HANSNG UNIV.

OSEK-VDX

번역 좀 해보자

유준희

2009-09-14

걍 번역 함 해볼라고.

서문

OSEK/VDX는 자동차 공업의 협동 프로젝트이다. 이 프로젝트의 목적은 탈 것의 분산 컨트롤 유닛을 위한 개방된 산업 표준이다.
OSEK 프로젝트의 목적과 파트너의 자세한 정보는 "OSEK Binding Specification"을 참조하기 바란다.
이 문서는 모터를 이용한 탈 것에 이용될 멀티태스킹이 가능한 실시간 운영체제의 개념을 서술하고있다. 이 문서는 구현상세(specific implementation)와 관련된 제품에 대한 설명이 아니다.
이 문서는 OSEK 운영체제의 API를 설명하고 있다.
일반적인 협약에 의해, 용어와 약어는 "OSEK Binding Specification"의 추가적인 내부 프로젝트인 "OSEK Overall Glossary"에 모아 두었다.
구현과 시스템 생성에 관해서는 "OSEK Implementation Language"(OIL)상세를 참조하기 바란다.

1. 소개

OSEK 운영체제의 명세는 자동차 컨트롤 유닛 응용 소프트웨어 자원의 효율적인 최적화 통합된 환경을 제안한다. OSEK 운영체제는 분산 임베디드 컨트롤 유닛을 위한 단일 프로세서 운영체제이다.

1.1 시스템 철학

자동차 어플리케이션은 강한 실시간 요구사항으로 특징지어진다. 그렇게 때문에 OSEK 운영체제는 이벤트 드리븐 컨트롤 시스템(Event driven control system)을 지원하기 위해 필요한 기능성을 제공한다.

명세된 운영체제 서비스 다양한 제조업체가 만들어낸 소프트웨어 모듈의 통합의 기반을 구성한다. 자원의 최소 소비 요구와 성능에 의해 결정된 각각의 컨트롤 유닛들의 명시된 특징에 대응하기 위한 주 초점은 어플리케이션 모듈간의 100% 적합성이 아닌 모듈의 직접적인 이식 가능성이다.

OSEK 운영체제가 다양한 컨트롤 유닛을 쓰는 것에 목적을 둔 것과 같이 운영체제는 시간에 민감한 (time-critical)어플리케이션을 다양한 범주의 하드웨어에서 지원할 것이다. 높은 단계의 모듈성과 유연한 구성 능력은 운영체제를 낮은 사양의 마이크로프로세서와 복잡한 컨트롤 유닛 같은 것에 적합하도록 만들기 위해 필수적이다. 이러한 요구사항들은 "적합 클래스" (챕터 3.2, 적합 클래스 참조)의 정의와 특정한 어플리케이션 개작을 위한 확실한 가용성으로 인해 지원된다.

시간에 민감한 어플리케이션을 위한 시스템 개체의 동적 생성은 버려졌다. 대신, 시스템 개체의 생성이 시스템 생성 단계로 지정되었다. 운영체제 내부의 오류 질의는 불필요하게 전체 시스템의 속도에 영향을 미치지 않도록 넓은 영역으로 옮겨지게 되었다. 다른 한편으로, 테스트 단계와 시간에 덜 민감한 어플리케이션이 단일 표준 시스템의 모습을 보장하고 있어도 이를 위해 확장된 에러 질의 시스템 버전을 정의되고 있다.

메모 [유준희1]: 요 문단은 전반적으로 해석이 힘들었다.
On the other hand, a system version with extended error inquiries has been defined. It is intended for the test phase and for less time-critical applications. Even at that stage defined uniform system appearance is ensured.

표준화된 인터페이스

운영체제와 어플리케이션 소프트웨어 사이의 인터페이스는 시스템 서비스에 의해 정의된다. 이 인터페이스는 다양한 프로세서 군 에서 동작하는 운영체제의 구현을 위해 동일하다.
시스템 서비스는 ISO/ANSI-C-like 문법을 따른다. 그러나 시스템 서비스의 구현 언어는 명시하지 않는다.

Scalability

서로 다른 적합 클래스, 다양한 스케쥴링 메커니즘과 구성 특징은 OSEK 운영체제를 광범위한 어플리케이션과 하드웨어에서 실행 가능하게 한다.

메모 [유준희2]: broad spectrum of

OSEK 운영체제는 최소한의 하드웨어 자원(RAM, ROM, CPU 시간)요구와 8비트 마이크로컨트롤러에서도 동작하도록 디자인 되었다.

에러 확인
OSEK 운영체제는 개발 단계의 확장된 상태와 생산 단계의 표준 상태 두 가지 레벨의 에러 확인을 제공한다.
확장된 상태는 운영체제 서비스를 호출할 때 향상된 적합성 체크를 허가한다. 추가적인 에러 확인을 위해 확장 상태는 표준 버전에 비해 더 많은 실행 시간과 메모리 공간을 필요로 한다. 그러나, 많은 에러들은 테스트 단계에서 찾아낼 수 있다. 모든 에러들이 제거되고 나면, 시스템은 표준 버전으로 다시 컴파일 될 수 있다.

메모 [유준희3]: 원문에는 plausibility로 나와있다. 본래의 뜻은 '그럴듯한'

어플리케이션 소프트웨어의 이식성
OSEK의 목표 중 하나는 어플리케이션 소프트웨어의 이식성과 재사용성을 지원하는 것이다. 따라서 어플리케이션 소프트웨어와 운영체제 사이의 인터페이스는 기능적으로 잘 정의된, 표준화된 시스템 서비스에 의해 정의되어야 한다. 표준화된 시스템 서비스의 사용은 유지보수와 어플리케이션 소프트웨어의 이식, 개발 비용을 줄여준다.

이식성은 어플리케이션 내부의 큰 변경 없이 하나의 ECU에서 다른 ECU로 소프트웨어 모듈을 옮길 수 있는 능력을 얘기한다. 운영체제의 표준화된 인터페이스(서비스 콜, 타입 정의와 상수)는 소스코드 레벨에서 이식성을 지원한다. OSEK명세는 목적코드의 교환을 제시하지 않는다.

어플리케이션은 운영체제와 어플리케이션 명세에 준하는, OSEK 명세에 표준화 되지 않은 입력/출력 시스템 인터페이스에 의존한다. 어플리케이션 소프트웨어 모듈은 몇 개의 인터페이스를 가질 수 있다. 운영체제 실시간 제어와 자원 관리 인터페이스뿐만 아니라 시스템으로의 인터페이스나 어플리케이션이 마이크로컨트롤러 모듈과 직접적으로 작업을 지시 받았을 때 하드웨어로의 완전한 기능성을 제시하기 위한 다른 소프트웨어 모듈로의 인터페이스가 있다.

메모 [유준희4]: There are interfaces to the operating system for real time control and resource management, but also interfaces to other software modules to represent a complete functionality in a system and at least to the hardware, if the application is intended work directly with microcontroller modules.

보다 나은 어플리케이션 소프트웨어의 이식성을 위해 OSEK는 표준화된 구성 정보를 위한 언어를 정의한다. 이 "OIL" (OSEK Implementation Language)이란 언어는 모든 OSEK명세의 개체를 "태스크"와 "알람" 등으로 간편한 설명을 지원한다.

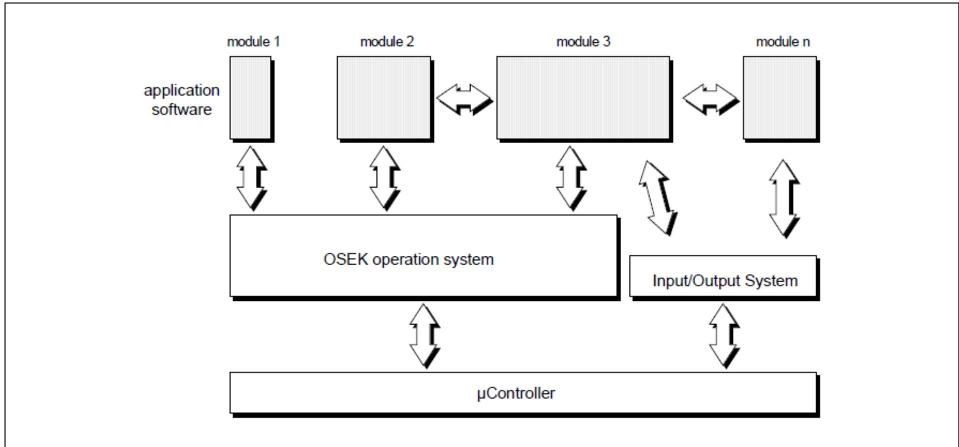

Figure 1-1 Software interfaces inside ECU[1]

하나의 ECU에서 다른 ECU로 어플리케이션 소프트웨어를 이식하는 동안, 개발환경, ECU의 하드웨어 구조 등의 소프트웨어 개발 과정의 특성의 고려가 필요하다. 예를 들면:

- 소프트웨어 개발 가이드라인
- 파일 관리 시스템
- 컴파일러의 데이터 할당과 스택 사용법
- ECU의 메모리 구조
- ECU의 행동 타이밍
- 다양한 마이크로컨트롤러 인터페이스 명세, 예를 들어 포트, A/D 컨버터, 직렬통신과 왓치독 타이머(watchdog timer)
- API 호출의 배치

메모 [유준희5]: Timing behavior of the ECU

이것은 OSEK 명세가 완전한 OSEK 구현을 설명하기엔 충분하지 않다는 것을 의미한다. 구현은 분명한 문서를 제공해야 할 것이다.

이식성의 지원

인증 작업은 서로 다른 구현이 명세서에 적합한가를 보장한다. 이 문서의 챕터 14가 다양한 OSEK 구현 사이에서 어플리케이션의 이식성을 증가시키기 위해 고려해야 할 자세한 사항이다. 여기선 어플리케이션으로의 운영체제 인터페이스만 고려한다.

자동차의 요구사항을 위한 특별한 지원

OSEK 운영체제를 위한 자세한 요구사항은 자동차 컨트롤 유닛을 위한 소프트웨어 개발의 어플리케이션 문맥에서 발생한다. 다음 특징들은 신뢰성, 실시간 가용성, 가격 민감도 등의 요구사항을 제시한다:

[1] OSEK OS allows direct interfacing between application and the hardware.

- OSEK 운영체제는 정적으로 구성되고 기준된다. 사용자는 정적으로 태스크의 수와 자원 그리고 필요한 서비스를 자세히 설명한다.
- OSEK 운영체제의 명세서는 구현이 ROM에서 동작하는 것을 허용한다. 코드는 ROM으로부터 실행될 수 있다.
- OSEK 운영체제는 어플리케이션 태스크의 이식성을 지원한다.
- OSEK 운영체제의 명세서는 운영체제 구현이 가능하도록 자동차의 실시간 요구에 대해 예측가능하고 명문화된 동작을 제공한다.
- OSEK 운영체제의 명세서는 예측 가능한 성능 파라미터의 구현을 허가한다.

1.2 이 문서의 목적

다음 설명은 보통 OSEK 운영체제의 구현을 위해 필수적인 설명으로 간주된다. 이러 것들은 전략과 기능적인 호출의 인터페이스에 대한 일반적인 설명, 의미와 파라미터와 발생할만한 에러코드의 선언에 대해 고려한다.

이 문서는 확실한 범주의 유연성을 남긴다. 한편으로는 장래의 업그레이드에 충분할 만큼의 일반적인 설명이, 다른 한편으로는 명백하게 지정된 상세 구현 영역이 설명에 있다.

