*Transactions of KSAE, Vol. 17, No. 3, pp.127-141 (2009)*

Copyright © 2009 KSAE
1225-6382/2009/099-17

# 동적 메모리 사용 감소를 위한 OSEK OS 커널 구현 메커니즘

임 진 택* · 금 한 홍 · 박 지 용 · 홍 성 수

서울대학교 전기컴퓨터공학부

# OSEK OS Kernel Mechanisms for Reducing Dynamic Memory Usage

Jintack Lim* · Hanhong Keum · Jiyong Park · Seongsoo Hong

*School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea*

***(Received 11 November 2008 / Accepted 24 January 2009)***

**Abstract** : While the ever-increasing complexity of automotive software systems can be effectively managed through the adoption of a reliable real-time operating system (RTOS), it may incur additional resource usage to a resultant system. Due to the mass production nature of the automotive industry, reducing physical resources used by automotive software is of the utmost importance for cost reduction. OSEK OS is an automotive real-time kernel standard specifically defined to address this issue. Thus, it is very important to develop and exploit kernel mechanisms such that they can achieve minimal resource usage in the OSEK OS implementation. In this paper, we analyze the task subsystem, resource subsystem, application mode and conformance classes of OSEK OS as well as the OSEK Implementation Language (OIL). Based on our analysis, we in turn devise and implement kernel mechanisms to minimize the dynamic memory usage of the OSEK OS implementation. Finally, we show that our mechanisms effectively reduce the memory usage of OSEK OS and applications.

**Key words** : OSEK OS, RTOS(실시간 운영체제), Memory optimization(메모리 최적화), Automotive software(차량용 소프트웨어)

## 1. 서 론

최근 지능형 자동차의 개발이 가속화 되면서 차량 내의 전자 장치의 수가 급증하고 있다. 이에 비례하여 전자 장치를 제어하는 소프트웨어의 크기와 복잡도도 급격히 늘어나고 있다. 실제로 최고급 차량의 경우 수 천만 줄의 코드를 가진 소프트웨어가 탑재되기도 한다.[1] 초기의 차량용 소프트웨어는 몇 개의 반복문과 인터럽트 서비스 루틴들로 구성될 정도로 간단하였지만,[2] 최근에는 점차 소프트웨어로 수행해야 할 기능들이 다양하고 복잡해짐에 따라 기존의 단순한 구조로 차량 소프트웨어를 작성하고 유지 보수하는 것이 어려워졌다. 이 문제를 해결하기 위해서는 실시간 운영체제(RTOS), 미들웨어, 컴포넌트 기반 설계 등을 도입하여 소프트웨어의 복잡성을 낮추어야 한다.[3]

이러한 필요성에 따라 1990년대 중반 유럽의 자동차 생산자들(BMW, Daimler-Benz 등)과 부품업체들(Bosch, Siemens 등)은 서로 연합하여 OSEK/VDX[4,5]라는 프로젝트 그룹을 조직하고, RTOS, 네트워크 관리, 통신 미들웨어의 표준을 발표하였다. OSEK OS는 이 중 RTOS의 표준이다. 이미 존재하는 상용 RTOS를 사용하지 않고 새롭게 표준을 만든 이유는 다양한 생산자와 부품업체의 응용 프로그램이 특정한 상용 RTOS 에 종속되지 않도록 하여 이

*Corresponding author, E-mail: jtlim@redwood.snu.ac.kr

식성을 높이기 위해서이다.

이식성에 더하여 자동차용 소프트웨어 작성 시 중요하게 고려되어야 할 부분 중에 하나가 자원(CPU, RAM, ROM) 사용의 효율성이다. 자동차 산업이 가지고 있는 대량 생산이라는 특성으로 말미암아 각 ECU의 자원요구량이 조금만 증가하여도 전체 생산 비용에는 막대한 영향을 끼치기 때문이다.

OSEK OS와 응용 프로그램이 사용하는 자원의 양은 OS 커널의 구현에 따라 달라진다. 이는 OSEK OS 표준 문서가 OS와 응용 프로그램간의 인터페이스와 기능만을 명시하고, 기능의 구현은 OS 개발자에게 맡기기 때문이다. 만약 개발자가 기능적으로는 문제없는 OS를 구현하여도 그 구현 과정에서 자원의 효율적인 사용을 고려하지 않았다면, 그 결과물은 불필요하게 자동차의 생산 비용을 증가시킬 수 있다. 따라서 OSEK OS 표준을 충족시키면서도 OS와 응용 프로그램이 사용하는 자원을 절감할 수 있는 커널 구현 기법을 고안하는 것은 중요한 문제이다. 이에 본 논문에서는 ECU의 자원 중 RAM의 사용량을 절약할 수 있는 커널 메커니즘을 제안한다. 본 논문에서 제안된 메커니즘은 Power Architecture를 사용하는 MPC5554[6]를 탑재한 평가 보드에 구현되었다.

본 논문의 구성은 다음과 같다. 2장에서는 OSEK OS와 타 RTOS의 차이점을 분석하고, 3장에서는 이 분석 결과를 바탕으로 메모리 사용량을 절감할 수 있는 세 가지 메커니즘을 제안한다. 4장에서는 샘플 응용 프로그램을 이용하여 제안된 메커니즘을 실제로 적용하는 방법을 기술하고, 메커니즘 사용 전/후의 메모리 사용을 실험을 통하여 정량적으로 분석한다. 5장에서는 관련 연구를 살펴보고 6장에서는 결론을 맺는다.

