor 들이 한 system 에 들어가서 동시에 작업을 수행하는 것이다.(multiprocessing 과는 다른 개념) 이때 여러 개의 processor 들을 관리하기 위해서 기존의 OS 가 아닌 multiprocessor 용 OS 가 필요하다. 최근 들어 그러한 용도로 real-time OS 가 쓰이고 있으나 이 과정에서는 multiprocessor 에 대한 내용은 다루지 않는다.

Embedded system 이라는 것은 일반적인 computer system 과는 달리 특정한 작업만을 하도록 설계되며 초기의 embedded system 은 비교적 단순해서 OS 가 필요 없이 인간이 순차적인 프로그램을 작성해서 수행하도록 했고 중간에 interrupt 가 발생되는 경우에만 그 순차적인 프로그램에서 잠시 벗어나는 정도였다. 그러나 최근 들어 multimedia 정보를 처리해야 하

는 embedded system 이 늘어나면서 그 system 이 해야 할 일들도 많아지고 복잡해 졌기 때문에 순차적인 프로그램 작성이 매우 어렵게 되었다. 따라서 embedded system 에서 OS 의 개념이 필요하게 되었으며 embedded system 의 특성상 real-time 이라는 요소를 만족해야 했다. 따라서 real-time OS 가 embedded system 에 도입된 것이다.
참고적으로 real-time 시스템에 대해 설명하자면 정해진 시간 제약을 해결할 수 있는 시스템을 말한다. 이 말은 주어진 시간 내에 어떠한 일을 반드시 처리해야 한다는 말이다. 즉 단지 빨리 처리해야 하는 것이 아니고 정해진 시간을 넘겨서는 안된다는 것이다. Real-time 에서 가장 대표적인 분야로는 군사용을 들 수 있다. 예를 들어 미사일이 어떤 목표를 추적함에 있어서 어떤 계산이 주어진 시간 내에 이루어지지 않는다면 그 미사일은 그 목표를 격추시킬 수 없는 것이다.
우리가 쓰는 컴퓨터에서는 editor 가 어느때는 좀 빨리 어느때는 좀 늦게 실행된다고 해서 크게 문제가 될 건 없다. 그러나 embedded system 에서 주어진 입력중에는 시간내에 처리하지 못하면 문제가 될 수 있기 때문에 real-time 요소를(보통 soft real-time)만족시켜야 한다.

1.3 Real-time OS 는 일반 OS 와 어떤 점이 다르며 왜 필요한가?

Real-time OS 라고 하면 일반적으로 말하는 OS 와는 막연히 무언가가 다르다고 생각한다. 그러나 Real-time OS 라고 해서 특별히 다른 것은 없다. 일반적인 OS 와 마찬가지로 여러 가지 task 를 동시에(가상적인 의미에서) 수행하는 것이다. 대부분의 Real-time OS 매뉴얼을 살펴보면 다음과 같이 task scheduling, task communication, task synchronization, memory management, interrupt service, I/O driver, timer 등의 기능을 하며 이러한 것은 일반적인 OS 에서도 필요한 기능이다.
말 그대로 real-time 이기 때문에 real-time 을 처리하는 OS 라고 생각할 수 있겠다. 그렇다면 무언가 다른 것이 있을 것이며 또 어떻게 처리해야 real-time 을 만족할 것인가?
여러 가지 차이점이 있겠으나 대체로 효율성, 속도(좀더 엄밀히 말하면 시간 제약), 공평성(fairness)등을 들 수 있겠다.
일반적인 OS 에서는 자원(예를 들어, 메모리, 하드 디스크 등등)을 얼마나 효율적으로 낭비없이 쓸 것인가에 초점이 맞춰져 있다. 그러나 real-time OS 에서는 효율적인 것도 중요하지만 그 속도에 좀 더 신경을 쓴다.
예를 하나 들어보자. 보통 C 프로그램을 작성할 때에 메모리를 할당하기 위해서는 malloc 이라는 함수를 쓰게 된다. 이 함수는 malloc 을 할 수 있는 영역에서 메모리를 찾게 된다. 만약 메모리가 당장 쓸 수 없는 경우에는 기다리는 일도 발생된다. 그러나 real-time OS 에서 메모리를 할당하는 경우에는 malloc 을 쓰지 않고 real-time OS 에서 제공하는 메모리 할당 함수를 쓴다. 이 함수가 malloc 과 차이점이 있다면 정해진 구역(이미 처음에 메모리를 할당하기 위한 메모리 풀을 선언하게 된다)에서 메모리를 할당하려고 시도하고 만약에 당장 할당할 메모리가 없다면 무작정 기다리는 것이 아니라 현재 할당할 수 없음을 알리게 된다. 이는 무

작정 메모리 할당을 기다리다가 정해진 시간 제약을 만족하지 못해서 real-time 으로 실행하지 못하게 됨을 막기 위한 것이다.
이 두가지 구현 방법을 살펴보면 일반 OS 에서는 메모리라는 자원을 좀 더 효율적으로 쓰려는 의도가 있는 것이고 real-time OS 에서는 약간의 메모리 낭비를 감수하고서라도 시간 제약을 맞추려는 의도가 있는 것이다.
또한 UNIX 와 같이 여러명의 사용자가 쓰는 경우에는 각 사용자들이 실행하는 프로그램이 task 로서 수행이 되고 대개의 경우에는 우선순위에 많은 차이를 두지 않는다. 이것은 그 task 들 사이에 공평성을 유지하려 하기 때문이다. 그러나 real-time OS 에서 task 는 대개 우선순위가 차이가 있도록 하며 이때 task 사이의 공평성은 고려하지 않는 것이다.
이상에서 본 것처럼 메모리를 할당하거나 task 를 scheduling 하는 작업은 두가지 OS 에서 모두 필요한 작업이므로 작업 자체는 공통적인 것으로 볼 수 있다.
한편, 구현을 함에 있어서는 원하는 목표가 다르기 때문에 차이가 나타난다.
결론적으로, 원래 real-time system 과 일반 시스템의 차이가 해야할 기능은 같지만 주어진 시간안에 처리할 수 있는가 아닌가라고 할 수 있듯이 OS 에서도 역시 결국 하는 작업은 같지만 어떻게 처리하는 가가 차이라고 할 수 있을 것이다.

1.4 Real-time OS 의 역사 및 표준화 작업

이전의 real-time OS 는 대개 특정 target application 과 architecture platform 을 위한 OS kernel 이었다. 그래서 매우 많은 real-time OS 가 생겼고 각 OS 마다 차이가 생겼다. 최근 들어서 이를 표준화 하려는 작업이 진행 중인데 미국의 POSIX 와 일본의 TRON 이 있다.

1.4.1 POSIX

현재 PC 에서는 대부분 Microsoft 사의 OS(DOS, Windows 95)를 쓰고 있지만 workstation 급 이상의 중대형 컴퓨터에서는 OS 로서 대부분 UNIX 를 쓰고 있다. UNIX 는 AT&T 의 Bell 연구소에서 근무하던 Ken Thompson, Dennis Ritchie 와 몇몇 사람들이 Multics Project 를 이어받아 만든 OS(Multics 를 비꼬는 말)로서 C 로 작성되어 이식성이 매우 높기 때문에 많은 시스템에 적용 되어 왔고 따라서 여러 가지 변종이 나타나게 되었다. 가장 대표적인 시스템은 Berkeley 대학에서 만든 BSD 계열과 AT&T 의 System V 계열이 있다. 이처럼 많은 변종이 나타나면서 UNIX 라는 이름을 갖는 OS 들 사이에서도 점점 차이점이 나타나게 되었고 따라서 표준화하기 위한 일환으로써 POSIX(Portable Operating System Interface (X 로 끝나는 이유는? 대부분의 OS 가 X 로 끝나니까))란 것이 제정되기에 이르렀다. POSIX 의 목표는 응용 프로그램이 source code portability 를 갖도록 하는 것이다.
POSIX 중에서도 real-time OS 에 대한 표준을 제정하기 위한 작업도 진행되어 있는데 이것이 POSIX 1003.4[3]이다. POSIX 는 원래 미국에서 나온 것이기 때문에 미국에서 나오는 대부분의 real-time OS 는 이 POSIX 를 따르거나 따르려는 계획을 가지고 있다. 그러나 아래에서 설명할 TRON project 에 비하면 아직 초기 단계라 할 수 있다. 또한 POSIX 의 시작이 UNIX 를 표준화하는 것이었으므로 원래 real-time OS 가 아니었던 것에 확장을 하는 개념이어서 문제

를 안고 있는 실정이다.

1.4.2 TRON

TRON 은 The Real-time Operating system Nucleus 의 약자로서 1984 년에 일본의 사카무라 박사를 중심으로 시작된 프로젝트이다. TRON 프로젝트는 real-time system 에 대한 전체적인 표준을 제정하려는 노력이다. TRON 에는 ITRON, BTRON, CTRON, MTRON 등 몇가지가 있다. 이 프로젝트에 관심이 있다면 http://tron.um.u-tokyo.ac.jp/TRON/ 을 참고한다.

Micro processor 에 대한 표준은 μTRON 으로서 이를 기준으로 하는 OS 목록은 http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html 에 가면 있는데 살펴보면 제작 회사가 모두 일본으로 대부분 chip maker 이다. 따라서 지원하는 프로세서 또한 자신들의 회사에서 만든 microprocessor 이다. 즉 여기서 알 수 있듯이 이 표준의 단점이라면 일본 내부에서만 쓰이는 표준으로 미국에서는 표준으로 쓰지 않는다는 점이다. 그러나 POSIX 에 비해 먼저 제정되어 이미 많은 제품이 나와 있다는 장점이 있으므로 설계하는 embedded system 에서 일본 chip maker 에서 만든 microprocessor 를 쓰는 경우에는 고려해 볼 필요가 있는 표준이다.

2 Basic Concepts for the Real-time OS

2.1 task 와 multitasking

UNIX 를 써본 적이 있는 사람은 여러 사람이 한꺼번에 login 을 해서 동시에 자신의 프로그램들을 수행할 수 있음을 알 것이다. 굳이 UNIX 를 예로 들지 않더라도 Windows 95 시스템인 경우만 해도 Sound card 를 통해서 음악을 들으며 C compiler 는 compile 을 수행하면서 printer 로 문서를 출력할 수 있음을 알고 있다. 이때 음악 연주 프로그램, compiler program, printer spooler 를 각각의 task 라고 한다. 그리고 이처럼 여러 개의 task 를 동시에(여기에서 말하는 의미는 어떤 기간을 말하는 것으로 어느 한 시점으로 보면 아님, multiprocessor system 과 비교!) 실행할 수 있는 것을 multitasking 이라 한다.

Embedded system 에서도 처리할 작업이 많아지면서 multitasking 을 적용하게 되었으나 앞에서 말한 것과는 차이가 있다. 일반적인 컴퓨터에서는 각 task 들이 대부분 무관한 프로그램들이지만 embedded system 에서의 task 들은 하나의 큰 응용 프로그램을 논리적으로 나눈 것이다. 즉, 어떤 응용 프로그램이 수행되기 위해서는 여러 기능들이 동시에(물리적으로는 아님) 수행되어야 할 필요가 있고 이를 순차적으로 프로그램을 하기 어렵기 때문에 사용자가 프로그램을 쉽게 하기 위해서 도입한 개념이 바로 multitasking 이다. 하나의 응용 프로그램을 논리적으로 나누었기 때문에 기능상 매우 밀접한 관계가 있기 때문에 task 사이에 이루어지는 작업들이 매우 많다.

이처럼 일반 시스템과 embedded system 에서의 multitasking 이 다른 의미를 갖지만 아래에서 설명하는 것은 OS 에서 논의되는 일반적인 개념으로 어떤 시스템에도 적용할 수 있는 것이다.

2.2 context switching

우선, task1 에서 하던 작업이 중단되고 task2 가 수행되려 한다고 가정하자.(이렇게 되는 경우는 매우 많고 3 장의 task transition 에서 더 설명될 것이다.) task1 이 완전히 종료가 된 상태라면 별 문제 없겠지만 중단된 것이기에 task2 가 수행을 마치거나 중단되는 경우에 다시 task1 을 수행하게 된다면 이전에 수행되던 상태를 알아야 할 것이다.(이는 함수에서 함수를 다시 호출하는 경우와 비슷하다.)

따라서 task2 가 수행되기 이전에 task1 의 상태를 저장해 둘 필요가 있다. Computer system 에서 상태라는 것은 Register, 메모리, 하드 디스크 등에 저장된다. 이때 메모리 같은 자원은 task 마다 나누어 쓰면 별 문제가 없을 수 있지만 CPU 의 register 는 공유하는 자원이므로 구역을 나누어 쓸 수 없다. 따라서 상태를 저장해야 하는 가장 중요한 자원은 register 이다.