구현은 지정된 모든 구현 이슈를 정의해야 한다. 적합클래스는 정밀하게 지시된 구현에 의해 지원되어야 하며 이슈들은 문서화 되어야 하는 지정된 구현에 의해 확인되어야 한다.

OSEK 운영체제의 설명은 장래에 업데이트될 수 있도록 되어야 하고 새로운 요구에 의해 변화될 것이다. 그러므로 레퍼런스 설명으로 사용되어온 OSEK 문서의 버전으로 공식 인증된 각 구현은 자세히 설명되어야 한다. 공식적으로 인증된 OSEK 운영체제의 버전 문서는 x.y[2]로 이름이 지어진다. 이 문서는 "버전 2.2.3"을 나타내고 있다.

메모 [유준희6]: Therefore, each implementation shall specify which officially authorized version of the OSEK description has been used as a reference description.

1.3 이 문서의 구조

다음 글에선, 챕터의 상세한 내용이 간단히 서술된다:

챕터 2, 개요

이 챕터는 OSEK 운영체제의 개념에 대해 간단히 소개한다.

챕터 3, OSEK 운영체제의 구조

이 챕터는 OSEK 운영체제의 구조와 설계원칙에 대한 개관을 보여준다.

챕터 4, 태스크 관리

이 챕터는 다양한 태스크 타입과 스케쥴링 메커니즘과 함께 OSEK 태스크 관리에 대해 설명한다.

챕터 5, 어플리케이션 모드

이 챕터는 어플리케이션 모드와 어플리케이션이 어떻게 지원되는지 설명한다.

---

[2] 버전 업데이트 (철자법 등의 모양 변경은) x.y.z로 이름이 정해질 것이다.

챕터 6, 인터럽트 처리

이 챕터는 OSEK 인터럽트 전략과 다양한 타입의 인터럽트 서비스 루틴의 정보를 제공한다.

챕터 7, 이벤트 메커니즘

이 챕터는 이벤트 메커니즘과 스케쥴링에 의존적인 다양한 동작에 대해 설명한다.

챕터 8, 자원 관리

이 챕터는 OSEK 자원관리와 이점에 대한 논의, OSEK 우선순위 상한화 프로토콜(priority ceiling protocol)의 구현에 대해 설명한다.

챕터 9, 알람

이 챕터는 시간 기반(time-based : 하드웨어 타이머 등)이벤트와 비 시간 기반(non-time-based : 기울어진 각도 측정 등)이벤트를 지원하기 위한 두 단계의 개념을 설명한다.

챕터 10, 메시지

이 챕터는 프로세서 내부 통신을 위한 메시지 핸들링을 설명한다. 모든 메시지 핸들링은 OSEK COM 상세 설명서에 설명되어 있다.

챕터 11, 에러 핸들링, 트레이싱과 디버깅

이 챕터는 중앙화된 에러 핸들링을 달성하기 위한 메커니즘을 설명한다. 이 챕터는 또한 시스템의 초기화와 종료를 위한 서비스에 대해 설명한다.

챕터 12, 시스템 서비스의 설명

이 챕터는 설명을 위한 관례에 대해 설명한다.

챕터 13, 운영체제 서비스의 상세 설명

이 챕터는 사용자가 사용 가능한 모든 운영체제 서비스를 설명한다. 설명의 구조는 모든 서비스에 대해 명확하다; 이것은 사용자가 필요로 하는 서비스의 모든 정보를 담고 있다.

챕터 14, 구현과 상세한 어플리케이션 논제

이 챕터는 운영체제의 서비스, 데이터 타입, 상수를 포함한 모든 상세한 논제의 목록을 제공한다.

챕터 15, 상세 설명서 1.0에서 2.2까지의 변경사항

이 챕터는 버전 1.0부터 버전 2.0, 2.1, 2.1r1과 2.2까지 운영체제 명세의 주 변경사항을 제공한다.

챕터 16, 목록

모든 운영체제 서비스와 figures에 대한 목록

챕터 17, 역사

모든 공식적인 릴리즈의 목록

메모 [유준희7]: This chapter describes the two-stage concept to support time-based events (e.g. hardware-timer) as well as non-time-based events (e.g. angle measurement).

메모 [유준희8]: Tracing

메모 [유준희9]: 뭔가 구리다. 이 챕터를 읽고서 다시 수정할 것

2. 요약

OSEK 운영체제는 다양한 서비스 풀과 프로세싱 매커니즘을 제공한다.
OSEK 운영체제는 사용자가 시스템을 생성할 때 사용자의 구성 지시에 따라 빌드된다.
네 개의 적합클래스(conformance classes)는 OSEK 운영체제의 기능성(functionality)과 가용성(capability)을 고려하는 다양한 요구조건을 만족시키기 위해 설명된다. 따라서, 사용자는 운영체제를 컨트롤 태스크와 목적하는 하드웨어(target hardware)에 맞게 변경할 수 있다. 이 운영체제는 실행시간 이후에 변경될 수 없다.
일정한 적합클래스를 위한 응용프로그램은 OSEK 구현의 같은 클래스에 이식이 가능해야만 한다. 이런 응용프로그램은 서비스의 정의와 응용프로그램의 가용성 범위, 각 적합클래스의 동작에 의해 보장된다. 적합클래스들의 모든 서비스들은 OSEK 운영체제 구현에 따라 정의된 가용성 범위에 의해서만 제공된다.

서비스 그룹은 기능성의 용어에 구조 된다.

태스크 관리 (Task Management)

- 태스크의 활성화와 종료
- 태스크 상태의 관리, 태스크 스위칭

동기화 (Synchronization)

운영체제는 태스크에 영향을 미치는 두 가지의 동기화를 지원해야 한다.

- 자원 관리
  공동으로 사용되는 자원(논리적인)이나 장치에 나눠질 수 없는 연산에 대한 접근 제어, 혹은 프로그램 흐름의 제어
- 이벤트 컨트롤
  태스크 동기화를 위한 이벤트 관리

인터럽트 관리

- 인터럽트 처리를 위한 서비스들

Alarms

- 상대적이고 절대적인 알람들

처리장치 내부 메시지 핸들링 (Intra processor message handling)

- 데이터 교환을 위한 서비스

에러 처리 (Error treatment)

- 다양한 에러에 대한 사용자 지원 메커니즘

메모 [유준희10]: '홀수 같은 짝수' 같은 소릴 하고 있다. 무슨 의미인지는 끝까지 읽고 다시 적어두자.

3 OSEK 운영체제의 구조

3.1 처리 레벨

OSEK 운영체제는 서로 독립적인 응용 프로그램에 대해 기반(basis)을 제공하고, 응용 프로그램의 환경을 처리기 위에 제공해야 한다. OSEK 운영체제는 병행처리시 나타나는 몇몇의 프로세스에 대해 제어된 실시간 실행을 허가해야 한다.

메모 [유준희11]: provides their environment on a processor.

OSEK 운영체제는 사용자에게 정의된 인터페이스 셋(set)을 제공한다. 이러한 인터페이스들은 CPU에서 경쟁하고 있는 엔티티에 의해 사용된다. 이런 엔티티에는 두 가지 타입이 있다.

- 운영체제에 의해 관리된 인터럽트 서비스 루틴
- 태스크들 (기본 태스크들과 확장된 태스크들)

메모 [유준희12]: 매번 느끼는 거지만 정말 마땅한 단어가 없다. '존재'라고 하기도 좀 거시기 하고.

제어장치의 하드웨어 자원은 운영체제 서비스에 의해 관리될 수 있다. 이러한 운영체제 서비스는 응용 프로그램이나 운영체제 내부에서 단일 인터페이스(unique interface)에 의해 호출된다.

메모 [유준희13]: 이런 발번역.. These operating system services are called by a unique interface, either by the application program or internally within the operating system.

OSEK는 3가지 처리 레벨을 정의한다:

- 인터럽트 레벨
- 스케쥴러를 위한 논리 레벨
- 태스크 레벨

태스크 레벨 안에 태스크들이 사용자가 지정한 우선순위에 따라 스케쥴(비선점, 선점, 혼합선점 스케쥴링) 된다. 실행 문맥(run time context)은 실행 시간의 시작을 차지하거나 태스크가 종료되면 해제된다.

메모 [유준희14]: non, full or mixed preemptive scheduling

메모 [유준희15]: The run time context is occupied at the beginning of execution time and is released again once the task is finished.

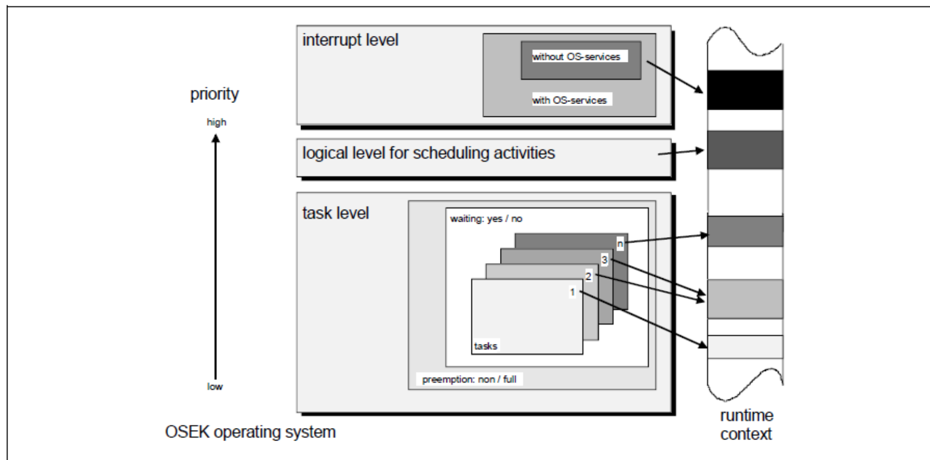

Figure 3-1 Processing levels of the OSEK operating system

다음과 같은 우선순위 규칙이 정해져 있다:

- 인터럽트는 태스크보다 우선순위를 가진다.
- 인터럽트 처리 레벨은 하나나 더 많은 인터럽트 우선순위 레벨로 구성된다.

메모 [유준희16]: 인터럽트 중첩이 된다는 말을 암시하는지?? 끝까지 봐봐야 겠음.

- 인터럽트 서비스 루틴은 인터럽트 우선순위 레벨에 따라 정적으로 지정된다.
- 인터럽트 레벨에 인터럽트 서비스 루틴을 지정하는 것은 구현과 하드웨어 구조를 따른다.
- 태스크 우선순위와 높은 숫자의 자원 우선순위 상한화(resource ceiling-priorities)는 높은 우선순위를 참조한다.
- 태스크의 우선순위는 사용자에 의해 정적으로 지정된다(태스크 우선순위의 의미는 챕터 4.5에서 기술된다).

처리 레벨은 태스크 조작과 연속적인 값의 범위로 간주되는 인터럽트 루틴을 위해 정의된다. 하드웨어 우선순위에 운영체제 우선순위를 맵핑하는 것은 구현 상세이다.

메모 [유준희17]: 대충 무슨 말인지는 알겠는데...
For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities.

스케쥴러에 우선순위를 지정한 것은 단지 직접적인 우선순위를 사용하지 않고 구현하기 위한 논리적인 개념이라는 것을 주목하기 바란다. 추가적으로, OSEK는 태스크간 우선순위 관계의 고려나 세부적인 마이크로프로세서 구조의 하드웨어 인터럽트 레벨에 대한 어떠한 규칙도 규정하지 않는다.

3.2 적합 클래스 (Conformance classes)

시스템을 위한 응용 소프트웨어의 다양한 요구사항과 상세 시스템(프로세서, 메모리)의 다양한 가용성은 운영체제의 여러 가지 특징을 요구한다. 다음 설명은 이러한 운영체제의 특징을 "적합 클래스"(CC)라 설명한다.