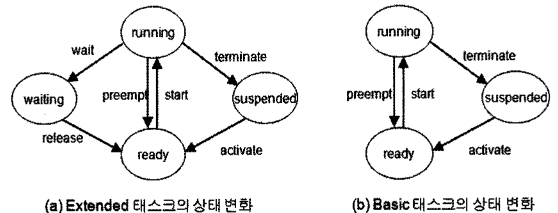
## 2. OSEK OS의 특성 분석

본 논문에서 제안하는 메커니즘은 OSEK OS 표준을 토대로 하기 때문에 본 장에서는 OSEK OS를 특징짓는 요소들을 분석한다. 먼저 OSEK OS를 구성하는 태스크, 알람, 이벤트, 리소스, 인터럽트의 5개 서브시스템 중에서 메모리 사용과 밀접한 태스크와 리소스 서브시스템을 자세히 분석한다. 이어서 응용 프로그램의 정적 설정, 응용 프로그램에 따른 OS 프로파일 선택, 상황에 따른 응용 프로그램의 선택 등의 특징을 살펴본다.

### 2.1 두 가지 태스크 타입을 지원하는 태스크 서브시스템

본 절에서는 태스크의 속성인 타입, 다중 활성화, 우선순위, 선점 허용 여부 등을 살펴보고 태스크의 속성을 반영한 스케줄링 방식을 살펴본다.

일반적인 RTOS가 태스크의 타입을 구분하지 않는데 반하여 OSEK OS는 태스크를 Extended 태스크와 Basic 태스크로 나눈다. 그 기준은 태스크가 WaitEvent() API를 호출하는지의 여부이다. 태스크가 WaitEvent() API를 호출하면 태스크 스스로 수행을 중지하고 응용 프로그램의 다른 구성 요소(태스크, 알람)가 자신을 깨워 줄 때까지 기다리게 되며, 이 상태를 대기(waiting) 상태라고 한다. Extended 태스크는 WaitEvent() API를 호출하는 태스크로 대기 상태와 더불어 종료(suspended), 실행(running), 준비(ready) 상태를 가진다. 반면 Basic 태스크는 대기 상태를 제외한 나머지 상태를 가진다. Fig. 1의 (a)와 (b)는 Extended 태스크와 Basic 태스크의 상태와 상태 변화를 보여준다.



(a) Extended 태스크의 상태 변화 (b) Basic 태스크의 상태 변화

Fig. 1 태스크의 상태 변화

Basic 태스크는 다중 활성화(multiple activation)가 가능하다. 활성화란 태스크를 종료 상태에서 준비 상태로 변화 시키는 동작이며, 다중 활성화란 태스크가 종료 상태에 있지 않을 때 활성화 요청을 받으면, 이를 OS가 기억했다가 태스크가 종료한 후에 곧바로 활성화 시키는 동작을 의미한다.

OSEK OS는 태스크의 우선순위를 기반으로 스케줄링하며, 각 태스크는 응용 프로그래머가 지정한 고정 우선순위(fixed priority)를 갖는다. 우선순위의

숫자가 클수록 높은 우선순위를 뜻한다. 스케줄링이 일어나는 시점은 각 태스크의 선점 허용 여부에 따라 달라진다. 선점을 허용하지 않는 태스크가 실행중이면 자발적으로 CPU를 양보하거나 태스크가 종료할 때만 스케줄링을 한다. 선점을 허용하는 태스크는 위의 경우에 더하여 어느 태스크라도 준비 상태로 변할 때 스케줄링을 수행한다.

## 2.2 명시적/암시적 동기화 메커니즘을 제공하는 리소스 서브시스템

리소스 서브시스템은 공유 자원을 사용하는 태스크의 동기화를 지원한다. 즉, 여러 태스크들이 하나의 자원을 공유하는 경우, 동시에 자원이 이용되어 자원의 일관성을 보장하지 못하는 경쟁 상태(race condition)에 놓이는 것을 방지하는 메커니즘을 제공한다. 이 메커니즘은 OSEK PCP(Priority Ceiling Protocol)라고 한다. OSEK PCP는 태스크가 자원을 사용할 때 현재 태스크의 우선순위를 자원을 사용하려는 다른 어떤 태스크들 보다 높게 변경하여 자원의 접근을 막는 방식이다. 구체적으로, 태스크 뿐만 아니라 자원에도 우선순위(ceiling priority)를 부여하고, 태스크가 자원을 사용하기 시작할 때 해당 자원의 우선순위를 상속받는 방식이다. 하지만 태스크의 현재 우선순위가 자원의 우선순위보다 높은 경우에는 현재 우선순위를 그대로 유지한다. 자원의 우선순위는 Table 1에서 제시된 조건을 만족하는 우선순위 중 임의의 값으로 정적으로 결정된다. 이는 응용 프로그램의 각 태스크가 어떤 자원을 이용하는지 모두 정적으로 결정되기 때문에 가능하며, 정적 설정의 특징은 2.3절에서 분석한다.

Fig. 2는 3 개의 태스크와 1 개의 자원으로 구성된 응용 프로그램에서 자원의 우선순위를 지정하는 예제를 보여준다. 우선순위가 각각 1과 7인 태스크 T1과 T2가 자원 myRes를 공유하므로, 조건 1에 의해서

Table 1 자원의 우선순위 지정 방법

| |
|---|
| **조건 1**: 자원을 이용하는 모든 태스크의 우선순위 중 가장 높은 우선순위 보다 크거나 같아야 한다. |
| **조건 2**: 자원을 이용하지 않고, 조건 1을 만족하는 범위에 있는 모든 태스크의 우선순위 중 가장 낮은 우선순위 보다 작아야 한다. |

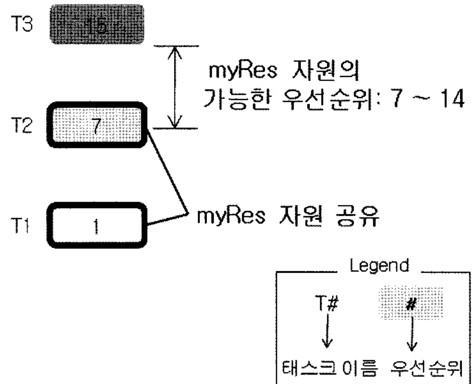



Fig. 2 자원의 우선순위 지정 예제

자원의 우선순위는 7 이상이 된다. 그리고 우선순위 15인 태스크 T3이 자원을 사용하지 않으므로 조건 2에 의해서 자원의 우선순위는 15 미만이 되어야 한다. 따라서 myRes의 우선순위는 7부터 14 까지의 임의의 값으로 결정할 수 있다.