그래서 task1 이 중단되기 전에 register 값들을 task1 의 stack 에 보관을 하고 task2 가 수행되고 다시 task1 이 수행될 시점이 오면 stack 에 저장되었던 값들을 register 로 옮겨와서 task1 을 다시 수행하게 된다. 이를 context switching(그림 2-1 참고)이라 한다.

Context switching 이 이루어지는 동안은 실제 응용 프로그램에서 원하는 작업은 이루어지지 않기 때문에 overhead 라고 볼 수 있다. 이것은 저장할 register 의 개수에 따라, 쓰는 프로세서 등에 따라 달라진다. Context switching 에 걸리는 시간이 짧으면 짧을수록 좀 더 효율적인 OS 라고 볼 수 있기 때문에 예전에 real-time OS 를 비교할때는(현재도 그러하지만) 이 시간이 얼마나 짧은가가 매우 중요한 척도였지만 프로세서의 성능이 점점 좋아지면서 OS 간에 차이가 줄어들고 있다. 이에 대해서는 4 장에서 다시 얘기하기로 하자.

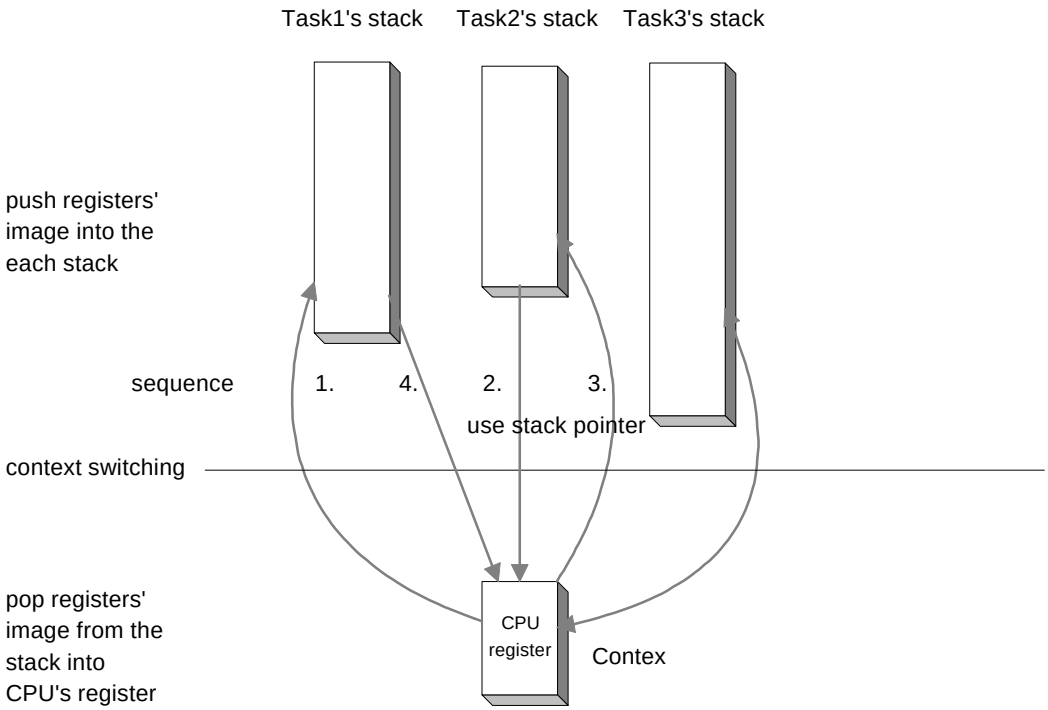


그림 2-1 multitasking and context switching

2.3 Kernel

OS 란 앞에서도 언급했지만 컴퓨터 자원을 효율적이고 쉽게 쓰도록 하기 위한 소프트웨어이다. OS 의 기능 중에서도 핵심이 되는 부분을 kernel 이라고 하고 여기에서는 앞에서 설명한 context switching, task scheduling, memory management 등의 작업을 한다.

2.3.1 Scheduler

multitasking 환경에서 다음에 어떤 task 가 수행될 것인지 결정하는 책임을 맡는다. kernel 이라고 부르는 부분에서도 핵심적인 부분이라고 할 수 있는 것이 바로 scheduler 로서 dispatcher 라고도 한다.

일반 multitasking OS 에는 여러 가지 scheduling 기법에 있으며 real-time OS 에도 이에 맡는 여러 가지 scheduling 기법들이 제안되었다. 그러나 구현상의 문제 등으로 실제로 대부분의 real-time OS 에서 쓰는 scheduling 기법은 priority-based scheduling 이다. 또한 같은 priority

에 대해서는 FIFO 나 round-robin 방법이 적용된다.

2.3.2 Non-Preemptive Kernel

우리말로 비-선점형 kernel 이라고 한다. 쉽게 말해서 kernel 의 능력이 크지 못하다고 할 수 있다. 즉, 어떤 task 가 수행되고 있을 때 kernel 이 중간에서 그 task 의 수행을 중지시키고 다른 task 를 수행시킬 수 있는 능력이 없다. 다른 말로 cooperative multitasking 이라고도 한다. 그래서 preemptive kernel 보다 design 하기 쉽다.

Preemptive 되는 경우는 오직 Interrupt Service Routine(ISR)에 의해서이다. 이때도 역시 ISR 에서 interrupted task 로 제어권을 수동으로 넘겨 줘야 한다.

장점으로는 interrupt latency 가 비교적 짧다는 것이고 또 하나 re-entrant function 을 쓸 수 있다는 것이다. 그러나 단점은 task 가 제어권을 넘기지 않으면 kernel 이 아무 지시도 내릴 수 없기 때문에 task 들이 매우 잘 계획이 되어야 한다는 것이다. 따라서 priority 가 높은 task 도 무한정 기다릴 수 있기 때문에 real-time system 에서는 일반적으로 잘 쓰이지 않는다. 우리가 알고 있는 OS 중에서는 Windows 3.1 이 non-preemptive 에 속한다.(Windows 3.1 은 OS 라고 말할 수 없는 성격이긴 하지만)

2.3.3 Preemptive Kernel

우리말로 선점형 kernel 이라고 한다. 이 kernel 은 어떤 task 를 중단시키고 다른 task 를 수행시킬 능력이 있는 것이다. 따라서 앞의 non-preemptive kernel 보다는 구현하기 복잡하다. 또한 interrupt latency 가 조금 길어진다는 단점이 있다.

그러나 이런 단점에도 불구하고 대부분의 real-time OS 에서 채택하는 이유는 priority 가 높은 task 가 먼저 수행될 수 있다는 점 때문이다. 즉, 가장 priority 가 높은 task 는 가장 먼저 수행될 수 있다는 deterministic 한 성격을 지닌다.

앞의 non-preemptive kernel 에서는 ISR 에서 interrupted task 로 제어권을 넘겨줘야 했지만 이 경우에 ISR 이 interrupted task 보다 높은 priority task 를 호출한다면 ISR 이 끝나고 interrupted task 로 진행되는 것이 아니고 priority task 로 진행이 되고 interrupted task 는 suspend 상태가 되는 것이다. 앞으로 설명할 OS 는 이 preemptive kernel 이다.

우리가 알고 있는 대부분의 OS(DOS, Windows 3.1)을 제외하고 모두 이 preemptive kernel 에 해당한다.

2.4 critical section

다른 task 에 의해서 중단되어서는 안되는 일련의 명령 혹은 코드의 블럭을 말한다. 예를 들어 shared memory 를 접근하는 부분에서는 중단이 되는 경우에는 그 결과를 보장할 수 없다. 이를 해결하기 위해서는 아래에서 설명하는 mutual exclusion 이 보장되어야 한다.

2.5 mutual exclusion

예를 들어서 두개의 task 가 동시에 printer 를 쓰려고 한다고 가정하자. 만약 task1 이 프린트하는 도중에 task2 가 실행되고 task2 가 실행되는 도중에 다시 task1 이 실행된다면 나온 결

과는 task1 이나 task2 모두 원하지 않는 결과일 것이다. 이때 mutual exclusion 이 되어야 할 필요가 있다. 또한 위에서 언급한 shared memory 도 마찬가지이다. 즉 shared resource 를 쓰는 경우에는 mutual exclusion 이 보장되어야 하는 것이다.

이를 해결하기 위해서는 몇가지 방법이 있는데 첫째 가장 쉬운 방법은 interrupt 가 발생하는 것을 막는 것이다. 어떤 task 에서 다른 task 로 제어권이 넘어가는 경우의 한가지로 자신이 message 를 기다리는 상태로 가는 상황을 생각해 볼 수 있는데 이런 것을 shared resource 를 쓰는 부분에서는 제외시킬 수 있다. 이런 상황을 막는다고 하면 남는 것은 round-robin 에 의해서 다른 task 로 제어권이 넘어가는 경우나 interrupt 가 발생하는 경우다. Round-robin 에서 다른 task 로 제어권을 넘기는 것 역시 timer(clock) interrupt 가 발생한 경우이므로 한 마디로 interrupt 를 막으면 된다. 즉, critical section 으로 들어가기 전에 interrupt 를 disable 시키고 빠져 나오면서 interrupt 를 enable 시킨다.

이외에 semaphore 라는 mechanism 이 이용된다.(semaphore 에 대해서는 아래에서 설명)

semaphore 를 얻지 못하면 shared resource 를 얻을 수 없으며 일단 semaphore 를 얻으면 shared resource 를 마음 놓고 쓸 수 있다. 다른 task 를 위해서 shared resource 를 다 쓰면 semaphore 를 풀어야 한다.

2.6 semaphore

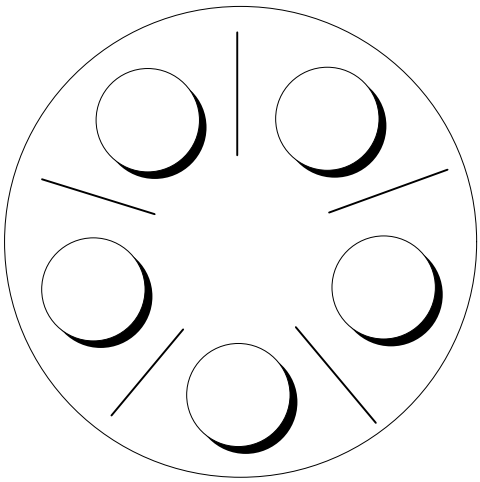
Semaphore 는 Edsger Dijkstra 가 60 년대 중반에 고안한 mechanism 으로서 대부분의 multitasking kernel 에서 쓰이고 있다. Semaphore 는 공유 변수로서 앞에서 설명한 mutual exclusion 을 만들어 공유 resource 를 제어할 수도 있고 task 사이에 synchronize 를 하기 위해서도 이용된다.

Shared resource 를 쓰는 경우를 예로 들어보면 우선 semaphore 를 만들고 shared resource 를 쓰기 직전에 semaphore 를 얻는다. 얻는다는 의미는 변수값이 0 이 아닌 양수인 경우 그 변수값을 1 을 감소시키고 계속 진행하는 것이며 만일 0 이라면 누군가 다른 task 에 의해 값이 양수로 변하기를 기다리고(누군가 shared resource 를 다 쓰면 그 값을 1 증가 시킬 것이다) 양수가 되면 그때 역시 1 을 감소시키면서 계속 진행한다.

Semaphore 가 0 과 1 만이 될 수 있는 것을 binary semaphore 라고 하고 그 이상의 값을 갖는 것을 counting semaphore 라고 한다.

2.7 Deadlock

deadlock 은 시스템이 더 이상 진행될 수 없는 상태에 도달하는 것을 말한다. Dijkstra 가 예로 든 '저녁식사하는 5 명의 철학자'이야기가 있다. 5 명의 철학자가 식탁에 앉아서 식사를 한다. 포크는 모두 5 개가 있다. 음식은 스파게티인데 매우 미끄러워 2 개가 필요하다. 철학자들은 각각 생각을 하다가 먹고 다시 또 생각한다. 이를 프로그램으로 시뮬레이션할 때 계속해서 진행되도록 할 수 있는가? 간단하게 다음처럼 생각해 볼 수 있을 것이다.



```
#define N           5
philosohper(int i)
{
    while (TRUE)
    {
        think();
        take_fork(i);
        task_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

이때 자신의 왼쪽 포크를 잡으면 오른쪽 포크를 찾아서 식사를 하게 한다. 만약 모두 왼쪽 포크를 들고 오른쪽 포크를 찾는다면? (프로그램으로 생각해본다면 철학자 한사람이 왼쪽을 들고 오른쪽을 들기전에 context switch 가 일어나서 다른 사람이 왼쪽을 들고, 다시 context switch 가 일어나고..) 이 사람들은 모두 자신의 오른쪽에 있는 포크가 내려질때까지 기다릴 것이고 그런 상황은 언제까지나 오지 않을 것이다. 이는 서로 남이 소유한 자원을 얻으려고 하기 때문에 일어난다.

이제 이를 조금 바꿔서 자신의 오른쪽 포크가 없다면 왼쪽 포크를 내려놓고 얼마간 기다린 후 다시 왼쪽 포크를 집도록 바꿔 보자. 만약 왼쪽 포크를 동시에 들었다면, 동시에 내려 놓을 것이고 얼마간 지나서 다시 동시에 다시 왼쪽 포크를 들 것이다.

그렇다면 내렸다가 드는 시간을 random 으로 한다면 이는 언젠가 해결될 수 있을 것이다.(확률상으로) 하지만 이는 좋은 해결법이 아니다.

이를 해결하기 위해서 semaphore 를 쓴다.