적합 클래스는 다음의 목적을 지원하기 위해 존재한다:

- OSEK 운영체제의 쉬운 이해와 논쟁에 편리한 운영체제 특징 그룹을 제공하기 위해
- 미리 정의된 지침에 따른 부분적인 구현을 하기 위해. 이러한 부분적인 구현은 OSEK에 따라 공인된 것이 좋다.
- OSEK와 연관된 특징을 사용하는 응용프로그램의 변경 없이 낮은 기능성의 클래스를 높은 기능성의 클래스로 업그레이드 하는 방법을 만들기 위해

인증되기 위해선, 완전한 적합클래스의 구현이 필수이다. 그러나 시스템 생성은 단지 특정한 어플리케이션이 요구하는 시스템 서비스를 연결시키는 것으로도 충분하다. 적합 클래스는 실행 중에 변경될 수 없다.

적합클래스는 다음과 같은 속성에 의해 결정된다:

- 챕터 4.3에 기술된 것과 같은 태스크 활성화의 다중 요구(Multiple requesting).
- 챕터 4.2에 기술된 것과 같은 태스크 타입.
- 우선순위 수와 같은 수의 태스크들.

만약 다른 방법으로 정해지지 않았다면, 다른 모든 OSEK 특징들은 필수적이다.

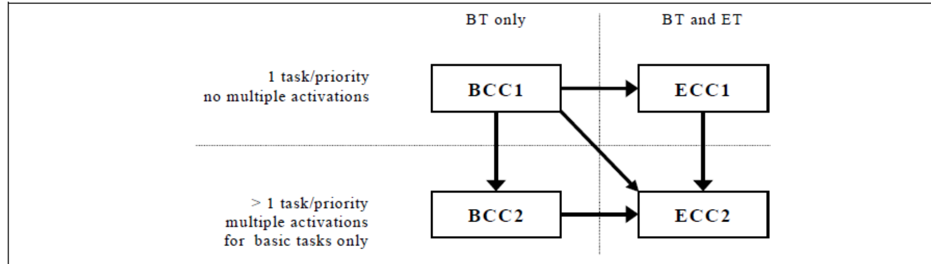

Figure 3-2 Restricted upward compatibility for conformance classes

다음과 같은 적합 클래스들이 정의되어 있다:

- BCC1 (모든 태스크들이 다른 우선순위를 가진다면, 하나의 태스크당 하나의 동작과 하나의 우선순위에 제한되는 기본 태스크)
- BCC2 (BCC1과 같으나 하나의 우선순위에 하나 이상의 태스크가 가능하고, 태스크 활성화의 다중 요구가 허가되는 경우)
- ECC1 (BCC1에 확장된 태스크를 추가)
- ECC2 (ECC1과 비슷하나 하나의 우선순위에 하나 이상의 태스크가 가능하고, 태스크 활성화의 다중 요구가 허가되는 경우)

어플리케이션의 이식성은 기본적인 요구사항들이 초과되지 않은 경우에만 취급한다. 적합 클래스를 위한 최소 요구사항은 figure 3-3에 보여지는 것과 같다.

메모 [유준희18]: 의미가 너무 모호하다. 끝까지 읽고 수정할 것.

메모 [유준희19]: 네. 발번역이죠. The portability of applications can only be assumed if the minimum requirements are not exceeded

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[3]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			

Figure 3-3 The minimum requirements for Conformance Classes

[3] BT = Basic Task, ET = Extended Task

3.3 OSEK OS와 OSEKtime OS의 관계

OSEKtime OS는 시간 시간 동작 구조(time triggered architectures)의 필요에 따라 특별히 맞춰진 운영체제이다. OSEKtime은 OSEK OS가 OSEKtime OS와 공존하는 것을 허가한다. 개념적으로, OSEKtime은 자신의 유휴시간(idle time)을 OSEK가 사용하도록 지정한다. OSEK OS 인터럽트와 태스크들은 OSEKtime OS의 비슷한 엔티티보다 덜 중요하다(낮은 우선순위를 가진다).

OSEK 인터페이스들, 그리고 시스템 콜 정의는 OSEKtime과 OSEK가 같이 있을 경우 변경하지 않는다. 단, OSEK가 유일한 지역적 책임을 지고 OSEKtime이 전반적인 시스템의 책임을 지고 있는 경우에 최소한의 예외가 있다.

이런 상황을 파악하기 위해, OSEKtime에는 만약 OSEKtimeOS와 공존될 것임이 예정되어 있다면 OSEK OS의 구현에 제한을 강요하는 기능정인 정의가 있다. 더 자세한 정보는 OSEKtime OS의 상세를 참조하기 바란다.

메모 [유준희20]: 이런 발번역... 그러나 번역하기 애매한 전문용어임은 분명하다.

메모 [유준희21]: 발번역..
There are minor exceptions with respect to system startup and shutdown due to the fact that OSEKtime is responsible for the overall system whereas OSEK is only locally responsible. These deviations are specifically mentioned within this specification.

메모 [유준희22]:
On top of this, there is functionality defined within OSEKtime which imposes restrictions on the implementation of OSEK OS if it is intended to coexist with OSEKtime OS. For more information, please refer to the specification of the OSEKtime OS.

4. 태스크 관리

4.1 태스크 개념
복잡한 제어 소프트웨어는 각 부분이 실행될 때의 실시간 요구에 따라 편리하게 다시 분할될 수 있어야 한다. 이 부분들은 태스크에 의하여 구현될 수 있다. 태스크는 함수의 실행을 프레임워크로 제공한다. 운영체제는 태스크의 동시성과 비동시성을 제공해야 한다. 스케쥴러는 태스크의 순서를 조직해야 한다.
OSEK 운영체제는 다른 시스템이나 응용 기능이 실행되고 있지 않을 때의 메커니즘을 포함하여 태스크 스위칭 메커니즘(스케쥴러, 챕터 4.4 태스크 스위칭 매커니즘 참조)을 제공한다. 이 매커니즘을 유휴-메커니즘(idle-mechanism)이라 한다. 두 가지의 다른 태스크 개념이 OSEK 운영체제에 의해 제공된다.

- 기본 태스크 (basic tasks)
- 확장 태스크 (extended tasks)

기본 태스크
기본 태스크는 오직 다음과 같은 경우에만 프로세서를 해제한다.

- 종료되었을 경우
- OSEK 운영체제가 보다 높은 우선순위의 태스크로 스위치 되었거나
- 인터럽트가 일어나 프로세서가 인터럽트 서비스 루틴으로 스위치 되었을 경우

확장 태스크
확장 태스크는 태스크를 대기 상태로 만드는 운영체제 시스템콜 대기 이벤트(Wait Event)의 허용으로 기본 태스크와 구분된다(챕터 7의 이벤트 메커니즘, 챕터 13.5.3.4의 WaitEvent 참조). 대기 상태는 실행 중인 확장 태스크의 종료 없이 프로세서에서 해제되고 낮은 우선순위의 태스크가 재할당 되는 것을 허가한다.

운영체제 관점에서, 확장 태스크의 관리는 기본 태스크 관리보다 복잡하고 더 많은 시스템 자원을 요구한다.

4.2 태스크 상태 모델
다음은 태스크 상태와 두 가지 태스크 타입 사이의 상태 전이를 설명한다.
프로세서가 한 번에 한 태스크의 명령만을 수행한다면, 몇 개의 태스크가 같은 시간에 프로세서를 사용하기 위해 경쟁을 하는 동안 몇 가지 상태로 바뀌어야만 한다. OSEK 운영체제는 언제든 필요 시 태스크 상태 전이와 관련된 태스크 문맥의 저장과 복원에 대한 책임이 있다.

4.2.1 확장 태스크
확장 태스크는 4가지 태스크 상태를 가진다:
실행 중 (running) CPU에 태스크가 할당 되어 태스크의 명령이 수행 가능한 실행상태이다. 많은 태스크들이 다른 모든 상태(실행상태를 제외한)에 있는 사이에 어느 시점에서

메모 [유준희23]: Complex control software can conveniently be subdivided in parts executed according to their real-time requirements.

메모 [유준희24]: Basic tasks only release the processor, if

메모 [유준희25]: 의역인데 의미는 통한다.
Extended tasks are distinguished from basic tasks by being allowed to use the operating system call *WaitEvent*, which may result in a *waiting* state (see chapter 7, Event mechanism, and chapter 13.5.3.4, *WaitEvent*)

메모 [유준희26]: 얘들이 allocation 대신 assigned를 쓴다.

메모 [유준희27]: in principal을 생략하였다. 무슨 의미로 쓰인거야 대체. 원리에 의하면? 규정에 의하면?

든 하나의 태스크만이 이 상태에 있을 수 있다.

준비 (ready) 실행상태로의 전이를 위한 모든 기능적인 선행조건이 존재하고, 오로지 프로세서에 할당되기만을 기다리는 상태. 스케쥴러는 다음에 실행될 준비 태스크를 결정한다.

대기 중 (waiting) 태스크가 적어도 하나의 이벤트를 기다려야 해서 계속 수행되지 못할 때(챕터 7 이벤트 메커니즘 참조).

중지 (suspended) 중지된 태스크는 수동적이며, 활성화 될 수 있다.

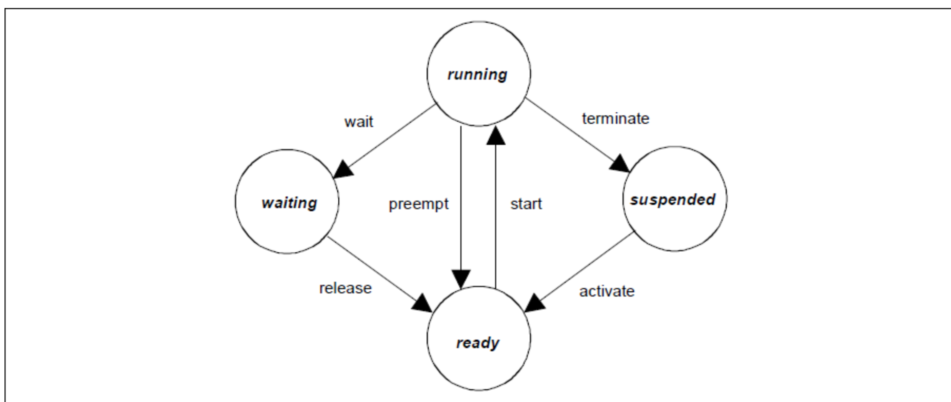

Figure 4-1 Extended task state model

*혼동을 막기 위해 설명 부분만 해석하였습니다. By kailieu

전이	이전 상태	새 상태	설명
activate	*suspended*	*ready*	새 태스크는 시스템 서비스에 의해 *준비* 상태로 설정된다. OSEK 운영체제 시스템은 태스크의 첫 번째 명령을 수행할 것을 보장한다.
start	*ready*	*running*	*준비* 태스크는 스케쥴러의 실행에 의해 선택된다.
wait	*running*	*waiting*	대기 중 상태로의 전이는 시스템 서비스에 의해 일어난다. 연산을 계속하기 위해선, *대기 중* 태스크는 이벤트를 필요로 한다.
release	*waiting*	*ready*	*대기 중* 태스크가 기다리던 적어도 하나 이상의

메모 [유준희28]: In the *running* state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

메모 [유준희29]: 자기 스스로 활성화 되거나 수행이 될 수 없다는 소리.

메모 [유준희30]: A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.