Fig. 3은 Fig. 2의 태스크 T3, T2, T1이 시간에 따라 실행되는 모습을 보여준다. 이 예에서 자원 myRes는 우선순위 8을 지정받는 것으로 결정되었다고 가정한다. T1이 GetResource라는 API를 통해 myRes를 획득하면 T1의 우선순위가 myRes의 우선순위인 8로 변경된다. 따라서 우선순위가 7인 T2가 T3에 의하여 활성화 되더라도, T1이 자원 사용을 완료하고 본래의 우선순위인 1로 복귀하기 전까지는 준비 상태에 머무른다. T2 역시 실행 중에 myRes 자원을 사용하면 우선순위가 8로 변하는 것을 볼 수 있다.

OSEK OS의 자원은 일반 자원과 내부 자원(internal Resource)의 두 종류로 나뉜다. 일반 자원은

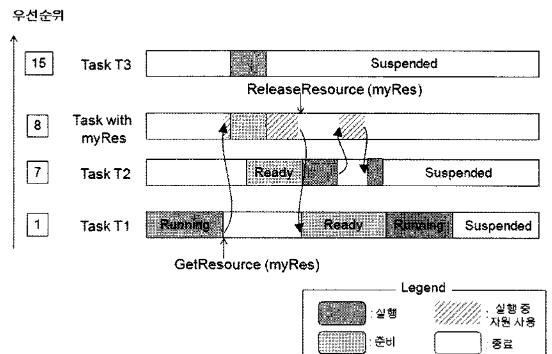



Fig. 3 자원 사용에 따른 우선순위의 변화

위의 예제와 같이 명시적인 API(GetResource, ReleaseResource)를 호출하여 사용한다. 반면에 내부 자원은 API를 통하여 사용할 수 없으며 오직 OS에 의하여 자동으로 관리된다.

내부 자원을 이용하는 태스크가 실행을 시작 할 때에는 OS에 의해 자동으로 내부 자원을 획득하게 된다. 내부 자원을 사용할 때에도 일반 자원을 사용하는 경우와 마찬가지로 태스크가 자원의 우선순위를 상속받는다. 내부 자원을 가진 태스크가 실행 중 우선순위가 더 높은 태스크에게 선점당하는 경우에 내부 자원을 반납하지 않으며 우선순위도 계속 유지한다. 따라서 내부자원을 공유하는 태스크들은 서로를 선점하지 못하기 때문에 순차적으로 실행되는 특징을 가지게 된다. Fig. 4는 위의 Fig. 3의 예제에서 myRes를 일반 자원에서 내부 자원으로 변경하였을 때의 수행을 보여준다. Fig. 3과 달리 T1이 T2에 의해 선점 당하지 않으며 T1 종료 후에 T2가 실행되는 것을 볼 수 있다.

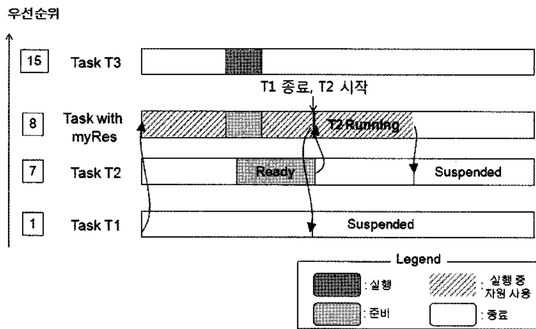



Fig. 4 내부 자원 사용에 따른 우선순위의 변화

## 2.3 응용 프로그램 구성 요소의 정적 설정과 OIL의 사용

OSEK OS는 응용 프로그램의 구성 요소를 동적으로 생성하는 것을 허용하지 않는다. 여기에서 구성 요소란 태스크, 알람, 이벤트, 일반/내부 자원을 말한다. 구성 요소의 동적 생성은 동적 메모리 할당을 필요로 하기 때문에 OS의 복잡도를 상승시키는 문제를 야기한다. 또한 동적 메모리 할당에 실패하는 경우에는 실시간성을 보장할 수 없는 문제도 초래한다. 이와 같은 문제들을 피하기 위해 OSEK OS는 응용 프로그래머가 미리 구성 요소와 그 속성을

```
TASK T1 {
  TYPE = Basic;
  PRIORITY = 8;
  //...
};
```

Fig. 5 태스크의 속성을 OIL로 기술한 예

정적으로 기술할 것을 요구한다.

OSEK OS는 이를 지원하기 위해 응용 프로그램의 구성 요소를 기술하는 언어인 OIL(OSEK Implementation Language)을 제공한다. 응용 프로그래머는 모든 구성요소를 하나의 OIL 파일에 기술한다. Fig. 5는 OIL 파일 중 태스크 T1을 선언하고 그 속성을 기술하는 부분을 보여준다.

OIL 파일은 OSEK OS 개발자가 제공하는 OIL 해석기/코드 생성기에 의하여 OS와 동일한 언어의 파일로 변환된다. 변환된 파일은 OS가 사용하는 자료구조를 담고 있기 때문에 각 OSEK OS의 구현에 의존적이다. 이에 더하여 변환된 파일은 인터럽트의 우선순위를 설정하고 인터럽트 벡터테이블을 기술하는 부분을 포함하기 때문에 하드웨어에도 의존적이다. 이렇게 응용 프로그래머가 OIL을 이용하여 응용 프로그램의 구성요소를 기술하면 응용 프로그램을 개발할 때 OS의 구현과 하드웨어에 의존적인 부분을 직접 작성하지 않아도 되고, OIL 해석기/코드 생성기가 이 부분을 담당하게 되므로 응용 프로그램의 이식성이 크게 높아진다.