```
#define N           5
#define LEFT        (i-1) % N
#define RIGHT       (i+1) % N
semaphore mutex = 1;
semaphore s[N];

philosopher (int i)
{
    while (TRUE)
    {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
take_forks(int i)
{
    get_semaphore(mutex);
    state [i] = HUNGRY;
    test(i);
    release_semaphore(mutex);
    get_semaphore(s[i]);
}
put_forks(int i)
{
    get_semaphore(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    release_semaphore(mutex);
}
test(i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
```

```
    {
        state[i] = EATING;
        release_semaphore(s[i]);
    }
}
```

프로그램이 약간 복잡해 진 것은 포크 5 개로 2 명이 먹을 수 있도록 하기 위해서이다. 만약 한번에 한사람만 먹을 수 있도록 한다면 fork 를 얻을 때 semaphore 를 걸고 먹고 나서 fork 를 내려놓으면서 semaphore 를 풀면 된다.

그러나 semaphore 가 만병통치약은 아니다. 2 개의 공유 자원이 있어 각각을 semaphore 로 접근하도록 했다고 하자. 2 개의 공유 자원이 모두 있어야지만 task 가 진행된다. 만약 두 task 가 하나씩 semaphore 를 얻었다면 이 둘은 더이상 진행할 수 없다.

이 deadlock 을 해결하는 방법은 몇가지가 있는데 deadlock 이 아예 일어나지 않게 하는 방법이 있고 deadlock 이 일어났을 때 이를 해소시켜 주는 방법이 있다. 가장 간단한 방법은 deadlock 의 상황이 일어나지 않도록 미연에 방지하는 것인데 이렇게 하려면

1) 모든 자원을 얻고 나서 수행을 하면서 이때 모든 자원을 동일한 순서로 갖는다. 즉 공유 자원이 A, B 가 있다면 A 를 먼저 얻고 B 를 얻도록 하는 것이다. 이렇게 되면 어떤 task 는 A 를 갖고 있고 다른 task 는 B 를 갖고 있는 상황은 발생하지 않을 것이다.

2) 여기에서 자원을 얻는다는 것은 semaphore 를 쓰는 경우가 많다. 이때 semaphore 가 일정시간이 지나면 expire 되게 한다.

3) 하나의 task 가 하나의 semaphore 만 갖도록 한다.

이 방법은 모두 task 를 어떻게 설계하는 가에 달려있다. 즉 OS 가 근본적으로 막는 것이 아니고 사용자가 이런 상황이 일어나지 않도록 설계한다는 것이다.

2.8 Priority Inversion

말 그대로 priority 가 역전되는 현상이다. 3 개의 서로 다른 priority 를 갖는 task 를 가정해보자. 이때 priority(task1) > priority(task2) > priority(task3)라고 가정하자. 현재 task1 과 task3 가 ready 상태라고 가정하자. Task1 이 가장 priority 가 높기 때문에 먼저 수행될 것이다. task1 과 task3 가 하나의 semaphore 를 쓴다고 가정하자. 만약 task3 가 semaphore 를 갖고 있으면 priority 가 높은데도 불구하고 task3 가 수행이 될 것이다. 만약 task3 가 곧 semaphore 를 release 해서 task1 이 수행된다면 이 상황은 큰 문제가 안될수도 있겠지만 만약 task3 가 수행도중에 task2 에 의해 제어권을 뺏기는 경우 task2 가 task1 보다 priority 가 낮은데도 불구하고 먼저 수행된다. 이를 priority inversion 이라고 한다. OS 가 이것을 막지 못한다면 software design 시에 이런 상황이 발생하지 않도록 잘 설계해야한다. 보통 이와같은 상황을 막기 위해서는 같은 semaphore 를 쓰는 task 는 같은 priority 를 주는 등의 방법을 쓴다.

2.9 task communication

task 간에 communication 하는 방법으로는 global data 를 쓰는 것과 message passing 의 2 가지 방법이 있다. Global variable 을 쓰는 경우에 exclusive access 를 해야하는데 ISR 이 포함된 interrupt 를 disable 해야하며 task 사이에서는 interrupt 를 disable 하는 방법 외에 semaphore 를 쓸 수도 있다. (앞에서 설명)

Message passing 의 방법으로는 mailbox, queue, pipe 등이 있다. 이 세가지는 message 의 size 에 따라 나눌 수 있다. Nucleus PLUS 를 예로 들면 다음과 같다.

	Size	number
mailbox	4 X 32 bit	1
queue	m X 32bit	n
pipe	m X 8bit	n

위의 3 개의 mechanism 중에 어느것을 쓸 것인가는 보낼 message 의 성격에 따라 결정해야 할 것이다.

그러나 위의 정의는 Nucleus PLUS 를 따른 것으로 어떤 OS 에서는 mailbox 하나만 있으며 여기에서의 mailbox 는 보낼 수 있는 양이 1 보다 크다. 따라서 자신이 쓰는 시스템에서 제공되는 함수를 잘 읽어봐야 할 것이다.

또한, OS 에 마다 구현되는 방법이 약간씩 다르다. 어떤 OS 에서는 task 하나에 하나의 message queue 가 있어서 task1 에서 task2 로 message 를 보내면 task2 의 message queue 로 전달이 되는 반면, 어떤 OS 에서는 특정 task 에 관련이 없이 message queue 가 존재해서 여러개의 task 가 하나의 message queue 를 기다릴 수도 있다. 이런 구조에서 queue 를 task 수만큼 만들면 task 하나에 하나의 message queue 가 있는 구조가 될 수는 있다. 단, 후자의 경우에 message 를 받는 task 가 특정 message queue 에서 자신의 메모리 영역으로 message 를 복사를 하게 되므로 시간이 오래 걸릴 수 있다. 이때에는 메모리 pool 을 만들어서 message queue 로 pointer 를 넘겨서 시간 및 메모리를 절약할 수 있다. 이에 대해서는 뒤에서 다시 설명하기로 하겠다.

2.10 task synchronization

task 사이에서 순서를 잡을 때 쓰인다. 보통 semaphore 를 쓸 수도 있고 event flag 나 signal 을 쓸 수도 있다. 예를 들어 어떤 task 가 I/O 를 초기화 한후 semaphore 를 기다린다. ISR 에서는 I/O operation 을 끝내고 semaphore 를 통해 알려주므로써 그 task 가 다시 resume 될 수 있도록 하는 역할을 한다. semaphore 에 대해서는 앞의 내용을 참고한다.

Event flag 는 Event 가 여러개 발생했을 때의 synchronize 를 위해서 필요하다. Nucleus 에서는 Event group 이라는 용어로 쓰인다. 이 group 에서는 한 bit 마다 특정 event 에 대응해서 어떤 event 가 발생되면 그 bit(event flag)가 set 되며 이때 그 event 를 기다리는 task 가

ready 상태가 된다.
signal 이란 event flag 와 비슷하지만 차이가 있다. signal 이 발생되면 그 task 는 interrupt 되고 특정 signal handling routine 이 호출된다. 이는 마치 interrupt service routine 의 경우와 비슷하다.

2.11 interrupt service

interrupt 는 asynchronous event 를 CPU 에 알리는 방법이다. Interrupt 는 외부에서 들어오는 중요한 신호로서 시간에 민감한 경우가 있기 때문에 interrupt latency 가 짧은 것이 좋다. 또한 ISR 자체도 짧은 것이 좋은데 그 이유는 ISR 자체가 길어지면 interrupt nesting 이 되기 쉽기 때문이다. 따라서 ISR 에서는 보통 그에 상응하는 task 수준의 service routine 을 부르고 (HISR)끝나도록 구성을 한다. HISR 에서는 마치 task 처럼 존재해서 수행이 된다.

2.12 soft real-time vs. hard real-time

real-time system 중에는 어떤 시간안에 반드시 처리해야 하며 그 시간이 지난 후의 결과 값은 정확해도 소용이 없는 경우가 있다. 이런 시스템을 hard real-time system 이라고 한다. 이에 반해서 어떤 시간안에 처리하면 좋지만 그렇지 못한 경우에 그 시간에서 약간 경과한 후의 값도 인정하는 경우를 soft real-time system 이라고 한다. 여기에서 설명되는 것은 soft real-time system 이다.

2.13 reentrancy

reentrant code 란 말 그대로 코드가 다시 진입할 수 있음을 나타낸다. 어떤 함수가 reentrant 라는 것은 task1 에서 function1 함수를 수행하다가 도중에 task2 로 제어권이 넘어가고 이 함수를 다시 호출해도 function1 이 제대로 동작한다는 것이다.
어떤 code 를 reentrant 하도록 하는 방법은 전역변수를 쓰지 않는 것이다. 전역변수라는 것은 shared resource 로 볼 수 있으므로 mutual exclusive 하도록 만들어 주지 못하면 문제가 생기는 것이다.
그렇지 않은 code(function)는 공유하지 않거나 꼭 공유해야 하는 경우에는 semaphore 를 쓴다거나 동일 priority 를 갖는 task 사이에서 round-robin 을 하지 않아야 한다.

Part 2: Using Real-time OS

3 Real-time OS Functions

Real-time OS 의 기능은 크게 task management, task synchronization, task communication, memory management, timer management, interrupt management 6 개로 나눌 수 있다.

3.1 Task Management(Scheduling)

Task 는 간단히 말하면 그림 3-1 에서 보이는 상태를 갖는다.

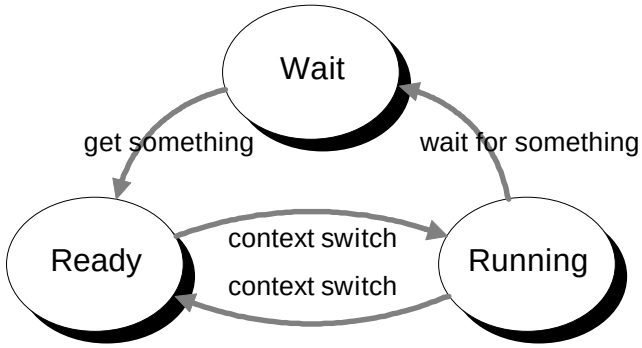


그림 3-1 Simple transition diagram

이 그림은 상태를 상당히 간략히 설명한 것이지만 가장 핵심적인 3 가지 state 를 설명하고 있다. task 는 일단 만들어지면서 ready 상태로 간다. 만약 running 중인 task 가 없으며 ready 상태에 있는 task 중에서 우선순위가 가장 높다면 이 task 는 running 상태로 진행된다. 중간에 다른 task 와 synchronize 를 하거나 공유 자원을 얻는 등의 작업을 하려면 wait state 로 가게된다. Wait state 에서 공유 자원을 얻게 되면 다시 ready 상태로 가서 running 되기를 기다린다.

실제 system 에서는 대개 다음의 그림 3-2 와 비슷한 상태 전이도를 따라 움직인다.

큰 차이점은 interrupt 와 suspend state 를 넣었다는 것인데 suspend state 는 wait state 와 마찬가지로 수행이 멈춰졌다는 것은 같지만 주체가 누구냐가 다르다. 즉, suspend state 는 다른 task 에 의해 이동되는 것이고 wait state 는 자신 스스로 기다린다는 점이 다르다.

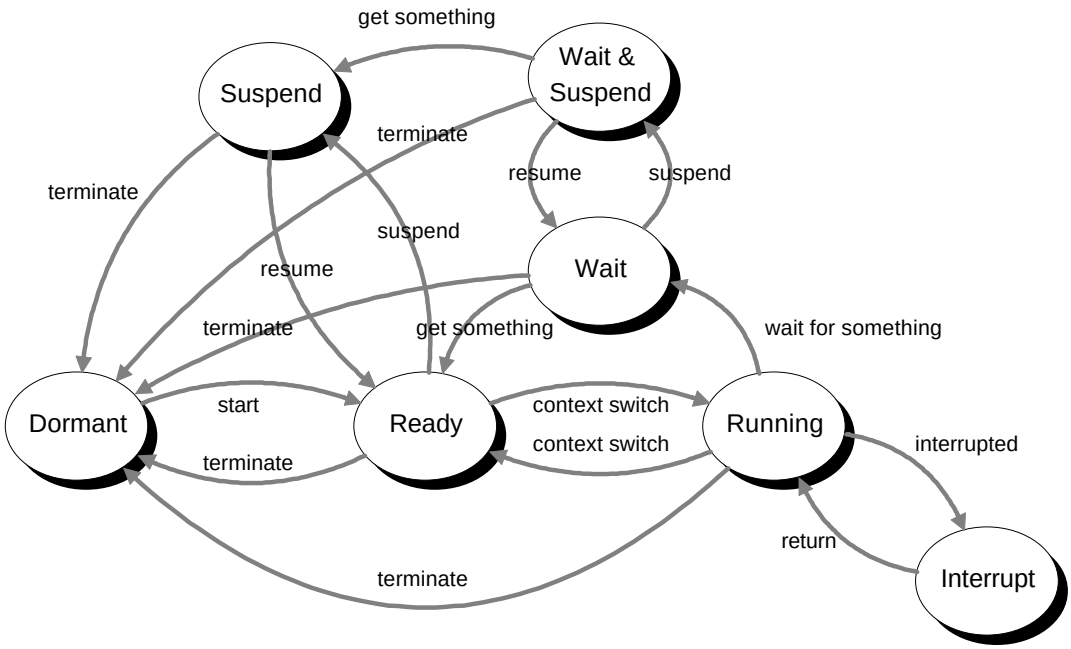


그림 3-2 Real trasition diagram

3.1.1 Create task

task 를 수행시키려면 무엇보다도 먼저 task 를 만들어야 한다. task 를 만들때 필요한 중요 parameter 는 stack, priority, time slice, preempt 등이 있다.

task 는 우리가 일반적으로 생각하는 하나의 프로그램이다. 어떤 프로그램이 수행될 때 함수를 호출할 것이며 그 과정에서 stack 이 쓰이게 된다. 또한 함수 내에서 쓰이는 변수들도 역시 stack 에 보관된다. 따라서 각각의 task 는 고유의 stack 이 필요하다. 이 stack 은 다른 task 와는 겹치지 않도록 되어 있다. 따라서 stack 의 크기를 무한정 크게 할 수 없으므로 필요한 만큼만 적당히 잡아야 한다. 그러기 위해서는 처음에 적당히 계산을 해보고 시스템을 실행시켜가면서 실제로 어느정도 쓰는지 검사를 해보면서 적당한 값을 잡아야 한다.

Priority 는 앞에서 설명한 것처럼 task 의 우선 순위를 주는 것이다. 우선순위를 줄때 명백히 높은 것(Interrupt 관련)이나 낮은 것(MMI) 것은 잡기쉽지만 나머지는 시스템을 실행시켜보면서 정해야 한다. 이에 대해서는 5 장에서 간략히 설명한다.

Time slice 는 같은 priority 인 task 사이에서 round-robin 을 하는 경우에 쓴다.

preempt 는 preemptive 될 것인지 아닌지 결정하는 것인데 보통 preemptive 되도록 한다.

아래에는 그 task 를 만드는 예를 들었다. 우선 메모리 pool 에서 그 task 가 쓸 stack 영역을 확보한 후에 task 를 create 한다. create 는 동적으로도 할 수 있지만 보통은 시스템 시작에 create 한다. Nucleus 의 경우엔 Application_Initialize 에서 하게된다.

예)