			이벤트가 발생하였을 경우
preempt	*running*	*ready*	스케쥴러가 다른 태스크를 시작하기로 결정했을 때. *실행 중* 태스크가 *준비* 상태로 바뀐다.
terminate	*running*	*suspended*	*실행 중* 태스크가 스스로 전이를 일으켜 시스템 서비스에 의해 *중지* 상태로 변경

Figure 4-2 States and status transitions for extended tasks

태스크의 종료는 스스로 종료("self-termination")할 때만 가능하다. 이 제한은 운영체제의 복잡도를 줄여준다. *중지* 상태를 *대기 중* 생태로 바로 바꾸기 위해선 어떠한 준비도 필요 없다. 이 상태 전이는 중복과 스케쥴러에 복잡성을 증대 시킬 것이다.

4.2.2 기본 태스크

기본 태스크의 상태 모델은 확장 태스크 상태 모델과 거의 흡사하다. 다른 점은 기본 태스크에는 *대기 중* 상태가 없다는 것이다.

실행 중 (running) CPU에 태스크가 할당 되어 태스크의 명령이 수행 가능한 *실행* 중 상태이다. 많은 태스크들이 다른 모든 상태(실행상태를 제외한)에 있는 사이에 어느 시점에서든 하나의 태스크만이 이 상태에 있을 수 있다.

메모 [유준희31]: In the *running* state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

준비 (ready) 실행상태로의 전이를 위한 모든 기능적인 선행조건이 존재하고, 오로지 프로세서에 할당되기만을 기다리는 상태. 스케쥴러는 다음에 실행될 준비 태스크를 결정한다.

중지 (suspended) 중지된 태스크는 수동적이며, 활성화 될 수 있다.

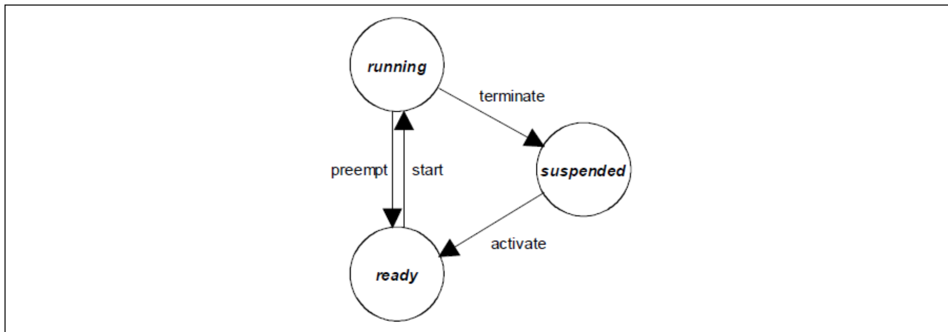

Figure 4-3 Basic task state model

전이	이전 상태	새 상태	설명
activate	*suspended*	*ready*[4]	새 태스크는 시스템 서비스에 의해 *준비* 상태로 설정된다. OSEK 운영체제 시스템은 태스크의 첫 번째 명령을 수행할 것을 보장한다.
start	*ready*	*running*	*준비* 태스크는 스케쥴러의 실행에 의해 선택된다.
preempt	*running*	*ready*	스케쥴러가 다른 태스크를 시작하기로 결정했을 때. *실행 중* 태스크가 *준비* 상태로 바뀐다.
terminate	*running*	*suspended*	*실행 중* 태스크가 스스로 전이를 일으켜 시스템 서비스에 의해 *중지* 상태로 변경

Figure 4-4 States and status transitions for basic tasks

4.2.3 태스크 타입의 비교

기본 태스크는 *대기 중* 상태를 가지지 않는다. 따라서 동기화 포인트는 태스크의 시작과 끝에 있다. 어플리케이션 부분의 내부 동기화 포인트는 하나 이상의 기본 태스크에 의해 구현된다. 기본 태스크의 이점은 실행 시간 문맥 (run time context : RAM)이 고려된 알맞은 요구사항이다. 확장 태스크의 이점은 하나의 태스크에서 동기화 요구의 활성 문제 없이 확실한 작업을 할 수 있다는 것이다. 다음 처리에 대해 현재 정보가 없으면, 확장 태스크는 *대기 중* 상태로 전환된다.

대기 중 상태는 이벤트 시그널을 받을 때나 요구되는 데이터 혹은 이벤트의 업데이트가 발생시 벗어날 수 있다. 확장 태스크는 기본 태스크에 비해 더 많은 동기화 포인트를 가진다.

4.3. 태스크의 활성화

태스크의 활성화는 운영체제의 서비스인 *ActivateTask*나 *ChainTask* 사용에 의해 발생한다. 활성화 이후 태스크는 첫 번째 문장을 실행할 준비가 된다.

OSEK 운영체제는 태스크의 시작 시 C언어와 비슷한 인자의 전달을 지원하지 않는다. 이러한 인자들은 메시지 통신(챕터 10, 메시지 참조)나 전역 변수들을 통해 전달되어야 한다.

태스크 활성화의 다중 요구

[4] 태스크 활성화는 다중 활성화가 요구된 태스크의 상태를 바로 바꾸지 않는다. 태스크가 중지되어 있지 않으면, 활성화는 기록만 되고 나중에 발생한다.

메모 [유준희32]: Task activation will not immediately change the state of the task in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later

메모 [유준희33]: A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.

메모 [유준희34]: 발번역.
이것만 보면 참 잘도 이해 되겠다.

메모 [유준희35]: 간단히 말하면... 장치가 준비되어 있지 않거나, 다른 태스크의 데이터 처리가 끝나지 않았거나 등등등

메모 [유준희36]: 드디어 요게 뭔지 나오는 군하... 간단히 말하면 기본 태스크를 여러 번 활성화 시키는 것.

적합 클래스에 의존하는 기본 태스크는 한번만 활성화 되거나 여러 번 활성화 될 수 있다. "다중 활성화 요구"(Multiple requesting of task activation)는 OSEK 운영체제가 이미 활성화된 기본 태스크를 병렬적으로 기본 태스크의 활성화를 수신하고 기록하는 것을 의미한다.

병렬적인 다중 요구의 횟수는 시스템이 생성될 때 기본 태스크의 지정된 속성으로 정의되어야 한다. 만약 다중 요구의 최대 횟수에 도달하지 않았다면, 요구는 큐에 저장된다. 기본 태스크의 활성화 요구는 활성화 순서에 따라 큐에 저장된다.

4.4 태스크 스위칭 메커니즘

일반적인 순차 프로그래밍과 다르게 멀티태스킹 원칙은 운영체제가 동시에 다양한 태스크를 수행하는 것을 허가한다. 그러므로 스케쥴링 규칙은 명확하게 정의되어야 한다(챕터 4.6 스케쥴링 규칙 참조).

어떤 태스크를 시작하고 OSEK 운영체제의 내부 활동을 유발할지 결정하는 존재를 스케쥴러라 부른다. 스케쥴러는 스케쥴 규칙에 따라 태스크 스위칭이 가능할 때 작동된다. 스케쥴러는 태스크에 의해 점유 및 해제되는 자원처럼 간주될 수 있다. 그러므로, 태스크는 스스로가 해제될 때까지 태스크 스위칭을 피하기 위해 스케쥴러를 점유할 수 있다. 보다 자사한 사항은 챕터 8.3, 자원으로서의 스케쥴러를 참고.

4.5 태스크 우선순위

스케쥴러는 *실행 중* 상태로 바꾸어질 다음 *준비* 태스크의 태스크 우선순위 (precedence)의 기반을 결정한다.

0값은 태스크의 가장 낮은 우선순위로 정의된다. 따라서 큰 숫자일수록 높은 우선순위를 의미한다.

효율을 높이기 위한 동적인 우선순위 관리는 지원되지 않는다. 그러므로 태스크의 우선순위는 실행 중에 바꿀 수 없는, 정적으로 정의된다. 그러나 실제로는 운영체제가 태스크를 지정된 높은 우선순위로 다루기도 한다. 이 내용은 챕터 8.5 OSEK 우선순위 상한화 프로토콜을 참조.

동일한 우선순위의 태스크는 적합 클래스 BCC2와 ECC2에 의해 지원된다. 챕터 3.2, 적합 클래스 참조.

같은 레벨의 우선순위의 태스크의 시작은 활성화된 순서에 달려있다. *대기 중* 상태의 확장 태스크들은 같은 우선순위의 순차 태스크 (subsequent task)의 시작을 막지 않는다.

선점된 태스크는 현재 우선순위 리스트의 첫 번째 (가장 오래된)*준비* 태스크로 간주될 수 있다.

*대기*상태로부터 벗어난 태스크는 *준비* 우선순위 큐의 마지막 (가장 새로운) 태스크로 취급된다.

메모 [유준희37]: Batch 시스템

메모 [유준희38]: Accordingly the priority of a task is defined statically, i.e. the user cannot change it at the time of execution.

메모 [유준희39]: 우선순위 큐에 쌓여있는 다른 태스크 순서들. 그림을 참고하면 이해가 빠를 듯

메모 [유준희40]: Preempted task : 걍 다음 태스크라 생각하자.

Figure 4-5는 각 우선순위 레벨에서의 스케쥴러 사용법의 구현 예를 보여준다. 서로 다른 우선순위의 몇몇 태스크가 준비 상태에 있다.; 예를 들어 우선순위 3의 세 태스크, 우선순위 2의 한 태스크, 우선순위 1의 한 태스크와 우선순위 0의 두 태스크. 요구된 순서에 따라 가장 오래 기다린 태스크는 각 큐의 바닥에 보인다. 프로세서는 방금 태스크를 처리하고 종료되었다. 스케쥴러는 다음 처리될 태스크를 선택한다(우선순위 3의 첫 번째 큐). 우선순위 2인 태스크들은 높은 우선순위의 모든 태스크들이 ***실행 중*** 상태와 ***준비***상태에서 시작된 후 종료되거나 ***대기 중*** 상태로 전이되어 큐에서 제거 되어야 처리될 수 있다.

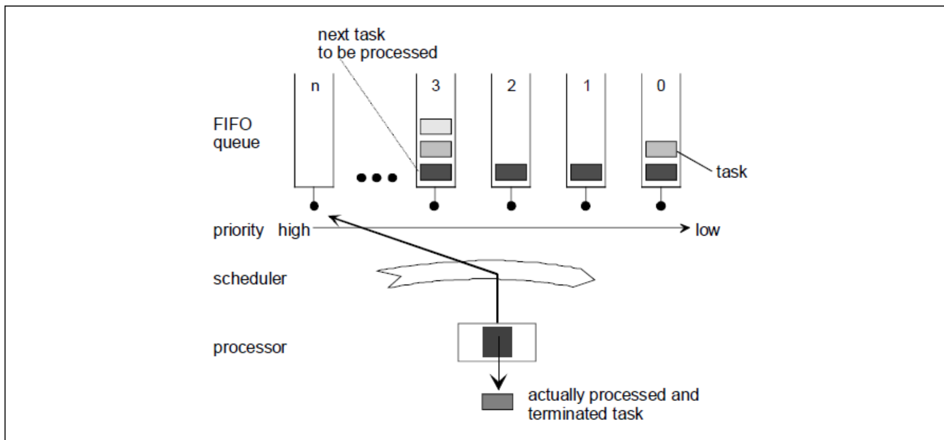

Figure 4-5 Scheduler: order of events

다음 기초적인 단계들은 다음 처리될 태스크를 결정하는데 필요하다:

- 스케쥴러는 *준비/실행 중* 상태의 모든 태스크를 찾아야 한다.
- *준비/실행 중* 상태에 있는 태스크 셋 (set) 을 스케쥴러는 태스크 셋들을 높은 우선순위로 결정한다.
- *준비/실행 중* 상태에 있는 태스크 셋 (set) 과 높은 우선순위 중에 스케쥴러는 가장 오래된 태스크를 찾는다.