변환된 파일은 OSEK OS 파일과 응용 프로그래머가 작성한 응용 프로그램과 함께 컴파일 되고, 이 결과로 타겟 하드웨어에서 실행될 이미지(Executable Image)가 생성된다. Fig. 6은 우리의 OSEK OS 구현을 반영하여 OIL을 이용한 응용 프로그램의 개발과정을 도식화 한 것이다. 우리는 OSEK OS를 C 언어로 작성하였으며, 이에 따라 OIL 해석기/코드 생성기 역시 OIL 파일을 해석하여 C 파일을 생성한다. 생성되는 intvect.c 파일은 인터럽트 벡터 테이블을 담고 있고, tcb.h, tcb.c 파일은 구성 요소들의 속성을 기술한다.

## 2.4 응용 프로그램 특성에 따른 최적의 OS 프로파일 선택

OSEK OS는 응용 프로그램을 구성하는 태스크의

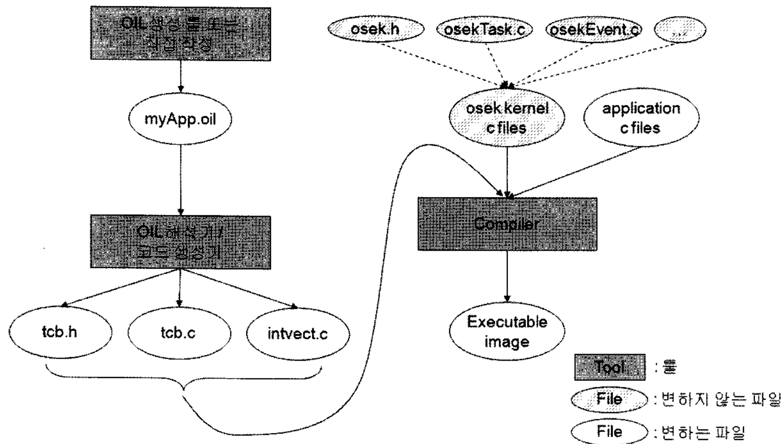



Fig. 6 OIL을 이용한 OSEK OS 응용 프로그램 개발 과정

특성에 따라 사용할 수 있는 네 가지의 OS 프로파일을 제공한다. 이는 CC(Conformance Class)라고 하며, 각 CC가 응용 프로그램의 특성에 맞는 서비스만 제공하도록 함으로써 불필요한 자원의 낭비를 막을 수 있다. OSEK OS는 BCC1, BCC2, ECC1, ECC2의 네 가지 CC를 정의하고 있다.

CC1(BCC1, ECC1)과 CC2(BCC2, ECC2)를 구분하는 기준은 여러 개의 태스크가 동일한 우선순위를 가질 수 있는지의 여부와 Basic 태스크의 다중 활성화를 지원 하는지의 여부이다. 이 두 조건은 결국 동일한 우선순위를 가진 둘 이상의 태스크가 동시에 준비 상태로 존재하는 것을 지원 하는 지의 여부이다. CC1은 이를 지원 하지 않고, CC2는 이를 지원한다. BCC(BCC1, BCC2)와 ECC(ECC1, ECC2)를 구분하는 기준은 Extended 태스크의 지원 여부이다. BCC는 오직 Basic 태스크만을 다룰 수 있고, ECC는 Basic 태스크와 Extended 태스크를 모두 다룰 수 있다. Fig. 7은 CC간의 하위 호환성 관계를 보여준다. CC간의 하위 호환성이란 하위 CC의 기능을 상위 CC가 모두 지원한다는 의미로 화살표의 시작이 하위 CC, 화살표의 끝이 상위 CC를 나타낸다. 예를 들어 BCC2의 OS는 BCC1이 제공하는 기능을 모두 지원하므로, BCC1을 사용하는 응용 프로그램은 BCC2에서도 잘 동작하게 된다. CC의 특성을 이용하여 효율적으로 메모리를 이용하는 커널 메커니즘을 사용 할 수 있는데, 이는 3.2절에서 자세히 살펴본다.

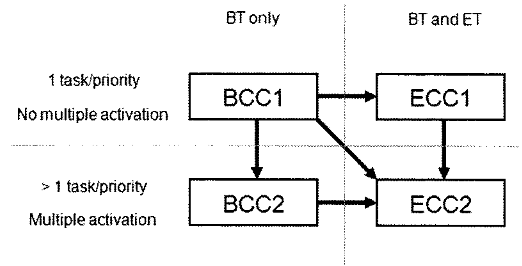



Fig. 7 Conformance Class의 분류

## 2.5 다양한 응용 프로그램을 실행할 수 있는 응용 모드의 제시

응용 모드(Application mode)는 OSEK OS가 ECU 내/외부의 다양한 상황에 따라 적절한 응용 프로그램을 선택하여 실행할 수 있도록 제시된 개념이다. 이는 윈도우즈 운영체제의 일반 모드와 안전 모드를 생각하면 쉽게 이해할 수 있다. 즉, 응용 프로그래머는 여러 개의 응용 모드를 만들 수 있고, 그 중 ECU 내/외부의 상황에 따라 OS 시작 직전에 하나의 응용 모드가 선택되어 실행되는 것이다. 응용 모드는 OS를 재시작 하지 않는 이상 변하지 않는다.