```
NU_TASK          Task_0;
STATUS           status;
void       task_0(UNSIGNED argc, VOID *argv);

NU_Create_Memory_Pool(&System_Memory, "SYSTEM", first_available_memory,
                              20000, 0, NU_FIFO);
NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
status = NU_Create_Task(&Task_0, task_0, 0, NU_NULL, pointer, 1000, 1, 20, NU_PREEMPT,
                              NU_START);
void task_1 (UNSIGNED argc, VOID *argv)
{
          /* code */
}
```

3.1.2 Delete task

더 이상 task **를 실행시키지 않게 하기 위해서 쓴다. 앞으로 다시는 쓸 일이 없는** task **라고 생각될 때** delete **한다.**

예)

```
NU_TASK          Task;
STATUS           status;

status = NU_Delete_Task(&Task);
```

3.1.3 Resume task

suspend **되었던** task **를** resume **시키는 것이다.**

예)

```
NU_TASK          Task;
STATUS           status;

status = NU_Resume_Task(&Task);
```

3.1.4 Suspend task

resume task **의 반대로 지정한** task **를 무조건** suspend **시킨다. 만약** suspend **된 상태에 있는 것을 또 한번** suspend **시킨 경우에는** resume **도** 2 **번해줘야 한다.**

예)

```
NU_TASK         Task;
STATUS          status;

status = NU_Suspend_Task(&Task);
```

3.1.5 Terminate task

지정한 task 의 수행을 종료시킨다. delete 와 다른 점은 reset 이 되면 다시 수행될 수 있다는 점이다.

예)

```
NU_TASK         Task;
STATUS          status;

status = NU_Terminate_Task(&Task);
```

3.1.6 Reset task

종료된 task 를 다시 수행할 수 있도록 해준다. 그러나 resume 을 해야 실제로 시작된다.

예)

```
NU_TASK         Task;
STATUS          status;

status = NU_Reset_Task(&Task, 0, NULL);
```

3.1.7 Sleep

sleep 을 호출한 현재 함수가 timer 의 tick 만큼 suspend 되는 것이다. 이때 1 tick 의 값은 Timer Interrupt 가 발생되는 주기가 된다.

예)

```
NU_Sleep(20);
```

3.1.8 Change preemption

preemption 될 것인지 아닌지를 바꾼다. preemption 이 되지 않게 하는 경우는 주의해서 처리해야 한다.

예)

```
OPTION          old_preempt;

old_preempt = NU_Change_Preemption(NU_NO_PREEMPT);
NU_Change_Preemption(old_preempt);
```

3.1.9 Change priority

지정된 task 의 priority 를 바꾼다.

예)

```
NU_TASK         Task;
OPTION          old_priority;

NU_Change_Priority(&Task, old_priority);
```

3.1.10 Change time slice

지정된 task 의 time slice 값을 바꾼다. 0 이면 time slicing 하지 않는다는 것이다.

예)

```
NU_TASK         Task;
UNSIGNED        old_time_slice;

NU_Change_Time_Slice(&Task, old_time_slice);
```

3.1.11 Relinquish

같은 priority 에 있는 task 를 모두 수행시킬 기회를 준다. 그 task 들이 끝나야 다시 차례가 온다.

예)

```
NU_Relinquish();
```

3.1.12 Monitor task status

- Stack Check

현재 task 에서 얼만큼의 stack 이 남아있는지(따라서 얼만큼 썼는지도)를 알려준다. 만약 현재 task 의 context 양보다 stack 이 조금 남아 있다면 overflow 가 될 것이다. 따라서 이 함수

를 쓰면서 stack 이 어느 정도 필요한지 조절할 필요가 있다.

예)

```
UNSIGNED        remaining;

remaining = NU_Check_Stack();
```

- Current Active Task Pointer

현재 active 인 task 를 알려준다.

예)

```
NU_TASK         *task_ptr;

task_ptr = NU_Current_Task_Pointer();
```

- Established Tasks

현재 establish 되어있는 task 의 개수를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
UNSIGNED        total_tasks;

total_tasks = NU_Established_Tasks();
```

- Task Pointer

현재 establish 되어있는 task 의 pointer 와 숫자를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
NU_TASK         *Pointer_Array[20];
UNSIGNED        number;

number = NU_Task_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- Task Information

지정한 task 에 대해서 다양한 값을 보여준다. task 의 이름, 상태, priority, stack 상태 등을 보여준다.

예)

```
NU_TASK         Task;
CHAR            task_name[8];
DATA_ELEMENT    task_status;
UNSIGNED        scheduled_count;
OPTION          priority;
OPTION          preempt;
UNSIGNED        time_slice;
VOID            *stack_base;
UNSIGNED        stack_size;
UNSIGNED        minimum_stack;
STATUS          status;

status = NU_Task_Information(&Task, task_name, &task_status, &scheduled_count,
                             &priority, &preempt, &time_slice, &stack_base,
                             &stack_size, &minimum_stack);
```

3.2 Semaphore (Task Synchronization)

synchronization mechanism 으로는 semaphore, event group, signal 3 가지가 있다.
semaphore 에 대한 설명은 2 장을 참고한다.

3.2.1 Create Semaphore

앞에서 semaphore 의 기본 개념에 대해서는 설명을 하였다. task 와 마찬가지로 쓰기 위해서는 역시 제일 먼저 create 을 해야한다. task 처럼 복잡하게 stack 을 잡는 것 따위는 필요없다. semaphore 를 만들때 설정해 줄 것은 초기 값이다. Binary semaphore 는 0 과 1 밖에 없지만 count semaphore 는 그 이상의 숫자가 가능하다.
또 한가지 semaphore 로 인해 wait 하게 될때 기다리는 것을 FIFO, PRIORITY 등으로 정할 수 있다.

예)

```
NU_SEMAPHORE    Semaphore;
STATUS          status;

status = NU_Create_Semaphore(&Semaphore, “Sema 1”, 1, NU_PRIORITY);
```

3.2.2 Delete

semaphore 를 없앤다 suspend 된 경우에는 이 semaphore 를 기다리고 있던 것은 resume 되

며 error signal 이 뜬다.

예)

```
NU_SEMAPHORE Semaphore;
STATUS                status;

status = NU_Delete_Semaphore(&Semaphore);
```

3.2.3 Obtain

mutual exclusion 을 시작할 때(critical section 에 들어갈 때) 쓴다. semaphore 변수를 1 감소시킨다. 만약 0 이라면 양수가 될 때까지 기다린다. 2 번째 paramter 를 NU_NO_SUSPEND 로 하면 전혀 기다리지 않는 것이고, 숫자를 넣으면 그 숫자 tick 만큼 기다린다.

예)

```
NU_SEMAPHORE Semaphore;
STATUS                status;

status = NU_Obtain_Semaphore(&Semaphore, 20);
```

3.2.4 Release

obtain 했던 semaphore 를 release 하는 것으로 semaphore 변수를 1 증가 시킨다.

예)

```
NU_SEMAPHORE Semaphore;
STATUS                status;

status = NU_Release_Semaphore(&Semaphore);
```

3.2.5 Reset

Semaphore 변수 값을 지정한 값으로 reset 시킨다.

예)

```
NU_SEMAPHORE Semaphore;
STATUS                status;
status = NU_Reset_Semaphore(&Semaphore, 1);
```

3.2.6 Monitor semaphore status

- NU_Etsablished_Semaphores

현재 establish **되어있는** semaphore **의 개수를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
UNSIGNED        total_semaphores;

total_semaphores = NU_Established_Semaphores();
```

- NU_ Semaphore_Pointers

현재 establish **되어있는** semaphore **의** pointer **와 숫자를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
NU_SEMAPHORE*Pointer_Array[20];
UNSIGNED                number;

number = NU_Semaphore_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- NU_Semaphore_Information

지정한 semaphore **에 대해서 다양한 값을 보여준다**. semaphore **의 이름**, **값**, **기다리는** task **들**, **첫째로 기다리는** task **등을 보여준다**.

예)

```
NU_SEMAPHORESemaphore;
CHAR                    semaphore_name[8];
UNSIGNED                current_count;
OPTION                  suspend_type;
UNSIGNED                tasks_suspended;
NU_TASK                 *first_task;
STATUS                  status;

status = NU_Semaphore_Information(&Semaphore, semaphore_name, &current_count,
                                  &suspend_type, &tasks_suspended, &first_task);
```

3.3 Event_Group (Task Synchronization)

Event_group 은 32 개의 event_flag 로 이루어져 있다. 각 event_flag 는 logical AND/OR 를 써서 set 이나 clear 할 수 있다. 또한 자동으로 reset 될 수 있다.

3.3.1 Create Event Group

역시 event group 을 쓰기 위해서는 먼저 create 를 해야 한다. 초기에는 0 으로 되어 있다.

예)

```
NU_EVENT_GROUP      Events;
STATUS              status;

status = NU_Create_Event_Group(&Events, “any name”);
```

3.3.2 Delete

Event group 을 없앤다. suspend 된 경우에는 이 event group 을 기다리고 있던 것은 resume 되며 error signal 이 뜬다.

예)