4.6 스케쥴링 규칙

4.6.1 완전 선점 스케쥴링 (Full preemptive scheduling)

메모 [유준희41]: 아 완전 어색해...

완전 선점 스케쥴링은 현재 *실행 중*인 태스크가 운영체제에 의해 미리 정한 트리거 조건이 발생되는 특정 명령에 의해 재스케쥴(reschedule)되는 것을 말한다. 완전 선점 스케쥴링은 높은 우선순위의 태스크가 ***준비***되는 대로 ***실행 중*** 태스크를 ***준비*** 상태로 바꿀 것이다. 태스크 문맥은 선점하는 태스크가 선점된 지점에서 저장된다.

메모 [유준희42]: 앞에서도 trigger가 나왔었는데, 도저히 마땅한 말이 없다. 방아쇠? 발생자?

메모 [유준희43]: The task context is saved so that the preempted task can be continued at the location where it was preempted.

완전 선점 스케쥴링을 사용하면, 수행 시간은 낮은 우선순위 태스크의 실행시간과 관계가 없다. 이러한 제한들은 문맥을 저장하기 위한 메모리 공간의 증가와 태스크 간의 동기화를 위한 특징들의 복잡한 향상에 관여된다. 이러한 각각의 태스크는 이론적으로 어느 위치에서, 동기화 되어야 하는 다른 태스크들과 공동으로 사용되는 데이터의 접근 시 재스케쥴 될 수 있다.

Figure 4-6에서, 낮은 우선순위의 태스크 T2는 높은 우선순위인 T1의 스케쥴링을 지연시키지 않는다.

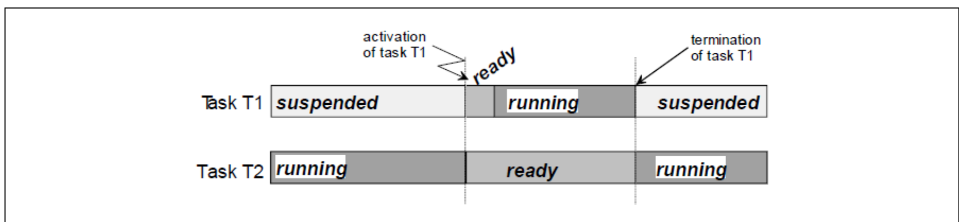

Figure 4-6 Full preemptive scheduling

완전 선점 시스템의 경우, 사용자는 *실행 중* 태스크의 선점을 지속적으로 기대할 수 있다. 만약 태스크 조각이 선점 되지 않아야 한다면 시스템 서비스 *GetResource*를 통해 스케쥴러를 일시적으로 막을 수 있다.

정리하자면, 재스케쥴링은 다음의 모든 경우에서 발생된다:

- 태스크의 성공적인 종료 (시스템 서비스 *TerminateTask*, 챕터 13.2.3.2 참조)
- 후속 태스크를 명백히 활성 시킨 태스크의 성공적인 종료 (시스템 서비스 *ChainTask*, 챕터 13.2.3.3 참조)
- 태스크 수준에서의 태스크 활성화 (예 : 시스템 서비스 *ActivateTask*, 챕터 13.2.3.1, 메시지 알림 메커니즘, 알람 만료, 태스크 활성화가 정의되어 있다면, 챕터 9.2 참조)
- 명시된 *대기* 호출이 *대기 중* 상태로 전이 시킨 경우 (확장 태스크 전용, 챕터 13.5.3.4 시스템 서비스 *WaitEvent* 참조).
- 태스크 수준에서의 *대기 중* 이벤트 설정 (예 : 챕터 13.5.3.1 시스템 서비스 *SetEvent*, 메시지 알림 메커니즘, 알람 만료, 이벤트 설정이 정의된 경우, 챕터 9.2 참조)
- 태스크 레벨에서 자원의 해제 (시스템 서비스 *ReleaseResource*, 챕터 13.4.3.2 참조)
- 인터럽트 레벨에서 태스크 레벨로의 리턴

인터럽트 서비스 루틴 중에는 재스케쥴링이 일어나지 않는다 (figure 3-1 참조).

스케쥴링 전략인 '완전 선점 스케쥴링을'(full preemptive scheduling)을 사용하는 어플리케이션들은 시스템 서비스인 *Schedule*을 필요로 하지 않는다. 하지만 다른 스케쥴링 규칙은 이 시스템 서비스를 사용해야 한다. 다른 스케쥴링 규칙에도 불구하고 어플리케이션 이식성을 확보하기 위해서, 사용자는 적합하다고 추측되는 곳에서 시스템 서비스 *Schdule*을 통해 재스케쥴하는 것을 강화할 수 있습니다.

4.6.2 비 선점 스케쥴링

비 선점 스케쥴링은 정의된 시스템 서비스에 의해서만 (재스케쥴링이 명시된 시점) 스위칭이 일어나는 경우를 의미한다.

메모 [유준희44]: Successful termination of a task with explicit activating of a successor task

메모 [유준희45]: the user can enforce a rescheduling via the system service *Schedule* at locations where he/she assumes a correct assignment of the CPU.

비 선점 스케뷸링은 태스크의 요구가 가능한 제한된 타이밍을 이용한다. 특히 우선순위가 낮은 *실행 중* 태스크의 비 선점 구간은 재스케쥴링 되었을 때 오는 높은 우선순위 태스크의 시작을 지연시킨다.

Figure 4-7에서, 낮은 우선순위 태스크 T2는 재스케쥴링 되었을 때 (이 경우엔 태스크 T2의 종료) 오는 높은 우선순위 태스크를 지연시킨다.

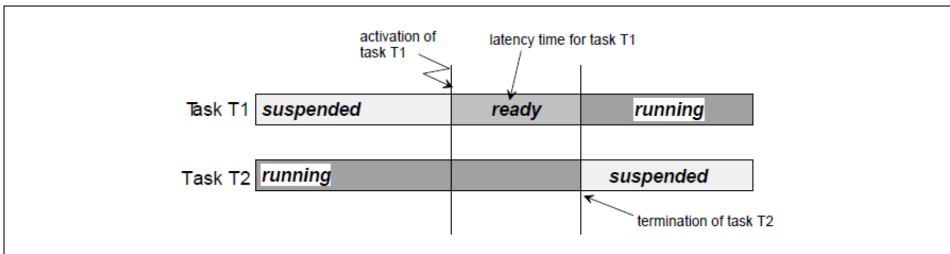

Figure 4-7 Non preemptive scheduling

재스케쥴링의 시점

비 선점 태스크의 경우, 재스케쥴링은 정확히 다음과 같은 경우에 일어난다:

- 태스크의 성공적인 종료 (시스템 서비스 *TerminateTask*, 챕터 13.2.3.2, 참조)
- 후속 태스크를 명백히 활성 시킨 태스크의 성공적인 종료 (시스템 서비스 *ChainTask*, 챕터 13.2.3.3 참조)
- 뚜렷한 스케쥴러의 호출 (시스템 서비스 *Schedule*, 챕터 13.2.3.4 참조)
- *대기 중* 상태로 전이되었을 때 (시스템 서비스 *WaitEvent*, 챕터 13.5.3.4 참조)[5]

비 선점 시스템의 구현은 재스케쥴링을 호출하는 것을 가장 높은 태스크 프로그램 레벨(태스크의 서브함수 : subfunction 가 아닌)[6]에서만 호출될 수 있도록 운영체제 서비스를 규정해야 한다.

4.6.3 태스크 그룹

운영체제는 태스크가 태스크 그룹을 정의하는 것에 의해 선점과 비 선점 스케쥴링의 양상을 결합하는 것을 허가한다. 그룹 내 가장 높은 우선순위와 같거나 낮은 우선순위를 가진 태스크들은 마치 비 선점 태스크들처럼 동작한다: 재스케쥴링은 챕터4.6.2에 설명된 재스케쥴의 시점에서만 일어날 것이다. 그룹 내 가장 높은 우선순위보다 높은 우선순위를 가지는 태스크들은 마치 선점 태스크처럼 동작할 것이다 (챕터 4.6.1 참조).

챕터 8.7은 내부 자원을 하용하여 그룹을 정의하는 메커니즘을 설명한다. 내부 자원 개념의 가장

[5] WaitEvent의 호출은 어떤 이벤트가 이벤트 마스크에 WaitEvent가 이미 설정시켰을 경우에 *대기 중*상태로 바뀌지 않는다. 이 경우 WaitEvent는 재스케쥴링을 유발하지 않는다.

[6] 이러한 스케쥴링 시점의 태스크 스위칭은 일반적으로 저장을 위한 태스크 문맥 정보가 보다 적게 요구된다.

메모 [유준희46]: Non preemptive scheduling imposes particular constraints on the possible timing requirements of tasks

메모 [유준희47]: Successful termination of a task with explicit activating of a successor task

메모 [유준희48]: The call of WaitEvent does not lead to a *waiting* state if one of the events passed in the event mask to WaitEvent is already set. In this case WaitEvent does not lead to a rescheduling.

메모 [유준희49]: 무슨 말인가 하면, 함수가 마구 호출된 후의 태스크 스택 크기는 호출된 함수의 개수에 비례하여 클 것이다. 이걸 태스크 스위칭 시 저장하고자 하면 많은 공간을 차지하게 되므로 효율적이지 못하다. 밑의 주석은 발번역.

메모 [유준희50]: 번역도 힘들고 이해도 힘든 문단이었다. 일단 태스크 그룹 내의 가장 높은 우선순위가 태스크 그룹의 우선순위를 대표한다는 것 같다. 그룹 내의 태스크들은 서로 비 선점 방식이고, 그룹의 우선순위보다 높은 그룹 밖의 태스크와는 선점방식으로 스케쥴링이 된다는 의미로 보인다. 적절한 예를 만들지 못해 이해가 어려웠다.

일반적인 사용 예는 비 선점 태스크 들이다; 가장 높은 태스크 우선순위가 특별히 지정된 내부 자원을 이용하는 태스크 들이다.

4.6.4 혼합 선점 스케쥴링

같은 시스템에 선점 태스크와 비 선점 태스크들이 섞여 있다면, 이러한 규칙을 "혼합 선점"(mixed preemptive) 스케쥴링이라 한다. 이런 경우의 스케쥴링 규칙은 실행 중인 태스크의 선점 속성에 의해 결정된다. 만약 실행 중인 태스크가 선점될 수 없으면 비 선점 스케쥴링이 일어난다. 실행 중인 태스크가 선점 가능하면 선점 스케쥴링이 일어난다.

비 선점형 태스크의 정의는 완전 선점 운영체제에서도 가능하다.

- 만약 태스크의 실행 시간이 태스크 스위칭의 시간과 같은 시간이 걸린다면.
- 만약 램이 경제적으로 사용되어 태스크 문맥의 저장을 위한 공간을 제공한다면.
- 태스크가 선점되지 않는다면.

많은 어플리케이션들은 완전 선점 운영체제를 편리하게 하기 위해 긴 실행시간에 적은 병행 태스크 만을 포함한다. 또한 비 선점 스케쥴링에서 정해진 시간의 짧은 태스크들은 더욱 효율 적일 것이다. 이러한 구성을 위해 혼합 선점 스케쥴링 규칙이 타협안으로서 개발되었다 (챕터 14.2.4 의 디자인 힌트 참조).