하나의 응용 모드는 OS 시작 시에 자동으로 수행을 시작할 태스크와 알람의 집합으로 정의된다. 이 태스크와 알람은 필요에 따라 다른 태스크들을 활성화 시킨다. 즉, 하나의 응용 모드에서 OS 시작 시 자동 수행되는 태스크와 알람은 응용 모드에서 수행되는 모든 태스크와 알람의 부분 집합인 것이다. 하나의 응용 모드에서 수행되는 태스크들은 다른

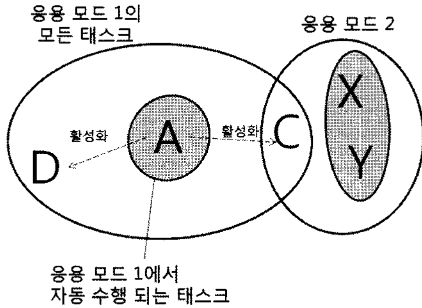



Fig. 8 두 개의 응용 모드와 태스크들

응용 모드에서도 수행될 수 있다. Fig. 8은 두 개의 응용 모드와 이에 속한 태스크를 보여준다. 태스크 A는 응용 모드 1에서 자동으로 수행을 시작하는 태스크이고 태스크 C, D는 태스크 A에 의하여 활성화되는 태스크이다. 태스크 C는 응용 모드 2에서도 수행됨을 알 수 있다.

## 3. 효율적 메모리 사용을 위한 커널 메커니즘

본 장에서는 2장에서의 분석을 바탕으로 OSEK OS와 응용 프로그램이 사용하는 RAM의 크기를 줄일 수 있는 메커니즘을 제시한다. 일반적으로 프로그램이 실행 될 때 RAM에는 힙 섹션, 스택 섹션, 데이터 섹션이 적재된다. 이때 OSEK OS는 동적 메모리 할당을 하지 않기 때문에 힙은 사용되지 않는다. 따라서 본 논문에서는 스택과 데이터 섹션의 크기를 줄이기 위한 세 가지 메커니즘을 제시한다. 이를 구체적으로 설명하기에 앞서, 우선 각 메커니즘을 고안하게 된 착안점을 기술한다. (S는 스택 섹션, D는 데이터 섹션을 의미한다).

- **S1**: 태스크가 실행 될 때에는 자신의 스택을 갖지만 실행되지 않을 때에는 스택을 유지하지 않도록 한다. (실행 종료 후에도 스택을 유지하는 것은 심각한 메모리 낭비를 초래한다.)
- **D1**: OS 커널은 각 CC의 특성에 맞추어 최소한의 자료구조만을 사용하도록 설계한다.
- **D2**: 실행 중 불변하는 데이터는 RAM이 아닌 ROM에 저장한다.

본 장의 각 절에서는 위의 아이디어를 구체화하여 태스크의 스택 공유 기법, CC에 최적화된 레디큐의 설계 기법, 불변하는 태스크 정보를 ROM에 저장하는 기법을 설명한다.

### 3.1 태스크의 스택 공유

OSEK OS와 같이 태스크를 동적으로 생성하는 것을 금지하는 RTOS의 경우, 각 태스크에 전용의 스택을 할당하는 것이 일반적이다. 따라서 각 태스크는 실행 여부에 상관없이 항상 일정한 스택 공간을 점유하게 된다. 그러나 실행이 종료된 태스크는 다시 활성화되기 전까지 스택을 사용하지 않으므로, 이 기간 동안 다른 태스크가 동일한 스택 영역을 사용할 수 있도록 한다면 메모리를 효과적으로 절약할 수 있다. 따라서 우리는 스택 사용기간이 겹치지 않는 임의의 두 태스크에 하나의 스택만을 할당하는 기법을 제시한다. Fig. 9에서 보듯이 스택 사용기간이란 태스크가 준비 상태에서 처음으로 실행상태로 변했을 때부터 시작하여 실행 상태에서 종료 상태로 변할 때까지의 기간이다. 이는 태스크가 대기 상태로 변하거나 실행 중 선점당하여 다시 준비상태로 존재하는 기간을 포함한다.

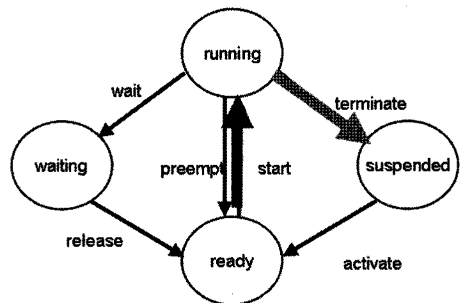



Fig. 9 태스크의 실행 상태

본 논문에서는 Table 2에 스택을 공유할 수 있는 4가지 경우를 제시한다. 스택 공유가 가능한 경우는 크게 응용 모드 내의 태스크들 간의 공유와 응용 모드 외의 태스크들 간의 공유로 나누어 볼 수 있다. 응용 모드 내의 태스크들이란 한 응용 모드에서 실행되는 모든 태스크들이다.

응용 모드 내의 태스크들 사이에 스택 사용 기간이 겹치지 않는 것을 보장하려면 태스크의 실행이

Table 2 스택을 공유 할 수 있는 4가지 경우

■ 응용 모드 내의 태스크들
① 비선점 Basic 태스크 들
② 동일한 내부자원을 공유하는 Basic 태스크 들
(단, ①, ②에 속하는 태스크는 Schedule() API를 호출하면 안 된다)
③ 동일한 우선순위를 가진 Basic 태스크 들
■ 응용 모드 외의 태스크들
④ 서로 다른 응용모드에 속한 태스크 들

순차적으로 이루어진다는 것을 밝히면 된다. 가장 쉽게 생각할 수 있는 경우는 비선점 태스크 간의 스택 공유이다. 비선점 태스크는 다른 어느 태스크에게도 선점당하지 않으므로 이들 태스크들은 항상 순차적으로 실행될 수밖에 없다. 이때 중요한 제약조건은 스택을 공유하는 비선점 태스크가 Schedule() API를 부르면 안 된다는 점이다. Schedule() API는 실행 중인 태스크의 선점 허용 여부에 관계없이 현재 우선순위가 가장 높은 태스크를 선택하여 실행시키는 API이다. 따라서 스택을 공유하는 비선점 태스크가 Schedule() API를 호출 할 경우 동일한 스택을 사용하고 우선순위 높은 비선점 태스크가 수행되어 이전 태스크가 사용했던 부분을 덮어 써버리는 문제가 생길 수 있다. 따라서 Schedule() API를 부르지 않는 비선점 Basic 태스크 간의 스택 공유가 가능하다.