```
NU_EVENT_GROUP      Events;
STATUS              status;

status = NU_Delete_Event_Group(&Events);
```

3.3.3 Set Events

지정한 event group 에서 지정한 event flag 를 set 한다. 여기서 set 한 것을 기다리던 task 는 resume 된다. 각 event flag 에 대해서 OR 와 AND 두가지가 있다.
OR 는 현재 event flag 와 지정하는 event flag 를 or 하라는 것이고 AND 는 현재 event flag 와 지정하는 event flag 를 and 하라는 것이다.

예)

```
NU_EVENT_GROUP      Events;
STATUS              status;

status = NU_Set_Events(&Group, 0x00000086, NU_OR);
```

3.3.4 Retrieve Events

지정한 event group 에서 지정한 event flag 조합을 retrieve 한다. 이 조합이 맞으면 종료된다. 4 가지 operation 이 있다. NU_AND 와 NU_AND_CONSUME 은 지정한 모든 event flag 가 설정되어야 하고, NU_OR 와 NU_OR_CONSUME 은 지장한 event flag 중 하나 이상이면 충분하다. CONSUME 은 event flag 를 자동으로 clear 할 때 쓴다.

예)

```
NU_EVENT_GROUP      Events;
UNSIGNED            actual_flags;
STATUS              status;

status = NU_Retrieve_Events(&Group, 0x86, NU_AND_CONSUME, &actual_flags,
                            NU_SUSPEND);
```

3.3.5 Monitor event group status

- NU_Established_Event_Groups

현재 establish 되어있는 event group 의 개수를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
UNSIGNED        total_event_groups;

total_event_groups = NU_Established_Event_Groups();
```

- NU_Event_Group_Pointers

현재 establish 되어있는 event group 의 pointer 와 숫자를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
NU_EVENT_GROUP      *Pointer_Array[20];
UNSIGNED            number;

number = NU_Event_Group_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- NU_Event_Group_Information

지정한 event group 에 대해서 다양한 값을 보여준다. Event group 의 이름, 값, 기다리는 task 들, 첫째로 기다리는 task 등을 보여준다.

예)

```
NU_EVENT_GROUP      Events;
CHAR                group_name[8];
UNSIGNED            event_flags;
UNSIGNED            tasks_suspended;
NU_TASK             *first_task;
STATUS              status;

status = NU_Event_Group_Information(&Events, group_name, &event_flags,
                                    &tasks_suspended, &first_task);
```

3.4 Signal (Task Synchronization)

3.4.1 Control Signal

이 함수를 호출하는 task 의 signal 을 enable 이나 disable 시킨다.한 task 당 32 개의 signal 이 있다. 이 signal 은 task 생성시 clear 된다.

예)

```
UNSIGNED old_signal_mask;

old_signal_mask = NU_Control_Signals(0);
NU_Control_Signals(old_signal_mask);
```

3.4.2 Register Signal Handler

signal handler routine 을 등록한다. 등록이 되어야 앞의 NU_Control_Signals 가 실효를 나타낸다.

예)

```
STATUS          status;

void Signal_Handler (UNSIGNED signals)
{
        /* signal 을 처리하는 routine */
}
status = NU_Register_Signal_Handler(Signal_Handler);
```

3.4.3 Send

지정한 task 에 지정한 signal 을 보낸다.

예)

```
NU_TASK         task;
STATUS          status;

status = NU_Send_Signals(&Task, 0x80000082);
```

3.4.4 Receive

현재 task 의 signal 을 받고 clear 한다.

예)

```
UNSIGNED        signals;
signals = NU_Receive_Signals();
```

3.5 Mailbox (Task Communication)

3.5.1 Create Mailbox

mailbox 로 통신을 하기전에 우선 먼저 mailbox 를 만든다.
Nucleus PLUS 에서의 mailbox 는 오직 1 개의 message 를 저장한다. task 가 suspend 될때의 순서는 FIFO 나 priority 2 가지가 있다.

예)

```
NU_MAILBOX      Mailbox;
STATUS          status;

status = NU_Create_Mailbox(&Mailbox, “any name”, NU_FIFO);
```

3.5.2 Delete

mailbox 를 없앤다. suspend 된 경우에는 이 mailbox 를 기다리고 있던 것은 resume 되며 error signal 이 뜬다.

예)

```
NU_MAILBOX      Mailbox;
STATUS          status;

status = NU_Delete_Mailbox(&Mailbox);
```

3.5.3 Send

mailbox 로 message 를 보낸다. mailbox 가 비어 있으면 바로 message 가 복사되고 차있는 경우는 suspend option 에 따라 작동한다.

예)

```
NU_MAILBOX      Mailbox;
UNSIGNED        message[4];
STATUS          status;

status = NU_Send_To_Mailbox(&Mailbox, &message[0], 25);
```

3.5.4 Broadcast

mailbox 의 message 를 기다리는 모든 task 로 message 를 보낸다.

예)

```
NU_MAILBOX      Mailbox;
UNSIGNED        message[4];
STATUS          status;

status = NU_Broadcast_To_Mailbox(&Mailbox, &message[0], 20);
```

3.5.5 Receive

maibox 에서 message 를 받는다.

예)

```
NU_MAILBOX      Mailbox;
UNSIGNED        message[4];
STATUS          status;
NU_Receive_From_Mailbox(&Mailbox, &message[0], 20);
```

3.5.6 Reset

현재 mailbox 에 있는 message 를 버린다. mailbox 에 의해 suspend 되어 있던 task 들은 모두 resume 된다.

예)

```
NU_MAILBOX      Mailbox;
STATUS          status;

status = NU_Reset_Mailbox(&Mailbox);
```

3.5.7 Monitor mailbox status

- NU_Established_Mailboxes

현재 establish 되어있는 mailbox 의 개수를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
UNSIGNED        total_mailboxs;

total_mailboxs = NU_Established_Mailboxes();
```

- NU_Mailbox_Pointers

현재 establish 되어있는 mailbox 의 pointer 와 숫자를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
NU_MAILBOX              *Pointer_Array[20];
UNSIGNED                number;

number = NU_Mailbox_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- NU_Mailbox_Information

지정한 mailbox 에 대해서 다양한 값을 보여준다. Mailbox 의 이름, 기다리는 방법(FIFO 또는 priority), 현재 message 가 있는지 없는지, 기다리는 task 들, 첫째로 기다리는 task 등을 보여준다.

예)

```
NU_MAILBOX          Mailbox;
CHAR                mailbox_name[8];
OPTION              suspend_type;
DATA_ELEMENT        message_present;
UNSIGNED            tasks_suspended;
NU_TASK             *first_task;
STATUS              status;

status = NU_Mailbox_Information(&Mailbox, mailbox_name, &suspend_type,
                                &message_present, &task_suspended, &first_task);
```

3.6 Queue (Task Communication)

앞에서 말한 것 처럼 mailbox 와 크기와 개수를 빼고는 거의 차이가 없다. 여러개의 message 를 보관할 수 있기 때문에 queue 의 맨 앞으로 보내는 명령이 추가 되었다.

3.6.1 Create Queue

3.6.2 Delete

3.6.3 Send

3.6.4 Send to front

queue 의 맨 앞으로 보낸다. 급한 message 를 보낼 때 쓴다.

명령형태는 NU_Send_To_Queue 와 같다.

3.6.5 Broadcast

3.6.6 Receive

3.6.7 Reset

3.6.8 Monitor queue status

3.7 Pipe (Task Communication)

Queue 와 크기가 다르다는 차이외에는 거의 같다. 명령어 형태도 마찬가지이다.

3.7.1 Create Pipe

3.7.2 Delete

3.7.3 Send

3.7.4 Send to front

3.7.5 Broadcast

3.7.6 Receive

3.7.7 Reset

3.7.8 Monitor pipe status

3.8 Partition Pool (Memory Management - Fixed)

C 프로그램에서 메모리를 할당하고 싶으면 보통 malloc, calloc 과 같은 함수를 쓰고 다 쓴 메모리는 free 를 한다. 이것은 물론 real-time 환경을 고려하지 않은 함수이기 때문에 real-time OS 에서는 메모리를 관리하는 함수를 별도로 제공하고 있다. 대개의 real-time OS 에서는 fixed length memory 와 variable length memory 를 제공하는데 가끔 fixed length memory 만을 제공하는 OS 도 있다.

Fixed length memory 라는 것은 할당할 메모리 block 이 모두 같은 크기로 되어 있다는 것이고 variable length memory 라는 것은 할당할 메모리 block 의 크기가 정해져 있지 않은 것이다. Fixed length 를 쓰는 경우에는 deterministic 하다는 장점이 있고 variable length memory 를 쓰는 경우에는 메모리를 좀 더 효율적으로 쓸 수 있다는 장점이 있다.

그렇다면 언제 쓸 것인가? 물론 malloc 을 대신해 쓸 수도 있다. 또한 message 를 넘겨줄 때도 쓸 수 있다. 만약 40byte 의 크기를 넘겨야 한다면 단순히 구현하면 복사를 해야하므로 메모리를 복사하는데 시간이 많이 걸린다. 이때 메모리를 할당하고 포인터만을 넘기면 메모리 복사를 하지 않아서 시간을 줄일 수 있고 메모리 사용에서도 효과적이라 할 수 있다.

3.8.1 Create Partition Pool

fixed size memory partition 을 할당한다. 메모리 시작번지, 전체 pool 의 크기, partition 하나의 크기, suspend type 등을 지정할 수 있다.

예)

```
NU_PARTITION_POOL     Pool;
STATUS                status;

status = NU_Create_Partition_Pool(&Pool, “any name”, (VOID *) 0xB000, 2000, 40, NU_FIFO);
```

3.8.2 Delete Partition Pool

partition pool 을 없앤다. suspend 된 경우에는 이 partition pool 을 기다리고 있던 것은 resume 되며 error signal 이 뜬다.

예)

```
NU_PARTITION_POOL     Pool;
STATUS                status;

status = NU_Delete_Partition_Pool(&Pool);
```

3.8.3 Allocate

Partition pool 에서 하나의 partition 을 할당한다.

예)

```
NU_PARTITION_POOL     Pool;
VOID                  *memory_ptr;
STATUS                status;

status = NU_Allocate_Partition(&Pool, &memory_ptr, NU_SUSPEND);
```

3.8.4 Deallocate

할당한 partition 을 해제한다.

예)

```
STATUS                status;

status = NU_Deallocate_Partition(partition);
```

3.8.5 Monitor partition pool status

- NU_Established_Partition_Pools

현재 establish 되어있는 partition pool 의 개수를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
UNSIGNED      total_partition_pools;

total_partition_pools = NU_Established_Partition_Pools();
```

- NU_Partition_Pool_Pointers

현재 establish 되어있는 partition pool 의 pointer 와 숫자를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
NU_PARTITION_POOL     *Pointer_Array[20];
UNSIGNED              number;

number = NU_Partition_Pool_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- NU_Partition_Pool_Information

지정한 partition pool 에 대해서 다양한 값을 보여준다. Partition pool 의 이름, 기다리는 방법 (FIFO 또는 priority), 현재 message 가 있는지 없는지, 기다리는 task 들, 첫째로 기다리는 task 등을 보여준다.

3.9 Memory Pool (Memory Management - variable)

할당할 수 있는 메모리 크기가 가변적이라는 것을 제외하고는 Partition pool 과 같다.

3.9.1 Create Memory Pool

할당할 수 있는 최소 크기를 지정하는 것을 제외하고는 partition pool 과 같다.

3.9.2 Delete Memory Pool

3.9.3 Allocate

할당할 크기를 지정하는 것을 제외하고는 partition pool 과 같다.

3.9.4 Deallocate

3.9.5 Monitor memory pool status

3.10 Timer Management

Timer 는 일정 시간이 지나면 expire 하면서 특정 프로그램을 수행하도록 만드는 역할을 한다. Real-time OS 에서는 timer interrupt 를 하나 받아서 이를 바탕으로 여러 개의 software timer 를 쓸 수 있도록 해준다.

timer 는 가장 기본적으로는 system tick 으로 time slice 하는 경우, sleep 하는 경우 등에 쓰인다. Timer Interrupt 는 시스템에서 쓰는 timer 마다 다르기 때문에 뭐라 말할 수는 없지만 대개 timer 에 관한 예제 source 가 딸려 오므로 이를 참고해서 변경하면 될 것이다. timer 관련 함수를 쓰지 않는 경우에는 porting 시에 timer 를 지정하지 않아도 되지만 대개 tick 을 쓰게 되므로 timer 를 시스템에 맞게 고쳐야 한다.