4.6.5 스케쥴링 규칙의 선택

소프트웨어 개발자나 시스템을 완성하는 사람은 태스크의 우선순위와 태스크의 선점가능성을 지정하는 것으로 태스크의 순서를 결정해야 한다.
태스크 타입(기본 혹은 확장)은 태스크의 스케쥴링 타입(선점 혹은 비 선점)에 독립적이다. 완전 그러므로 선점 시스템은 기본 태스크를, 그리고 비 선점 시스템은 확장 태스크를 포함할 수 있다. 만약 운영체제 서비스가 실행 중이라면, 선점과 컨택스트 스위칭은 서비스의 종료까지 지연될 것이다.

4.7 태스크의 종료

OSEK 운영체제에서, 태스크는 오직 스스로만 종료할 수 있다("자기 종료 : self-termination").
OSEK운영체제는 실행 중인 태스크의 종료 바로 후에 특정 목적의 태스크가 활성 되도록 *ChainTask*서비스를 제공한다. 자기 체이닝은 우선순위 큐의 마지막 원소로 새로이 활성화된 태스크를 밀어 넣는다.
각 태스크는 태스크의 코드 끝에서 스스로 종료해야 한다. *TerminateTask*나 *ChainTask*를 호출하지 않고 태스크를 종료하는 것은 엄격히 금지되며 정의하지 않은 동작을 유발한다.

메모 [유준희51]: Non preemptable tasks are the most common usage of the concept of internal resources; they are tasks with a special internal resource of highest task priority assigned.

메모 [유준희52]: if RAM is to be used economically to provide space for saving the task context,
왠지 문맥상 '충분한'의 의미가 있어 보인다.

메모 [유준희53]: 원문에서는 chaning itself로 되어있다.

메모 [유준희54]: 자기 스스로 suspend에서 ready로 바꿔서 큐에 넣는다는 의미인 듯.

5. 응용 모드 (Application modes)

응용 모드는 OSEK 운영체제가 다양한 연산 모드에서 수행하는 것을 허락하기 위해 설계되었다. 지원되는 응용 모드의 최소 수는 한 개 이다. 이것은 오직 완전한 상호 배제 연산 모드를 위한 것이다. 연산의 두 개의 상호 배제모드의 예는 '라인의 끝' 연산(end-of-line) 프로그래밍과 노말 연산(operation)이 될 수 있다.

5.1. 응용 모드의 범위 (Scope of application modes)

많은 ECU들은 팩토리 테스트(Factory test), 플래쉬 프로그래밍이나 일반 연산 같은 완전히 독립적인 어플리케이션들을 실행할 수 있다. 응용 모드는 다양한 조건에 따른 소프트웨어 구동을 위한 구조화의 방법이고 완전히 분리된 시스템의 개발을 위한 깔끔한 메커니즘이다. 전형적으로 각 응용모드는 다양한 모드에서 태스크나 ISR을 실행하는 것에 제한이 없음에도 불구하고 가지고 있는 모든 태스크의 부분 집합(subset), ISR, 알람과 타이밍 조건들을 사용한다. 다른 조건에서 같은 기능이 다시 필요하다면 태스크/ISR/알람을 공유하는 것을 권장한다. 만약 기능이 완전히 같지 않다면 응용 모드가 동적으로 체크되거나 분리된 태스크가 정의되어도 실행시간과 자원 사이에 균형(trade-off)이 필요하다.

시스템의 생성을 하는 것과 최적화를 생각하면, 응용 모드는 고려되는 OS 객체의 개수를 줄이는 것에 효과적이다.

5.2. 시작 성능 (Start up performance)

시작 성능은 자동차 어플리케이션에서 ECU가 일반적인 연산을 하는 중에 초기화(reset)조건이 발생했을 경우 안전 결정적 문제(safety critical issue)이다.

결과적으로 코드는 어플리케이션 모드가 빨리 될 수 있도록 정해져야 한다. 시작 시, 시스템 서비스(Figure 11-2 참조)를 사용하지 않는 사용자 코드는 모드를 결정하고 API-서비스인 *StartOS*[7]의 인자처럼 넘겨진다. 모드를 결정하기 위해 적은 상태(pin states)나 조건 평가를 쉽게 하는 것을 추천한다. 모드는 커널이 시작되기 전에 결정되어야만 하고 그 결과로 나온 코드는 이식성이 없다. 길거나 복잡한 시작 프로시저는 가급적 피해야 하는 것이 분명하다.

어플리케이션 모드는 *StartOS*에게 운영체제가 자동으로 알맞은 태스크의 부분집합과 알람을 시작하는 것의 허용권을 넘긴다. 어플리케이션 모드에 태스크의 자동시작과 알람을 지정하는 것은 OIL 파일에서 정적으로 만든다.

5.3. 어플리케이션 모드를 위한 지원

적합 클래스의 서브셋에 대한 어플리케이션 모드의 제한은 없다. 이것은 모든 클래스에게 요구된다.

종료(shutdown)기능에 영향이 없다.

실행 시간 중에 어플리케이션 모드의 변경은 지원되지 않는다.

[7] OSEK와 OSEKtime이 공존하는 시스템의 경우, 응용 모드는 OSEKtime이 사용하는 모드로 넘겨진다.

메모 [유준희55]: Totally mutually exclusive. 의미가 상호 배제인 것 같아 그렇게 번역하였습니다.

메모 [유준희56]: 발번역... Typically each application mode uses its own subset of all tasks, ISRs, alarms and timing conditions, although there is no limitation to having a task or ISR running in different modes.

메모 [유준희57]: 이해 안됬음...

메모 [유준희58]: In case of a system where OSEK and OSEKtime coexist, the application mode passed to OSEKtime is used.

메모 [유준희59]: 발번역의 점입가경이다... The assignment of autostart tasks and alarms to application modes is made statically in the OIL file.

메모 [유준희60]: There is no restriction of application modes to a subset of conformance classes. It is required for all classes. There is no impact on the shutdown functionality. Switching between application modes at runtime is not supported. 바꿔 놓으니 더 어색하다.

6. 인터럽트 처리

인터럽트 처리를 위한 함수 (Interrupt Service Routine: ISR)은 두 개의 ISR 카테고리로 나뉘어진다:

ISR 카테고리 1 ISR은 운영체제의 서비스를 이용하지 않는다.[8] ISR의 종료 후엔, 인터럽트가 태스크 관리에 영향을 주지 않았을 경우, 정확히 인터럽트가 발생했던 명령이 계속 처리된다. 이 카테고리의 ISR들은 적은 부담(overhead)을 가진다.

ISR 카테고리 2 OSEK 운영체제는 지정된 사용자 루틴을 위한 실행시간 환경의 준비를 위해 ISR-프레임을 제공한다. 시스템 생성 중에 사용자 루틴은 인터럽트로 지정된다.

인터럽트 서비스 루틴 중에, OSEK 운영체제 서비스의 사용은 Figure 12-1에 따라 제한된다.

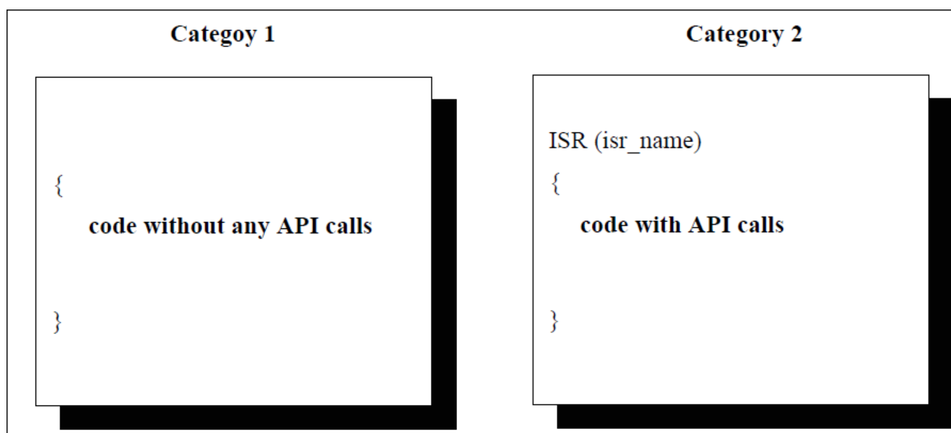

Figure 6-1 ISR categories of the OSEK operating system

ISR 내부에선 재스케쥴링이 발생하지 않는다. 재스케쥴링은 만약 선점이 가능한 태스크가 인터럽트되고 다른 인터럽트가 활성화 되어 있지 않다면 ISR 카테고리 2의 종료에서 발생한다. 구현은 태스크가 OSEK 스케쥴링 포인트에 따라 태스크들이 실행되는 것을 보장한다(챕터 4.6.1 완전 선점 스케쥴링 참조). 이것을 위해 구현은 구상하는 시간에 모든 카테고리의 ISR을 위한 인터럽트 우선순위 레벨이나 발생을 확인하는 방법(혹은 둘 다)에 대한 제한들을 규정해야 한다 (챕터 14.2.3.1, 다른 카테고리의 인터럽트 중첩 참조).

메모 [유준희61]: To achieve this the implementation may prescribe restrictions concerning interrupt priority levels for ISRs of all categories and/or perform checks at configuration time

인터럽트 우선순위들의 최대 숫자는 구현과 사용되는 컨트롤러에 의존한다. 인터럽트의 스케쥴링은 하드웨어에 의존적이고 OSEK에 의해 지정되지 않는다. 인터럽트는 태스크가 스케쥴러에 의해 계획되듯이 하드웨어에 의해 계획된다. 인터럽트 우선순위 레벨을 고려하는 것은 14.2.3.1이 설명에 따라 제한될 것이다. 인터럽트는 태스크를 중단할 수 있다(선점 가능한 태스크이든 아니든). 만약 태스크가 인터럽트 루틴에 의해 활성화 되었다면 태스크는 모든 인터럽트 루틴의 종료 후에 스케쥴링 된다.

인터럽트 서비스 루틴 내에서 사용 가능한 시스템 서비스는 Figure 12-1에서 나열된 것만 가능하

[8] 인터럽트 활성화와 비활성화 시스템 서비스는 예외다. Figure 12-1 참조

다.

빠른 비활성화/활성화 API-함수

OSEK는 모든 인터럽트를 비활성화 하고 (챕터 13.3.2.1, *EnableAllInterrupts*, 13.3.2.2, *DisableAllInterrupts*, 13.3.2.3, *ResumeAllInterrupts*와 13.3.2.4 *SuspendAllInterrupt* 참조) 카테고리 2의 모든 인터럽트를 비활성화하는 (챕터 13.3.2.3, *ResumeOSInterrupts*와 13.3.2.6, *SuspendOSInterrupts* 참조) 빠른 함수를 제공한다. 일반적으로 짧은 임계영역의 보호를 위해 사용된다. "suspend/disable"가 "resume/enable"와 짝지어진 보호된 임계영역 내에선 인터럽트에서 복귀하는 것을 허가하지 않는다. 운영체제 서비스 콜은 Suspend- 그리고 Resume- 짝, 나아가 *SuspendOSInterrupts / ResumeOSInterrupts*짝이나 *SuspendAllInterrupts / ResumeAllInterrupts*짝으로만 허가한다.

메모 [유준희62]: "suspend/disable" have to have a matching "resume/enable"

7. 이벤트 메커니즘

이벤트 메커니즘은

- 동기화의 방법이다.
- 확장 태스크를 위해서만 제공된다.
- 태스크의 상태 전이 초기화와 *waiting* 상태에서 상태를 전이.

메모 [유준희63]: 깔끔하게 안떨어진다.
initiates state transitions of tasks to and from the *waiting* state.