동일한 내부 자원을 공유하는 태스크들도 2.2절에서 설명한 것처럼 항상 순차적으로 실행되므로 역시 스택을 공유할 수 있다. 하지만, 이 경우도 Schedule() API가 호출되어서는 안 된다는 제약 조건이 있다. 내부 자원을 사용하는 태스크가 Schedule() 함수를 부르면 자원을 반납하면서 자신의 본래 우선순위로 바뀌기 때문에, 내부 자원을 공유하고 자신보다 우선순위가 높은 태스크에게 선점당할 수가 있기 때문이다.

동일한 우선순위를 가진 Basic 태스크들은 서로 선점을 못하므로 이 태스크들은 항상 순차적으로 실행된다. 우선순위가 높은 태스크에게 선점을 당하는 경우 OSEK OS의 정책에 따르면 선점당한 태스크는 해당 우선순위의 레디큐의 가장 앞에 위치하게 된다. 따라서 우선순위가 높은 태스크의 실행이 끝나면 선점을 당했던 태스크가 우선순위가 동일한 다른 태스크 보다 먼저 실행되기 때문에 우선순위가 같은 Basic 태스크들은 항상 순차적으로 실행된다.

위의 세 가지 경우가 모두 Basic 태스크만을 대상으로 한 것은, Extended 태스크의 사용이 태스크의 순차적인 수행을 방해할 수 있기 때문이다. 예를 들어 스택을 공유하는 두 개의 태스크 중 하나가 실행 상태에서 대기 상태가 되어 CPU를 양보하면 다른 태스크가 실행을 시작하여 스택의 사용기간이 겹치게 되는 문제를 일으킬 수 있다.

응용 모드 외의 태스크들 간의 스택 공유는 직관적이다. 응용 모드는 실행 시 바뀌지 않으므로 서로 다른 응용 모드에 속한 태스크들은 태스크의 속성(태스크 타입, 우선순위 등)에 관계없이 절대로 동시에 실행될 수 없다. 따라서 이들 태스크 간에는 스택을 공유할 수 있다. Fig. 8의 예에서는 A와 X, D와 Y가 공유 하거나 A와 Y, D와 X가 스택을 공유할 수 있다. 두 경우 중 태스크가 사용하는 스택의 총 합이 작은 경우를 선택하면 된다.

Table 3은 5개의 태스크로 이루어진 응용 프로그램의 예시를 보여주며, 스택 공유 여부를 판단할 수 있는 정보를 기술하였다. 태스크 T1과 T4는 서로 다른 응용 모드에서 실행되므로 스택 공유가 가능하다. 또한 동일한 응용 모드에서 실행되는 T1과 T2는 비선점 Basic 태스크이고 Schedule() API를 이용하지 않으므로 역시 스택을 공유한다. 결과적으로 T1, T2, T4가 하나의 스택을 사용하며 그 크기는 셋 중 최댓값인 180 바이트로 결정된다.

Table 3 스택 공유 기법을 적용할 수 있는 응용 프로그램

| 태스크 이름 | Idle | T1 | T2 | T3 | T4 |
|---|---|---|---|---|---|
| 타입 (B: Basic, E:Extended) | B | **B** | **B** | B | E |
| 선점 여부 (P: Preemptable, N: Non-preemptable) | P | **N** | **N** | P | P |
| Schedule() 이용 여부 | X | X | X | X | X |
| 스택필요량 (byte) | 50 | 100 | 120 | 150 | **180** |
| 응용 모드 | A, B | **A** | A, B | A, B | **B** |

### 3.2 레디큐 설계의 최적화

본 절에서는 CC의 특징을 이용한 레디큐를 설계하고, 이를 리소스 서브시스템에 적용했을 때의 문제점과 해결점을 논한다.

#### 3.2.1 Conformance Class를 고려한 레디큐 설계

태스크가 0~63까지의 우선순위를 가진다고 하면 모든 CC에 공통적으로 사용할 수 있는 레디큐를 Fig. 10의 좌측과 같이 설계할 수 있다. 여러 태스크가 동일한 우선순위를 가지고 준비 상태로 존재하는 것을 허용한 구조이다. 레디큐는 크기가 64인 배열이다. 배열의 인덱스는 태스크의 우선순위와 일치하고, 배열의 각 요소는 head와 tail 포인터로 이루어진 구조체이다. Head와 tail 포인터는 우선순위 별로 존재하는 원형 큐의 처음과 끝을 가리키고 있다. 원형 큐의 크기는 각 우선순위 별로 동시에 준비 상태가 될 수 있는 태스크 개수의 최댓값과 같다. 즉, 한 우선순위에 존재하는 모든 태스크의 개수와 각 태스크들이 허용하는 다중 활성화의 횟수를 더한 값이다.

모든 CC에 사용할 수 있는 레디큐는 이렇게 복잡하지만, 각 우선순위에 한 개의 태스크만 준비 상태에 있을 수 있는 CC1의 경우에는 Fig. 10의 우측과 같이 훨씬 효율적인 레디큐 설계가 가능하다. 이전의 경우와 달리 원형 큐를 관리할 필요가 없으므로 레디큐는 크기가 64이고 각 요소는 태스크의 ID를 저장하는 배열로 충분하다. Fig. 10의 우측 레디큐에서는 우선순위가 각각 62, 3, 0 이고 태스크 ID가 5, 10, 0인 태스크들이 준비 상태인 것을 볼 수 있다.