3.10.1 Create Timer

하나의 software timer 를 만든다. 그 Timer 가 expire 될 때 수행되는 expiration routine 을 넣을 수 있다. 처음에 expire 하기까지의 tick 수, 그 이후에 expire 하기 까지의 tick 수등을 지정할 수 있다.

예)

```
NU_TIMER        Timer;
STATUS          status;

status = NU_Create_Timer(&Timer, “any name”, timer_expire, 0, 23, 5, NU_ENABLE_TIMER);
```

3.10.2 Delete

timer 를 없앤다. 없애기 전에 우선 그 timer 를 disable 시켜야 한다.

예)

```
NU_TIMER        Timer;
STATUS          status;

status = NU_Delete_Timer(&Timer);
```

3.10.3 Control

timer 를 enable 시킬 것인지 disable 시킬 것인지를 결정한다.

예)

```
NU_TIMER        Timer;
STATUS          status;

status = NU_Control_Timer(&Timer, NU_DISABLE_TIMER);
```

3.10.4 Set Clock

시스템 clock 을 지정한 값으로 설정한다.

예)

```
NU_Set_Clock(0);
```

3.10.5 Retrieve Clock

현재 시스템 tick clock 을 알아낸다. 이 counter 는 timer interrupt 가 발생할 때마다 1 씩 증가한다.

예)

```
UNSIGNED        clock_value;

clock_value = NU_Retrieve_Clock();
```

3.10.6 Reset

지정한 timer 에서 새로운 expire 루틴, reset 부터 expire 하기까지의 tick 수, 그 이후에 expire 하기 까지의 tick 수등을 지정할 수 있다.

예)

```
NU_TIMER        Timer;
STATUS          status;

status = NU_Reset_Timer(&Timer, new_expire, 3, 30, NU_ENABLE_TIMER);
```

3.10.7 Monitor timer status

- NU_Established_Timers

현재 establish **되어있는** timer **의 개수를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
UNSIGNED        total_timers;

total_timers = NU_Established_Timers();
```

- NU_Timer_Pointers

현재 establish **되어있는** timer **의** pointer **와 숫자를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
NU_TIMER                *Pointer_Array[20];
UNSIGNED                number;

number = NU_Timer_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- NU_Timer_Information

지정한 timer **에 대해서 다양한 값을 보여준다**. Timer **의 이름**, enable **인지** disable **인지**, expire **한 횟수**, **처음에** expire **하기까지의** tick **수**, **그 이후에** expire **하기 까지의** tick **수등을 알 수 있다**.

예)

```
NU_TIMER        Timer;
CHAR            timer_name[8];
OPTION          enable;
UNSIGNED        expirations;
UNSIGNED        remaing_time;
UNSIGNED        id;
UNSIGNED        initial_time;
```

```
UNSIGNED        reschedule_time;
STATUS          status;

status = NU_Timer_Information(&Timer, timer_name, &enable, &expirations, &id, &intial_time,
                              &reschedule_time);
```

3.11 Interrupt Management

interrupt 란 외부나 내부의 event 에 즉시 반응할 수 있는 장치를 말한다. interrupt 가 발생되면 processor 는 현재 수행 중이던 프로그램을 중단시키고 제어권을 Interrupt Service Routine(ISR)로 넘긴다. 앞에서도 언급했지만 ISR 에서는 실제 수행할 task(HISR)를 호출하고 종료되고, HISR 이 이후에 실제 작업을 담당하게 된다.

3.11.1 Create HISR

HISR 을 만드는 것으로 task 와 마찬가지로 HISR 자체의 stack 이 필요하며 HISR 사이에는 0~2 의 priority 를 지닌다.

예)

```
NU_HISR         HISR;
STATUS          status;
void            HISR_Entry(INT vector);

NU_Create_Memory_Pool(&System_Memory, “SYSTEM”, first_available_memory,
                      20000, 0, NU_FIFO);
NU_Allocate_Memory(&System_Memory, &stack_pointer, 400, NU_NO_SUSPEND);
status = NU_Create_HISR(&HISR, “any name”, HISR_Entry, 2, stack_pointer, 400);
```

3.11.2 Delete HISR

더 이상 HISR 을 실행시키지 않게 하기 위해서 쓴다. 앞으로 다시는 쓸 일이 없는 HISR 이라고 생각될 때 delete 한다.

예)

```
NU_HISR         Hisr;
STATUS          status;

status = NU_Delete_HISR(&Hisr);
```

3.11.3 Register LISR

LISR 함수를 interrupt vector 와 연결시킨다.

예)

```
STATUS  status;
VOID    (*old_lisr)(INT);

void LISR_example(INT vector_number)
{
        /* vector_number contains the actual interrupt vector number */
}
status = NU_Register_LISR(10, LISR_example, &old_lisr);
```

3.11.4 Activate HISR

Interrupt 가 걸리면 우선 LISR 이 호출되는데 LISR 안에서는 실행할 수 있는 명령이 몇개 없다. 실제로 처리하는 루틴은 HISR 에 존재하는데 LISR 에서 HISR 을 부르는 루틴이다.

예)

```
extern NU_TASK  *Task_0_Ptr;
NU_HISR         HISR_Control;
CHAR            HISR_Stack[500];
VOID            (*old_lisr)(INT);
VOID            Example_LISR(INT vector);
VOID            Example_HISR(VOID);

NU_Create_HISR(&HISR_Control, “EXMPHISR”, Example_HISR, 2, HISR_Stack, 500);
NU_Register_LISR(10, Example_LISR, &old_lisr);
VOID Example_LISR(INT vector)
{
    NU_Activate_HISR(&HISR_Control);
}
VOID Example_HISR(void)
{
    /* code */
    NU_Resume_Task(Task_0_Ptr);
}
```

3.11.5 Setup Vector

vector 로 지정된 interrupt vector 를 caller 에서 제공하는 Interrupt Service Routine 으로 바꾼다.

예)

```
VOID *old_vector;

old_vector = NU_Setup_Vector(5, asm_ISR);
```

3.11.6 Control Interrupts

interrupt 를 enable 시키거나 disable 시킨다. 이때 level 을 지정할 수 있는데 이것은 프로세서와 관련있는 값이다. 그러나 NU_DISABLE_INTERRUPTS 나 NU_ENABLE_INTERRUPTS 를 쓰면 모든 interrupt 를 enable 시키거나 disable 시킬 수 있다.

예)

```
INT old_level;

old_level = NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);
NU_Control_Interrupts(old_levels);
```

3.11.7 Protect

critical data structure 를 protect 하는 것이다. I/O Driver 에서는 task 와 HISR 에서 동시에 access 하는 것을 막을 경우가 생긴다. 이 서비스는 I/O Driver 에 있는 data structure 를 보호하기 쓰인다.

예)

```
NU_PROTECT Protect_Struct;

NU_Protect(&Protect_Struct);
```

3.11.8 Unprotect

NU_Protect 에 의해 보호되었던 critical data structure 에 대한 보호를 푸는 것이다.

예)

```
NU_Unprotect(&Protect_Struct);
```

3.11.9 Monitor HISR status

- Current Active Task Pointer

현재 active **인** HISR **을 알려준다**.

예)

```
NU_HISR         *HISR_ptr;

HISR_ptr = NU_Current_HISR_Pointer();
```

- Established HISRs

현재 establish **되어있는** hisr **의 개수를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
UNSIGNED        total_hisrs;

total_hisrs = NU_Established_HISRs();
```

- HISR Pointer

현재 establish **되어있는** HISR **의** pointer **와 숫자를 알려준다**. delete **한 것은 포함되지 않는다**.

예)

```
NU_HISR         *Pointer_Array[20];
UNSIGNED        number;

number = NU_HISR_Pointers(&Poiner_Array[0], 20); /* list 최대 크기 */
```

- HISR Information

지정한 HISR **에 대해서 다양한 값을 보여준다**. HISR **의 이름**, **상태**, priority, stack **상태 등을 보여준다**.

예)

```
NU_HISR         Hisr;
CHAR            hisr_name[8];
UNSIGNED        activations;
DATA_ELEMENT priority;
```

```
VOID            *stack_base;
UNSIGNED        stack_size;
UNSIGNED        minimum_stack;
STATUS          status;

status = NU_HISR_Information(&Hisr, hisr_name, &activations, &priority, &Stack_base,
                             &stack_size, &minimum_stack);
```

3.12 I/O Driver

3.12.1 Create Driver

I/O Driver 를 create 한다. 실제로 driver 가 invoke 되지는 않는다.

예)

```
NU_DRIVER       Driver;
STATUS          status;

VOID Driver_Entry(NU_DRIVER *driver, NU_DRIVER_REQUEST *request)
{
    switch(request -> nu_function)
    {
        case NU_INITIALISE:
        /* initialization processing */
        break;
        case NU_ASSIGN:
        /* assign processing */
        break;
        case NU_RELEASE:
        /* release processing */
        break;
        case NU_INPUT:
        /* input processing */
        break;
        case NU_OUTPUT:
        /* output processing */
        break;
        case NU_STATUS:
```

```
            /* status processing */
            break;
            case NU_TERMINATE:
            /* terminate processing */
            break;
            default:
            /* bad request processing */
            break;
    }
}
status = NU_Create_Driver(&Driver, “any name”, Driver_Entry);
```

3.12.2 Delete

I/O Driver 를 delete 한다. Terminate request 가 들어온 경우에 수행된다.

예)

```
NU_DRIVER       Driver;
STATUS          status;

status = NU_Delete_Driver(&Driver);
```

3.12.3 NU_Request_Driver

2 번째 paramater 가 가리키는 structure 로 reqeust 를 보낸다.
아래의 예에서 request.nu_function 을 NU_INITIALSE, NU_ASSIGN, NU_RELEASE 등으로 지정하면 그에 해당되는 routine 이 호출된다. (NU_Create_Driver 참고)

예)

```
NU_DRIVER               Driver;
NU_DRIVER_REQUEST       request;
STATUS                  status;

request.nu_function = NU_INITIALIZE; /* case NU_INITIALIZE:에 해당되는 루틴 수행 */
request.nu_timeout = NU_NO_SUSPEND; /* suspend 여부 */
status = NU_Request_Driver(&Driver, &request);
```

3.12.4 NU_Resume_Driver

NU_SUSPEND_Driver 에 의해서 suspend 되었던 task 를 resume 한다.

예)

```
NU_TASK         Task;
STATUS          status;

status = NU_Resume_Driver(&Task);
```

3.12.5 NU_Suspend_Driver

I/O Driver 내에서 task 를 suspend 시킨다. 이렇게 하면 I/O Driver 가 internal data structure 를 protect 하고 그 task 를 suspend 한 상태로 할 수 있다.

예)

```
NU_Suspend_Driver(NU_NULL, NU_NULL, 0);
```

3.12.6 Monitor driver status

- NU_Established_Drivers

현재 establish 되어있는 I/O Driver 의 개수를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
UNSIGNED        total_drivers;

total_drivers = NU_Established_Drivers();
```

- NU_Driver_Pointers

현재 establish 되어있는 Driver 의 pointer 와 숫자를 알려준다. delete 한 것은 포함되지 않는다.

예)

```
NU_DRIVER       *Pointer_Array[20];
UNSIGNED        number;

number = NU_Driver_Pointers(&Pointer_Array[0], 20);
```

3.13 Development Services

3.13.1 Enable History Saving

internal history saving 을 enable 시킨다.

예)

NU_Enable_History_Saving();

3.13.2 Disable History Saving

internal history saving 을 disable 시킨다.

예)

NU_Enable_History_Saving();

3.13.3 Make History Entry

history-log 가 enable 되어 있으면 system history log 를 만든다.

예)

NU_Make_History_Entry(1,2,3);

3.13.4 Retrieve History Entry

위에서 만든 log 를 retrieve 한다.

예)

```
DATA_ELEMENT        id;
UNSIGNED            param1;
UNSIGNED            param2;
UNSIGNED            param3;
UNSIGNED            time;
NU_TASK             *task;
NU_HISR             *hisr;

NU_Retrieve_History_Entry(&id, &param1, &param2, &param3, &time, &task, &hisr);
```

3.13.5 License Information

customer 의 serial number 와 product description 정보를 알려준다.

예)

```
CHAR    *license_string;

license_string = NU_License_Information();
```

3.13.6 Release Information

Nucleus PLUS Release **정보를 알려준다**.

예)