이벤트는 운영체제에 의해 관리되는 객체이다. 이것들은 독립적인 객체는 아니지만 확장 태스크에 할당이 가능하다. 각 확장 태스크는 정해진 이벤트의 숫자를 가진다. 이 태스크는 이벤트들의 소유자(owner)로 칭한다. 각각의 이벤트는 그 소유자와 이벤트의 이름(혹은 마스크)에 의해 확인된다. 이벤트들은 그들을 소유한 확장 태스크로 이진정보통신(binary information communication)을 위해 사용될 수 있다. 이벤트의 의미는 타이머의 만료, 자원의 가용성, 메시지의 수신 등의 시그널링(signaling) 등등 어플리케이션에 의해 정의된다.
다양한 옵션은 제시된 태스크가 이벤트의 소유자이거나 그것을 필요로 하지 않는 다른 확장 태스크에 의해 이벤트를 조작하는 것을 가능하게 한다. 오직 소유자만이 자신이 가지고 있는 이벤트의 제거(clear)와 수신(=셋팅 : setting)이 가능하다.
이벤트는 *waiting*상태를 *ready*상태로 바꾸는 확장 태스크의 전이를 위한 기준이다. 운영체제는 셋팅과 제거, 그리고 이벤트의 호출신호와 이벤트의 발생을 기다리는 서비스를 제공한다.

메모 [유준희64]: The operating system provides services for setting, clearing and interrogation of events and for waiting for events to occur.

모든 태스크나 카테고리 2의 ISR은 정지되지 않은 확장태스크를 위한 이벤트 설정을 할 수 있다. 그리고 이런 이벤트를 통해 상태 변화를 확장태스크에게 알린다.
이벤트의 수신자는 아무 상태의 확장태스크가 될 수 있다. 따라서, 인터럽트 서비스 루틴이나 기본 태스크는 이벤트를 기다릴 수 없다. 이벤트는 해당 이벤트의 소유 태스크로 인해서만 해제(clear)될 수 있다. 확장 태스크는 그들이 가진 이벤트만 해제 하는 것에 반해 기본 태스크는 이벤트 해제를 위한 운영체제 서비스를 사용할 수 없다.
*waiting*상태의 확장 태스크는 적어도 하나의 기다리고 있던 이벤트가 발생되면 *ready*상태로 풀린다. 만약 *running* 확장 태스크가 이벤트를 기다리기 위해 시도하는 중 이 이벤트가 이미 일어났다면, 태스크는 *running*상태로 남는다.

메모 [유준희65]: The operating system provides services for setting, clearing and interrogation of events and for waiting for events to occur. 번역이 구린데, 곱씹어 보면 당연한 말이다.

Figure 7-1은 완전 선점 스케쥴링에서 이벤트들의 셋팅으로 확장 태스크 T1이 높은 우선순위를 가졌을 때의 확장 태스크의 동기화를 설명한다.

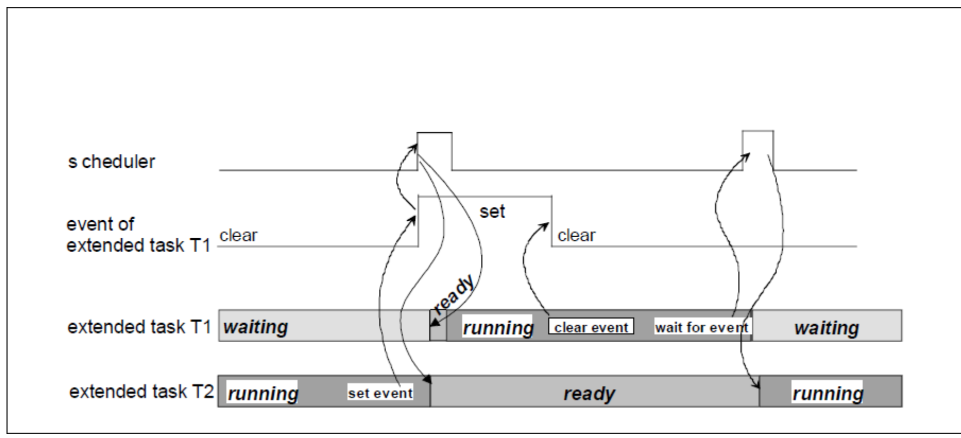

Figure 7-1 Synchronisation of preemptable extended tasks

Figure 7-1은 이벤트의 셋팅에 의해 절차가 받은 영향을 설명한다: 태스크 T1이 이벤트를 기다린다. 태스크 T2가 T1을 위한 이벤트를 셋팅한다. 스케쥴러가 활성화 된다. 그 후에, T1은 *waiting*상태에서 *ready*상태로 전이된다. T1의 우선순위가 높기 때문에 태스크 스위칭이 일어나 T2는 T1에 의해 선점된다. T1은 이벤트를 리셋한다. 그 후에 T1은 다시 이벤트를 기다리고 스케쥴러는 T2의 실행을 재개한다.

비 선점 스케쥴링에선 이벤트가 설정된 후에 곧바로 재스케쥴링이 일어나지 않는다(T1이 높은우선순위 확장 태스크인 Figure 7-2 참조).

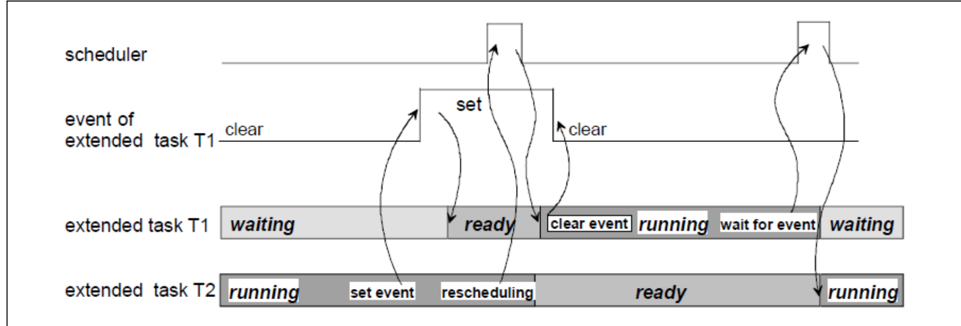

Figure 7-2 Synchronisation of non preemptable extended tasks

8 자원 관리

자원 관리는 엔티티의 관리 (스케쥴러), 프로그램 순서, 메모리나 하드웨어 영역 등의 공유된 자원에 다른 우선순위를 가진 몇몇 태스크의 동시(concurrent) 접근을 위해 사용된다.
자원 관리는 모든 적합 클래스들을 위해 필수적이다.
자원 관리는 태스크와 인터럽트 서비스 루틴의 동시 접근을 위해 선택적으로 확장될 수 있다.
자원 관리는 다음과 같은 사항을 보장한다:

- 두 태스크가 동시에 같은 자원을 점유할 수 없다.
- 우선순위 역전은 일어나지 않는다.
- 이런 자원들을 사용함으로 인해 교착상태가 발생하지 않는다.
- 자원에 접근하는 것의 결과로 *waiting*상태가 생기지 않는다.

만약 자원 관리가 인터럽트 레벨로 확장되었다면 추가적으로 다음을 보장한다.

- 두 태스크나 인터럽트 루틴은 같은 자원을 동시에 점유할 수 없다.

다음과 같은 경우에 자원관리의 기능은 유용하다.

- 선점 태스크
- 시 선점 태스크, 만약 사용자가 어플리케이션 코드를 다른 스케쥴링 규칙으로 지시한 경우도 포함.
- 태스크와 인터럽트 서비스 루틴 사이의 자원 공유
- 인터럽트 루틴 사이의 자원 공유

사용자는 태스크나 인터럽트로 인한 중지를 막기 위해, 그들은 재스케쥴링이 일어나지 않도록 하는 enable/disable interrupt 운영체제 서비스를 이용할 수 있다 (챕터 6, 인터럽트 처리와 챕터 13.3, 인터럽트 핸들링 참고).

8.1 점유된 자원에 접근 시 동작

OSEK OS는 OSEK 우선순위 상한 프로토콜(챕터 8.5 참조)을 규정한다. 따라서, 태스크나 인터럽트가 점유된 자원으로 접근을 시도하는 상황은 발생하지 않는다.
만약 자원의 개념이 태스크나 인터럽트를 동등하게 하는데 사용된다면 OSEK 운영체제는 인터럽트 서비스 루틴의 수행이 해제되기 전에 인터럽트 서비스 루틴이 필요한 모든 자원을 점유해야 수행을 보장한다.
OSEK는 같은 자원에 대한 중첩된 접근을 엄격히 제한한다. 드물게 중첩 접근이 필요한 경우, 두 번째 자원의 사용을 첫 번째 자원의 동작처럼 할 것을 권한다 ('연결된 자원'이라고도 한다: linked resources).

메모 [유준희66]: If the resource concept is used for co-ordination of tasks and interrupts the OSEK operating system ensures also that an interrupt service routine is only processed if all resources which might be occupied by that interrupt service routine during its execution have been released. 문장이 길어서 어순이 꼬인다. '필요한'은 문맥상 필요할 것 같아 추가하였다.

메모 [유준희67]: 영문으로도 이해가 안 되었다.

8.2 자원 사용 시 제한사항

*TerminationTask, ChainTask, Schedule, WaitEvent*는 자원이 점유되어 있는 동안 불려질 수 없다.
인터럽트 서비스는 자원이 점유된 채로 종료될 수 없다.
하나의 태스크 내에서 여러 자원을 점유한 경우, 사용자는 LIFO 원칙(스택 같은)에 따라 자원을 요구하거나 해제해야만 한다.

8.3 자원으로서의 스케쥴러

만약 태스크가 다른 태스크의 선점으로부터 스스로 보호되고 싶다면, 스케쥴러를 잠글 수 있다. 스케쥴러는 모든 태스크가 접근 가능한 자원처럼 다뤄진다. 그러므로 자동적으로 RES_SCHEDULER라는 이름의 자원이 생성된다.
인터럽트는 RES_SCHEDULER 자원의 상태에 독립적으로 수행되고 수신된다.

메모 [유준희68]: Interrupts are received and processed independently of the state of the resource RES_SCHEDULER. However, it prevents the rescheduling of tasks.

8.4 동기화 메커니즘의 일반적인 문제

8.4.1 우선순위 역전의 설명

일반적인 동기화 메커니즘의 문제 – 예를 들어 세마포어의 사용 – 은 우선순위 역전의 문제와 관련되어 있다.
우선순위 역전은 낮은 우선순위 태스크가 높은 우선순위의 실행을 지연시키는 것을 의미한다.
OSEK는 우선순위 역전을 피하기 위해 *OSEK Priority Ceiling Protocol*(챕터 8.5 참조)를 규정한다.
Figure 8-1는 일반적인 두 태스크가 하나의 세마포어 접근하는 것에 대한 순차처리를 설명한다 (완전 선점 시스템에서, 태스크 T1이 가장 높은 우선순위를 가진다).
낮은 우선순위를 가진 태스크 T4는 세마포어 S1을 점유하고 있다. T1은 T4를 선점하고 같은 세마포어를 요구한다. 세마포어 S1이 이미 점유되었기 때문에, T1은 *waiting*상태로 들어간다. 이제 낮은 우선순위 T4는 T1과 T4사이의 태스크들에 의해 선점되고, 인터럽트 당한다. 모든 낮은 우선순위 태스크들이 종료되어야만 T1이 수행 될 것이고 세마포어 S1이 해제될 것이다. T2와 T3가 세마포어 S1을 사용하지 않지만, T1을 그들의 실행시간만큼 지연시킬 것이다.