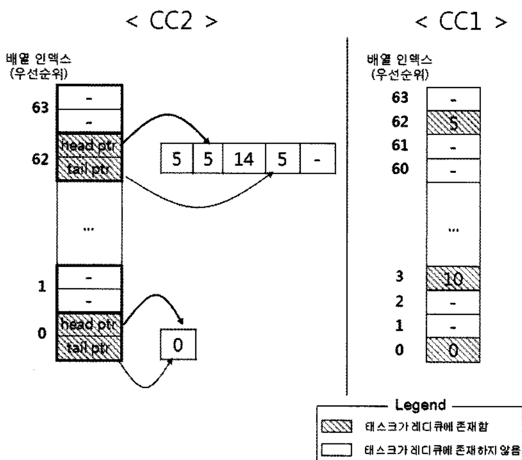



Fig. 10 CC1과 CC2의 레디큐 구현

이와 같이 CC1과 CC2의 특성에 따라 최적화된 레디큐를 설계할 수 있다. 만약 모든 태스크가 동일한 우선순위를 갖지 않는 경우에 CC2의 레디큐를 이용한다면 head, tail 포인터와 원형 큐 등의 불필요한 자료구조 사용으로 메모리를 낭비하게 된다.

이에 더하여, 응용 프로그램에 존재하는 태스크의 개수를 정적으로 알 수 있기 때문에, 이 정보를 이용하여 레디큐의 크기를 줄일 수 있다. 응용 프로그램 실행 시 필요한 레디큐의 크기는 응용 프로그램에 존재하는 서로 다른 우선순위의 개수와 일치한다. 왜냐하면 존재하지 않는 우선순위에 대해서는 레디큐를 유지할 필요가 없기 때문이다. 따라서 레디큐 중에서 쓰이지 않는 우선순위 부분은 제거할 수 있다.

이를 위해서는 OIL파일에서 파악한 태스크의 우선순위를 C 코드 생성 시에 0~n 까지 순차적으로 매핑 시키는 방법을 사용한다. Fig. 11은 응용 프로그램이 전체 64개의 우선순위 중에서 4개의 우선순위만 사용하는 경우를 보여준다. 이렇게 응용 프로그램이 소수의 우선순위만을 이용하는 경우에는 레디큐의 크기를 상당히 줄일 수 있다.

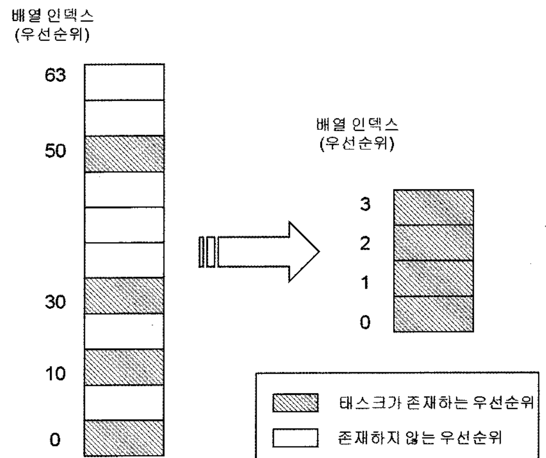



Fig. 11 CC1의 레디큐 크기 최소화

#### 3.2.2 리소스 서브시스템에의 적용

리소스 서브시스템에서 제공하는 우선순위 조정을 통한 동기화 기법을 사용할 경우, 위에서 언급한

효율적인 CC1의 레디큐 설계를 그대로 이용할 수 없다는 문제가 있다. 왜냐하면 동기화 기법을 적용하여 실행 시에 태스크의 우선순위를 변경하게 되면 CC1의 경우에도 두 개의 태스크가 동시에 같은 우선순위를 가지고 준비 상태에 존재하는 문제가 생기기 때문이다.

Fig. 12는 문제가 발생하는 예제를 보여준다. 이 예제는 Fig. 3의 예제에 3.2.1항에서 제시된 레디큐 최소화 기법을 적용한 결과이다. 즉, 태스크의 우선순위가 15, 7, 1에서 3, 2, 1로 변경되었고 이에 따라 자원 myRes가 가질 수 있는 우선순위는 오직 2 뿐이다. Fig. 12의 왼쪽은 시간에 따른 응용 프로그램의 실행을 나타내고, 오른쪽은 레디큐의 상태를 나타낸다. 응용 프로그램이 시작되면 가장 먼저 T1이 실행되고, T1이 자원 myRes를 획득하여 우선순위가 2로 변경된다. 이후 T3이 활성화되어 T1을 선점하면 T1은 우선순위 2를 유지한 채 준비 상태로 존재한다. T3 실행 중 T2를 활성화 시키면 T2는 T3보다 우선순위가 낮으므로 역시 준비 상태로 존재해야 한다. 이 때 우선순위 2인 T2를 레디큐에 삽입 할 경우 이미 존재하는 T1의 ID를 T2의 ID로 덮어쓰므로 태스크 T1이 준비 상태로 존재하고 있다는 정보를 잃어버리게 되는 문제가 발생한다.

이에 대한 간단한 해결책으로는 CC1의 OS도 동일한 우선순위를 가진 여러 개의 태스크를 관리할 수 있는 CC2의 레디큐를 사용하는 것이다. 하지만 이는 CC에 따라 최적화된 레디큐의 설계를 의미 없게 하므로 적합하지 않다.

우리는 CC1 레디큐의 장점을 유지하면서 문제를 해결하기 위하여 자원의 우선순위를 태스크와 중복되지 않는 유일한 값으로 설정하는 방법을 사용하였다. 즉, 자원의 우선순위를 자원을 사용하는 태스크의 최대 우선순위보다 항상 1만큼 큰 값으로 부여한다. 그리고 부여한 자원의 우선순위보다 같거나 큰 태스크의 우선순위를 모두 1만큼씩 증가시켜 태스크와 자원의 우선순위가 중복되지 않도록 한다.