```
CHAR    *release_pointer;

release_pointer = NU_Release_Information();
```

Part 3: Real-time System Design

4 어떤 Real-time OS를 쓸 것인가?

Real-time OS를 쓰기로 결정했다면 과연 어떤 real-time OS를 쓸 것인지 결정할 지 상당히 고민이 될 것이다. 현재 나와있는 real-time OS 제품들이 그 수가 수십 개에 달하기 때문이다.

초기에는 real-time OS를 선정할 때에는 latency를 얼마나 최소화할 수 있는가 - external interrupt, kernel service, task switching과 관련된 latency를 최소화 - 였으나 오늘날에 microprocessor가 발전하면서 latency 최소화의 중요성이 점차 약해지고 있다.

이제는 이런 것 보다 더 다양한 면에서 여러 real-time OS를 비교할 수 있는 기준이 필요하다. 그렇다면 우선 real-time OS를 쓰는 이유는 생각해 봐야 할 것이다.

이상적으로 말한다면 성능을 저하시키지 않고 메모리를 많이 요구하지 않으면서 코드를 복잡하게 하지 않으면서 적절한 system service call을 제공하고 task scheduling에서도 유연성을 줄 수 있어야 한다. 즉, 간단히 말해서 효율적이면서 쉽게 개발할 수 있도록 해야 한다는 것이다. 우선 이런 점을 만족시켜야 할 것이지만 실제로 선정을 할 때에는 이런 기능적인 측면 외에도 다른 면들도 고려해야 한다. 이제부터 한가지씩 살펴보기로 하자.

4.1 원하는 processor를 지원하는가? 또 새로운 processor에 대한 portability는?

우선 무엇보다도 시스템에서 쓰고자 하는 processor를 지원해야 할 것이다. 이미 나온 지가 오래된 processor에 대해서는 지원하는 RTOS들이 꽤 있을 것이지만 최근에 나온 processor에 대해서는 지원하는 OS들이 많지 않을 수도 있다.

또한 개발한 시스템을 upgrade한다고 가정해 보자. 이때 새로운 processor를 쓰게 될 수도 있다. 이전의 시스템에서 쓰던 RTOS가 새로운 processor를 지원하지 못한다면 다른 RTOS를 쓸 수밖에 없다.

이를 해결하기 위해서는 2가지 방법이 있을 수 있는데 하나는 처음부터 새 processor에 대한 지원이 빠른 RTOS를 선정하는 방법이고 다른 하나는 TRON이나 POSIX와 같이 표준을 지원하는 RTOS를 쓰는 방법이다. 이때는 표준의 API를 쓰게 되므로 새 processor를 지원하는 다른 RTOS가 같은 API를 제공한다면 이 역시 porting이 쉽게 된다. 그러나 아직 이

표준은 완벽하지 못하며(이에 대해서는 뒤에서 다시 설명한다) 다른 RTOS 가 API 를 지원해야 한다는 문제가 있다. 따라서 가장 좋은 방법은 새 processor 에 대한 portability 가 좋은 RTOS 를 선정하는 것이다.
새로운 processor 가 나왔을 때 RTOS 가 어떻게 port 되는가를 조사해보면 그 다음 새로운 processor 가 나왔을 때 그 RTOS vendor 가 빨리 port 할 수 있을지 도움이 될 것이다.
예전에는 RTOS 가 application 에 부가하는 overhead 가 작고 external event 에 대해 빠르고 결정된 반응을 하기 위해서 assembly 언어로 작성되었다.
그러나 새로운 processor 에 porting 이 어려웠기 때문에 최근에는 C 언어로 바꿔왔다. 이는 또한 RISC 프로세서처럼 좋은 compiler 에 의존하는 경우를 생각해 본다면 자연스러운 일이다. 대개 time-critical scheduling 과 task switching routine 은 assembly 언어로 작상하고 그 외의 부분은 모두 C 로 작성한다.

4.2 Scalability

memory 가 제한이 되어 있는 시스템인 경우에 - 특히 RISC - OS 의 크기도 줄이려는 경우가 있다. 이때 어떤 OS 들은 매우 기본적인 일만 하도록 하는 kernel 을 작성한 후 필요에 따라서 추가하도록 한다. 이때 추가가 쉽고 효율적이어야 한다는 것이다.

4.3 Multiprocessor support

최근에는 real-time OS 를 embedded system 뿐 아니라 multiprocessor system 에서도 많이 쓰고 있다. 기존의 OS 는 단일 processor 에서만 작동하도록 되어 있기 때문에 processor 가 여러개 있는 경우 이를 제어할 OS 가 필요하고 여기에 real-time OS 가 쓰이고 있다. 따라서 multiprocessor 시스템을 설계할 때엔 어떤 real-time OS 가 지원되는지 살펴봐야 할 것이다.

4.4 Extended services, Vertical Application

실제 응용프로그램을 만들다 보면 여러가지 service 가 필요하다. 예를 들어 network 과 연계되는 시스템을 설계한다고 할때 network protocol 이 필요할 것이며 가장 대표적으로 TCP/IP, SNMP, ISDN 등이 있다. 또한 File system 을 붙이는 시스템도 있을 것이다.
이런 것들을 위해서 real-time OS vendor 들이 이런 표준들에 대해서는 자신들이 software 를 만들어서 판매를 한다.
가격은 좀 비싼 편이지만 자체 개발비와 개발기간등을 고려해본다면 자신들의 시스템에 적용할 수 있는 것은 사고 나머지를 개발하는 것도 좋은 방법이다.

4.5 standards/POSIX compliance

POSIX 는 앞에서도 말한 바 있지만 standard API 이므로 이를 따르는 RTOS 들 사이에서는 더욱 porting 이 쉬워진다. 다음의 예를 보면 왜 표준의 API 를 따르는 것이 좋은 지 이해가 될 것이다.

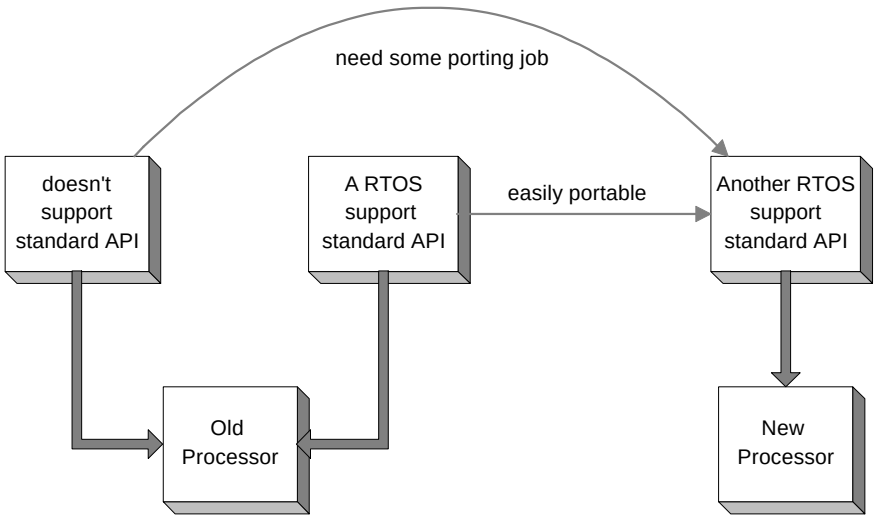


그림 4-1 표준 API 를 쓸 때의 장점

표준의 API 란 것은 쉽게 말해서 함수의 포맷이 같다는 것이다. 즉, 함수에게 전달되는 매개변수의 type 이 같기 때문에 표준의 API 를 쓰는 RTOS 사이에서는 다른 RTOS 를 써도 service call 을 바꿀 필요가 없는 것이다. 단 여기에서 말하는 것은 응용프로그램의 portability 이지 RTOS 자체의 portability 와는 무관한 것이다.
그러나 현재 POSIX 계열은 완벽하지 못하기 때문에 당장에는 적용하기 힘든 상황이지만 TRON 계열을 쓰는 경우에는 고려해볼만 하다.
만약 표준의 API 를 쓰지 않는 경우에는 RTOS 와 응용 프로그램사이에 Interface 를 두는 것이 좋다. 이에 대해서는 5 장에서 다시 설명하겠다.

4.6 Language support

task 프로그램을 무엇으로 작성할 수 있는가를 말한다. 현재 C 언어가 가장 많이 쓰이고 있으므로 기본적으로 지원되어야 할 것이며 최근들어 각광받고 있는 Object-Oriented Language, 특히 C++에 대한 지원이 가능하면 더욱 좋을 것이다. 이것은 일반적인 경우이며 경우에 따라서 ADA 나 Fortran 이 필요한 경우에는 이 언어를 지원하는 OS 를 찾아야 할 것이다.

4.7 Development environment

Real-time OS 를 개발하는 경우는 그 debugger 또한 기존의 source level debugger 와는 다른 OS 전용 debugger 가 필요하다. Source level debugger 는 kernel 의 상태를 알기 힘들기 때문에 이를 알수 있는 OS 전용 debugger 가 필요하다. 이런 debugger 를 쓰면 현재 어떤 task 가 suspend 되었다거나 어느 task 가 semaphore 를 잡고 있다는 등의 정보를 알 수 있다.
보통 이런 OS debugger 는 application 에 추가로 하나의 task 로 붙게 되어 있다. 그래서 debugger user interface 를 통해서 질의가 들어오면 그 task 가 그때의 값들을 넘겨주게 되어

있다. 개발을 편하게 하기 위해서는 이런 user interface 가 source level debugger 와 합쳐져있는 것이 좋다. 대개 real-time OS vendor 에서는 몇가지 source level debugger 에 맞춰서 제품을 팔고 있는데 자신이 쓰려는 debugger 를 real-time OS vendor 에서 지원해주고 있는 지 알아볼 필요가 있다.

4.8 Licensing arrangements and price

RTOS 를 선정할 때 고려하는 또 하나의 중요한 점은 license 문제이다. 대개의 RTOS 는 system 이 판매될 때 1 개당 얼마의 license fee 를 받는다. 예를 들어 10cent/1piece 의 license 를 받는다면 100 만개가 팔리는 경우에 license fee 만 10 만$를 지불해야 한다. 따라서 어느 RTOS 가 license fee 가 작은가를 살펴보는 것도 중요하다.

4.9 Real-time OS 의 미래

앞으로 embedded system 에서 real-time OS 의 숫자가 점차 늘어날 것이다. Handheld PC 에서 채용한 Windows-CE 에서 CE 란 Consumer Electronics 의 약자로서 가전제품에도 OS 를 넣겠다는 것이다. 즉, 무수한 제품에 눈에 보이지 않는 컴퓨터가 들어갈 것이며 가전제품들 사이에 network 기능을 이용해서 제어하는 home automation 시스템도 가능할 것으로 생각된다. 따라서 real-time OS 의 숫자도 현재와는 달리 매우 많이 늘어날 것이다.
또한 최근에 늘고 있는 multiprocessor 시스템에서 적용되는 것은 multiprocessor 시스템의 특성상 그 수요가 급격히 증가하지는 않겠지만 역시 점차 그 수가 늘어날 것으로 생각된다.

5 Real-time Embedded System 설계

5.1 기존의 Embedded System 설계

Real-time OS 를 쓰지 않고 Embedded System 을 설계하는 경우는 프로그램을 순차적으로 작성해야 한다. 그렇지 않은 경우는 interrupt 가 발생하는 경우다.