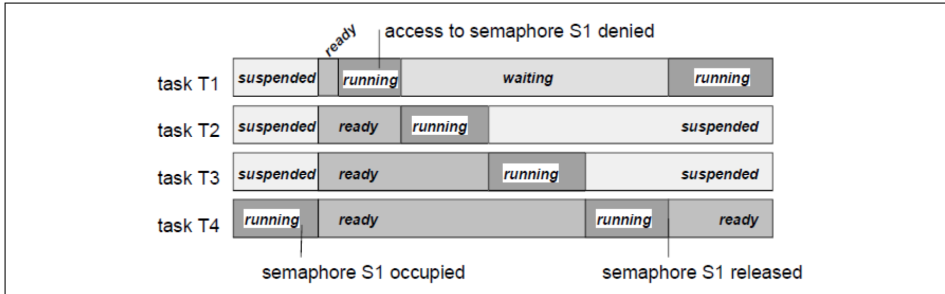

Figure 8-1 Priority inversion on occupying semaphores

8.4.2 교착상태

세마포어의 사용 같은 일반적인 동기화 메커니즘의 또 다른 전형적인 문제는 교착상태의 문제이다. 이런 경우 교착상태의 의미는 잠긴 자원들을 서로 무한히 기다리기 때문에 태스크 수행의 불가능성을 의미한다.
다음의 시나리오는 교착상태의 결과이다 (Figure 8-2 참조).
태스크 T1이 세마포어 S1을 점유하고 이벤트를 기다리는 등의 이유로 계속적인 진행을 할 수 없다. 그러므로 낮은 우선순위 태스크 T2가 실행 중인 상태로 전이된다. 이것은 세마포어 S2를 점유하고 있다. 만약 T1이 다시 실행 준비가 되고 세마포어 S2를 점유하기 위해 시도하면, 이는 다시 대기 중 상태로 들어간다. 만약 지금 T2가 세마포어 S1의 점유를 시도하면, 이것이 교착상태이다.

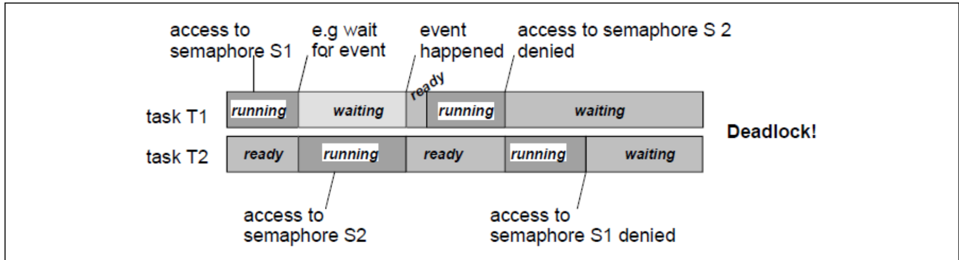

Figure 8-2 Deadlock situation using semaphores

8.5 OSEK 우선순위 상한 프로토콜

우선순위 역전과 교착상태의 문제를 피하기 위해 OSEK 운영체제는 다음 동작을 요구한다.

- 시스템 생성 시, 자원은 각각 자신만의 상한 우선순위를 정적으로 지정되어 가진다. 상한 우선순위는 적어도 이 자원에 접근하는 모든 태스크들이나 연계된 어떠한 다른 자원들 중 가장 높은 우선순위와 같거나 높아야 한다. 상한 우선순위는 자원에 접근하지 않는 태스크들의 가장 낮은 우선순위 보다, 그리고 자원에 접근하는 모든 태스크들의 가장 높은 우선순위보다 높아야 한다.
- 만약 태스크가 자원을 요구하면, 그리고 태스크의 현재 우선순위가 자원의 상한 우선순위 보다 낮으면 태스크의 우선순위는 자원의 상한 우선순위로 올려진다.
- 만약 태스크가 자원을 해제하면, 태스크의 우선순위는 자원을 요구하기 전에 동적으로 지정되었던 우선순위로 돌아간다.

우선순위 상한은 자원 우선순위와 같거나 낮은 속성의 태스크를 위해 가능한 만큼의 시간 지연을 결과로 가져온다. 이 지연은 낮은 우선순위 태스크에 의해 점유된 자원의 최장 시간에 의해 제한된다.

같은 자원을 요구하는 태스크들은 실행 중인 태스크보다 낮거나 같은 우선순위 때문에 *running*상태로 들어갈 수 없을 것이다. 만약 자원이 태스크의 해제로 점유되면, 자원을 요구한 다른 태스크는 *running*상태가 될 수 있다. 선점 태스크에선 이것이 재스케쥴링이 되는 포인트다.

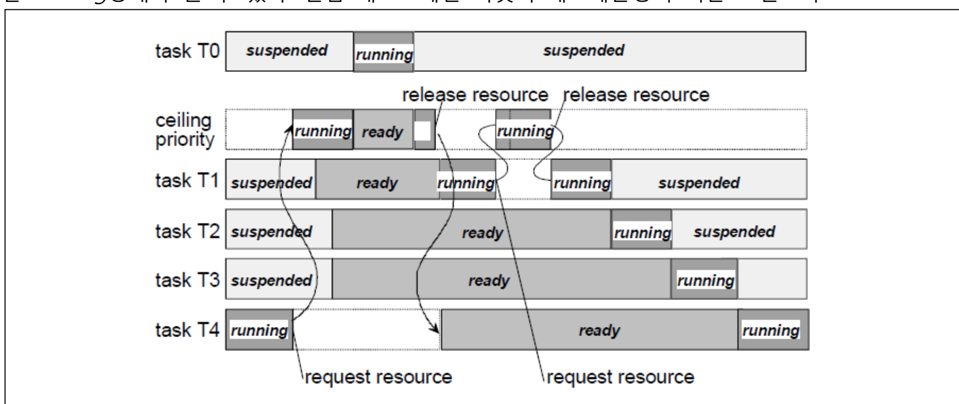

Figure 8-3 Resource assignment with priority ceiling between preemptable tasks.

메모 [유준희69]: Execution time을 말하는 건진 잘 모르겠음

메모 [유준희70]: Priority ceiling results in a possible time delay for tasks with priorities equal or below the resource priority. This delay is limited by the maximum time the resource is occupied by any lower priority task.

Figure 8-3에서 보여지는 예제는 우선순위 상한의 메커니즘을 설명한다. 태스크 T0가 가장 높은, 그리고 태스크 T4가 가장 낮은 우선순위를 가진다. 태스크 T1과 T4는 같은 자원에 접근을 원한다. 시스템은 우선순위 역전이 발생하지 못하게 하는 것을 깔끔하게 보여준다. 높은 우선순위 태스크 T1이 T4로 인한 자원 점유의 최대 지속시간보다 짧게 기다린다.

메모 [유준희71]: no unbounded priority inversion is entailed.

메모 [유준희72]: 말이 애매한데, 이전 figure 8-1을 보면서 이해하도록 하자.

8.6 인터럽트 레벨로의 OSEK 우선순위 상한 프로토콜 확장

인터럽트 레벨로 자원관리의 확장은 선택사항이다.

자원에서 사용되는 자원의 상한 우선순위를 결정하기 위해선, 가상 우선순위들이 인터럽트에 지정된 태스크들의 속성들 보다 높아야 한다. 소프트웨어 우선순위와 하드웨어 인터럽트 레벨의 조작은 구현에 달렸다.

- 시스템 생성 시, 자원에게 각각의 상한 우선순위를 정적으로 지정한다. 상한 우선순위는 자원에 접근하는 모든 태스크들과 인터럽트 루틴들, 그리고 연관된 자원들의 가장 높은 우선순위 보다 같거나 높아야 한다. 상한 우선순위는 자원에 접근하지 않는 모든 태스크와 인터럽트 루틴들의 가장 낮은 우선순위 보다 낮아야 하고 같은 시간에 자원에 접근하는 모든 태스크와 인터럽트 루틴의 가장 높은 우선순위보다 높아야 한다.
- 만약 태스크나 인터럽트 루틴이 자원을 요구하고 현재 우선순위가 자원의 상한 우선순위 보다 낮으면, 태스크나 인터럽트의 우선순위는 자원의 상한 우선순위로 올려진다.
- 만약 태스크나 인터럽트 루틴이 자원을 해제하면, 태스크나 인터럽트의 우선순위는 자원을 요구하기 이전에 동적으로 지정되었던 우선순위로 바뀐다.

실행중인 태스크나 인터럽트 루틴이 점유한 자원의 점유를 원하는 태스크나 인터럽트 루틴은 지금 수행중인 태스크나 인터럽트 루틴의 우선순위보다 낮거나 같은 우선순위 때문에 수행되지 않는다. 만약 태스크에 의해 점유된 자원이 해제되면, 점유를 원하는 다른 태스크나 인터럽트 루틴이 실행될 수 있다. 선점이 가능한 태스크는 태스크의 새 우선순위가 인터럽트의 가상 우선순위가 아니라면 이 때가 재스케쥴링을 하는 시점이다.

메모 [유준희73]: For preemptable tasks this is a point of rescheduling if the new priority of the task is not the virtual priority of an interrupt.

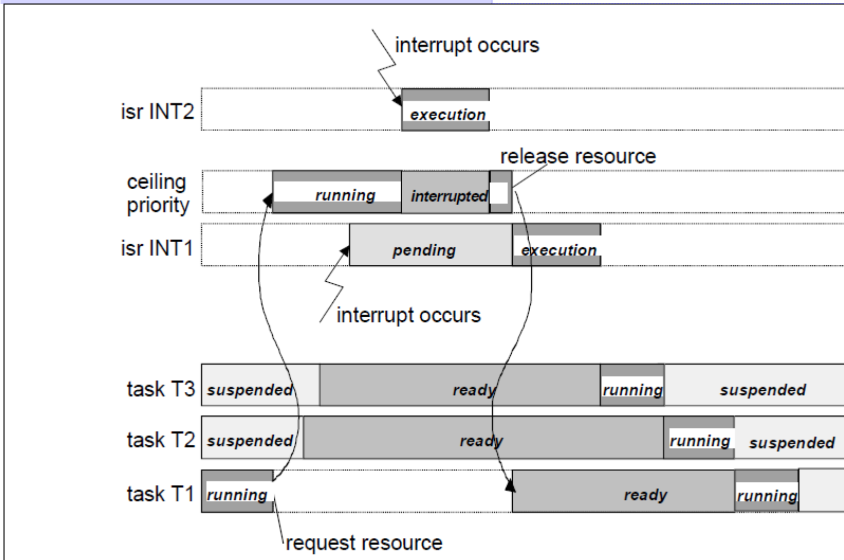

Figure 8-4 Resource assignment with priority ceiling between preemptable tasks and interrupt services routines.

예제 8-4의 예제는 다음 시나리오를 보여준다:
선점 가능한 태스크 T1이 실행 중이고 인터럽트 서비스 루틴 INT1과 공유된 자원을 요구한다. 태스크 T1은 보다 높은 우선순위 태스크 T2와 T3보다 활성화 시킨다. OSEK 우선순위 상한 프로토콜에 의해 태스크 T1은 계속 수행된다. 인터럽트 INT1이 발생한다. OSEK 우선순위 상한 프로토콜에 의해 인터럽트 INT1은 대기(pending)되고 태스크 T1이 계속 수행된다. 인터럽트 INT2가 발생한다. 인터럽트 서비스 루틴 INT2는 태스크 T1을 중단시키고 실행된다. INT2가 완료된 후에 T1이 계속 된다. 태스크 T1이 자원을 해제한다. 인터럽트 서비스 루틴 INT1이 실행된다. INT1이 완료된 후 태스크 T3가 수행된다. 태스크 T3