따라서 전체적인 시간 분배같은 작업은 개발자가 해야 하며 중간에 interrupt 가 발생되는 경우도 고려해야 한다. 개발자가 계획을 정확히 세운다면 OS 를 쓰는 것보다 좀 더 효율적이 될 수 있지만 프로그램이 복잡해지면 점점 어려워진다. assembly 언어를 쓰는 경우 high level 언어를 쓰는 것보다 좀 더 효율적으로 작성할 수 있으나 훨씬 많은 개발 시간을 요하며 개발자가 실력이 없으면 오히려 효율적인 compiler 를 쓰는 것보다 못할 수도 있다는 것이 이와 비슷한 예라고 하겠다.

5.2 Real-time OS 를 쓰는 경우의 장점

최근에는 System 으로 들어오는 입력이 많아졌다. 따라서 각각을 처리하는 부분을 독립적으로 프로그램을 작성하고 하나의 프로그램에서 순차적으로 하는 것은 매우 어렵다.
따라서 기능별로 task 를 작성하고 priority 를 적절히 준다면 개발자가 시스템을 설계하는 것이 훨씬 쉬워질 것이다.

5.3 어떻게 Task 로 나누는가?

간단한 시스템을 예로 들어보자.

1. 매 100ms 마다 data 를 읽는다.
2. 1 초마다 평균 data 를 출력한다.
3. keyboard 입력을 기다리고 반응한다.

이때 cyclic schedule 을 한다면 100ms 를 기다린후 data 를 읽고 keyboard 입력이 있는지 검사하고 이 loop 이 10 번이 돌때마다 data 를 출력하게 할 수 있을 것이다. 이경우는 1 개의 task 로만으로도 작성할 수 있다. 그러나 만약 1 초마다 평균 data 를 출력하는 걸 바꾼다고 생각하면 이 코드 전체에 영향을 미친다.
이럴 경우에는 아예 task 를 3 개로 나눌 수도 있다. 100ms 마다 읽는것, 1 초마다 출력하는 것은 각각 하나의 task 로 만들고 시간에 관련해서는 timer 를 설정하고 keyboard 도 별도로(물론 interrupt 로 처리할 수도 있겠다) task 를 만든다. 이때는 약간의 overhead 가 붙겠지만 시스템을 이해하는 경우엔 task 를 3 개로 나눈 것이 좀 낫다.
그러나 task 를 나누어 design 하는 것이 좋다고 무조건 task 를 많이 만든다면 이 또한 좋지

않다.

실제 응용프로그램에서 task 를 나누는 예로 이해하기 좋은 것으로 Network model 이다. OSI 의 7 layer 같은 경우에 각 layer 는 일단 1 개 이상의 task 로 구성하는 것이 좋다. network 에서 송신측이 어떤 message 를 format 에 맞춰 수신측으로 보내려면 자신의 layer 보다 하나 아래의 layer 로 message 를 보내야 한다. 이때 각 layer 를 task 로 보고 전달하고자 하는 message 는 mailbox, queue, pipe 와 같은 개념을 이용하면 된다.

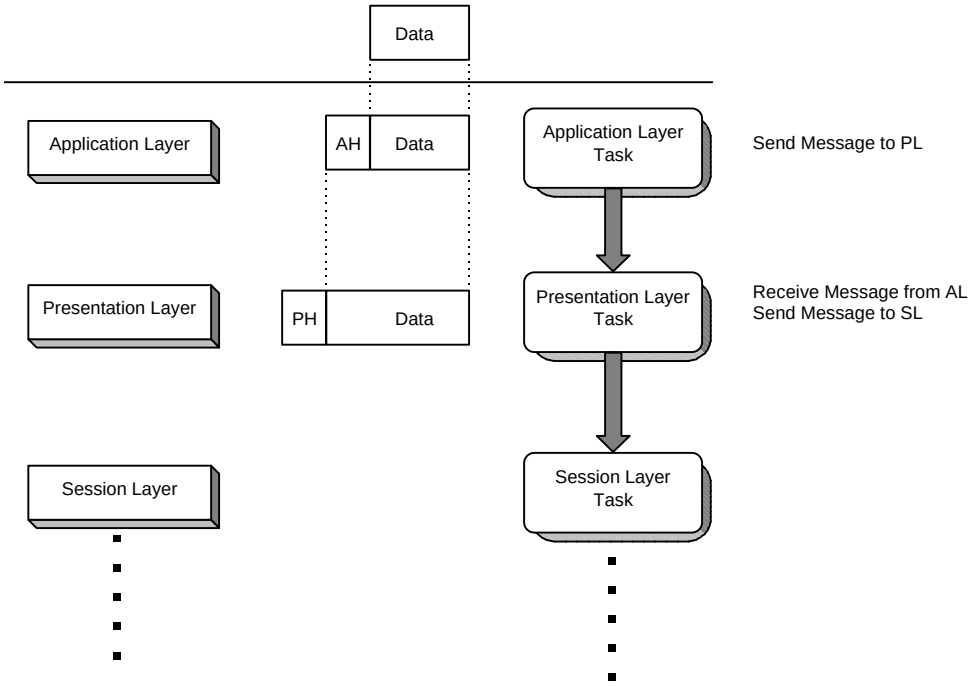


그림 5-1 OSI 7 layer 를 각각 task 로 구성

또한 특정 layer 에서도 해야할 작업이 많다면 이 또한 여러개로 나눌 수 있다. Layer 1 의 경우는 Hardware 에 관련된 것이 많으면서 동시에 layer 2 와도 통신을 해야하기 때문에 같은 layer 1 에서도 layer2 와 통신하는 부분, Hardware 를 제어하는 부분 등을 task 로 나눌 수 있다.

어떤 interrupt 가 있다면 그 ISR 에서 높은 priority 를 갖는 task 를 만들도록 하는 것도 하나의 방법이다.

5.4 Priority 는 어떻게?

priority 를 주는 경우엔 매우 조심해야한다. Real-time OS 는 자원을 얼마나 공평하게 쓰는가가 목적이 아니기 때문에 높은 priority 를 갖는 task 가 제어권을 넘기지 않으면 다른 task 는 수행되지 못하게 된다.(이를 starvation 이라고 한다) 또한 priority 를 잘못주는 경우에는 context switching 을 더 많이 유발시킬 수도 있다.

따라서 시스템 설계를 할때 이점을 고려해야 하며 round-robin 과 같은 방법을 써서 일정시

간이 지난후 중단되도록 하는 것도 좋다. 혹은 starvation 을 막기 위해 starving task 를 점차로 priority 를 올려주는 것도 하나의 방법이다.
또한 대부분의 OS 에서 많은 level 의 priority 를 주고 있는데(대개 256 이상) 그렇다고 모든 task 를 전부 다른 priority 를 줄 필요도 없다.
priority 를 결정할 때에는 모든 priority 를 갖게 해보는 것도 하나의 방법이다. 그리고 어떤 특정 task 의 priority 를 높게 했을때 어떤 결과를 내는지 등을 분석하는 것도 하나의 방법이 되겠다.
ISR 과 같은 것은 대개 특정 task 를 호출하고 끝나게 되어 있다. 따라서 ISR 에서 호출된 함수의 경우에는 priority 를 높이는 것도 하나의 방법이다.
만약 주기적으로 발생되는 신호가 있다면 이를 기준으로 할 수도 있다. 그 주기적인 일은 다음번 주기가 되기 전까지 특정작업을 끝내야하니까 일단 그 주기에서 처리하도록 하고 그렇게 하기 위해서 그 특정작업을 중요한 부분과 그렇지 않은 부분으로 나누고 중요한 부분에는 priority 를 높여주고 그렇지 않은 부분에서는 priority 를 낮춰줄 수 있겠다.

5.5 OS Kernel 과의 Interface

특정 OS 만을 대상으로 프로그램을 작성한다면 OS 에서 제공하는 함수를 직접 호출해도 무방하겠으나 만약 시스템을 upgrade 하는 경우에 그 processor 에 맞는 real-time OS 가 원래 쓰던 OS 와 다르다면 전체 시스템에서 이전 OS 에서 쓰던 함수부분을 모두 고쳐야 할 것이다.
이런 경우에 대비해서 POSIX compliance 라는 것이 있으나 POSIX 라는 표준자체가 미흡한 점이 있어서 마땅한 표준이 없는 상태이다. 따라서 OS Kernel 과 응용 프로그램사이에 interface 를 두면 다른 시스템에 porting 시에 효과적이다.

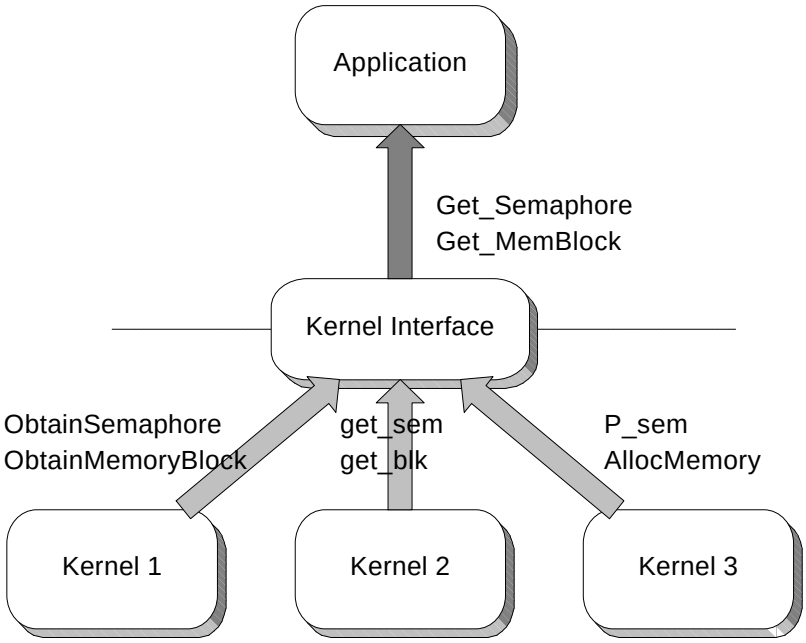


그림 5-2 Kernel Interface

5.6 Real-time OS porting

이 절을 설명하기 전에 여기에서 말하는 porting 에 대해서 잠깐 언급하고 넘어가기로 하자. 4.1 절에 나오는 portability 라는 것은 새로운 processor 에서 OS 가 수행되도록 하는 것을 말하는 것으로서 이런 작업은 OS 를 매우 잘 알면서 새로운 processor 의 구조 또한 매우 잘 아는 전문가가 아니면 할 수 없는 일이다.

그러나 이 절에서 설명하고자 하는 것은 OS 를 쓰고자 하는 시스템에 맞도록 환경을 설정해 주는 작업을 말한다. 앞으로 이절에서 이야기하는 것도 이러한 porting 을 말하는 것이다.

그저 막연히 porting 은 어렵다고 생각했겠지만 앞으로 설명할 작업은 생각보다 어려울 것이 없는 작업이다. 다만, 초보자는 도대체 무엇을 고쳐야 할 지 정확한 개념을 잡지 못하기 때문에 어렵게 느껴질 수 있다. 그러나 OS 와 processor 에 대한 기본 개념을 갖고 있다면 해볼만한 작업이다.

Real-time OS 를 porting 해줄 때 가장 중요한 것은 memory, interrupt, timer 부분이다.

메모리는 원래 C 프로그램을 작성할 때 잡아주는 부분에 real-time OS 에 관련된 부분을 추가하는 것이다. 즉, real-time OS kernel 이 쓸 stack 을 잡아주는 것이다.

C 프로그램을 작성하는 경우에 분명히 그 프로그램은 code 와 data 로 나뉘어 질 것이다. data 영역은 또한 초기화 된 변수, 초기화 되지 않은 변수, 동적으로 사용되는 stack, heap 등으로 나뉜다. 결국 이것들은 메모리에 저장되므로 메모리의 어느 부분에 code 가 들어가고 어느 부분에 data 가 들어갈지를 정해야 하는데 이는 link 시에 결정된다.

이렇게해서 link 시에 stack 영역이 확보가 되었다고 하면 stack 을 다시 나누어 real-time OS

kernel 이 쓸 stack, 사용자들이 쓸 stack 들로 나뉜다.
porting 시에 해줄 작업은 이중에서도 주로 stack 에 관한 것이다. Nucleus PLUS 의 경우에 따라오는 int.s 화일에 보면 기본적으로 stack 을 잡아두고 있는데 이 값을 조정할 필요가 있는 경우엔 사용자가 변경하면 된다.
두번째로 interrupt 부분이다. Interrupt signal 이 발생되면 processor 마다 조금씩 다르겠지만 일단 processor 로 interrupt 가 발생한 경우 특정 번지로 jump 하게 된다.(예를 들어서 INT_IRQ) 이 루틴에서는 interrupt 의 source 가 어디인지 판별하고 그에 해당하는 ISR 로 다시 jump 한다. 이 ISR 을 LISR 이라고 한다. LISR 에서는 대개 단지 task 만을 호출하고 끝난다. 이때 호출된 task 를 HISR 이라고 한다. HISR 은 일반 task 보다 priority 가 높다는 것을 제외하고는 같게 수행된다.
일반적으로 Interrupt 부분을 처리하기 위해서는 시스템에서 제공하는 interrupt controller 를 정확히 이해하는 것이 바람직하다. 어느 interrupt 가 edge-trigger 인지 level-trigger 인지 결정하고 clear 를 해줄 필요가 있는 것은 clear 해주어야 한다는 것이다.
timer 는 시스템에서 쓰는 timer 마다 다르다. timer 가 제대로 설치되지 않으면 OS 에서 제공하는 함수중에서 sleep 이나 time slice 등에 대한 함수가 작동되지 않는다. 간혹 timer interrupt 가 발생한 후 interrupt clear 를 하지 않는 경우에도 문제를 일으킨다.
따라서 sleep 함수가 제대로 작동되지 못한다면 timer interrupt 의 설정이 잘못된 것이므로 다시 살펴봐야 한다.

참고 문헌

[1] Peterson, Silberschatz, “Operating System Conepts” Prentice-Hall.

[2] Jean J. Labrosse, “μC/OS The Real-Time Kernel”, R&D Publications

[3] Bill O. Gallmeister, “POSIX.4: Programming for the Real World”, O’reilly & Associates.

[4] TRON, http://tron.um.u-tokyo.ac.jp/TRON/**에서 찾을 수 있는 문서**