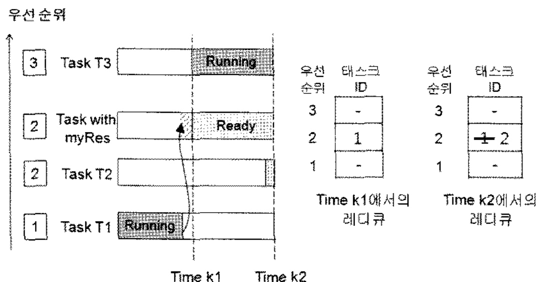



Fig. 12 우선순위 변경으로 인한 CC1 레디큐 문제

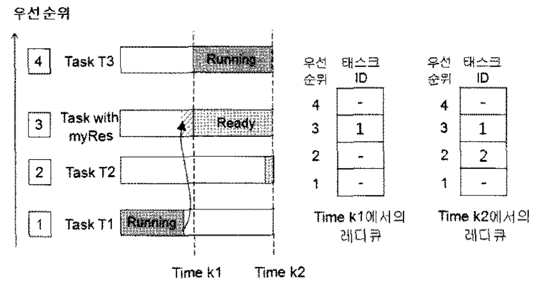



Fig. 13 자원의 우선순위 조정을 통한 문제 해결

Fig. 13은 이 방법을 적용하여 Fig.12의 문제가 해결된 예제이다. Fig. 12의 예제에서 우선순위 2를 가졌던 자원 myRes의 우선순위는 T2의 우선순위보다 1이 큰 3으로 결정되고, 이에 따라 우선순위가 3이었던 태스크 T3은 우선순위가 4로 변경된다. T1이 T3에게 선점될 때에는 우선순위 3을 유지한 채 레디큐에 존재하므로, T2가 활성화되어 레디큐에 삽입되어도 충돌이 일어나지 않는다.

이 방법의 단점은 자원에게 유일한 우선순위를 부여함으로 인해서 태스크가 가질 수 있는 우선순위의 개수가 줄어든다는 점이다. 이는 곧 사용 가능한 태스크의 개수가 줄어드는 것을 의미한다. 하지만 우리의 설계는 크기가 64인 레디큐를 사용하였고, 한 응용 프로그램에 존재할 수 있는 자원의 개수를 최대 8개로 제한하였기 때문에, 56개의 태스크를 이용하여 응용 프로그램을 구성하는데 무리가 없다고 판단된다.

### 3.3 태스크의 고정 정보를 이용한 TCB의 효율적 관리

일반적인 RTOS에서는 태스크의 정보를 TCB (Task Control Block)[7]에서 일괄적으로 관리한다. 이 방법은 직관적이지만, 응용 프로그램 실행 시에 동적으로 변하지 않는 정보까지 RAM에 로드하여 메모리를 효율적으로 이용하지 못하는 단점이 있다. Fig. 14는 이러한 TCB의 구조체를 보여주고 있는데,

```
typedef struct taskControlBlock {
  uint8  taskType; // BASIC or EXTENDED
  uint8  priority; // 1~63
  uint8  state;    // RUNNING or ..
  uint8  currentPriority;
                   //태스크의 현재 우선순위
  //...
}tcb_t;
```

Fig. 14 Task Control Block을 이용한 태스크 정보 관리

```
// 고정적 정보
typedef struct staticTCBType{
  uint8 type;
  uint8 priority;
  //...
}STCB_t;

// 유동적 정보
typedef struct dynamicTCBType{
  uint8 state;
  uint8 currentPriority;
  //...
}DTCB_t;
```

Fig. 15 실행시 변하는 정보와 변하지 않는 정보 관리

여기에서 taskType과 priority 등은 변하지 않는 정보이고, state나 currentPriority 등은 동적으로 변하는 정보이다.

RAM의 사용을 최소화하기 위해서 우리의 OSEK 구현에서는 동적으로 변하지 않는 부분은 ROM에, 변하는 부분은 RAM에 저장하도록 하였다. 이를 위해서 Fig. 14의 TCB처럼 모든 정보를 하나의 구조체에서 관리하지 않고, Fig. 15와 같이 고정 정보를 담고 있는 구조체와 유동적인 정보를 담고 있는 구조체를 각각 선언하여 사용한다. 그리고 고정적 정보를 담은 구조체는 const로 선언하여 ROM에 저장되도록 한다.

## 4. 구현 및 평가

본 논문에서 제안된 메커니즘은 2.3절에서 설명된 OSEK OS의 개발 과정 중 OIL 파일을 C 파일로 변환하는 과정에 적용된다. 즉, OIL 해석기/코드 생성기가 태스크의 속성을 분석하여 메모리를 절약하는 코드를 생성하는 과정이다. 본 장에서는 태스크의 속성을 OIL을 이용하여 기술하는 방법과 이에 따라 생성될 코드를 제시한다. 그리고 제안된 메커니즘을 응용 프로그램에 적용하고 메모리 사용량을 분석함으로써 제안된 메커니즘의 유용성을 확인한다.

Table 4 스택 공유 기법을 적용할 수 있는 응용 프로그램

| 태스크 이름 | Idle | T1 | T2 | T3 | T4 |
|---|---|---|---|---|---|
| 태스크 ID | 0 | 1 | 2 | 3 | 4 |
| 타입 (B: Basic, E: Extended) | B | B | B | B | E |
| 다중 활성화 횟수 | 0 | 0 | 0 | 0 | - |
| 우선순위 | 0 | 7 | 8 | 10 | 5 |
| 선점 여부 (P: Preemptable, N: Non-preemptable) | P | N | N | P | P |
| Schedule() 이용 | X | X | X | X | X |
| 스택필요량 (byte) | 50 | 100 | 120 | 150 | 180 |
| 수행되는 모든 응용 모드 | A, B | A | A, B | A, B | B |
| 자동 수행되는 응용 모드 | - | - | A | - | B |
| 사용하는 자원 | - | - | - | - | - |

OIL을 이용한 태스크 기술과 이에 따라 생성되는 C 코드, 절약되는 메모리양은 모두 응용 프로그램에 의존적이므로 본 장에서는 하나의 응용 프로그램의 예제를 이용하여 설명한다. 이 예제는 Table 4에 나타나 있으며 3.1 절의 Table 3에 기술된 예제를 확장한 것이다.

### 4.1 태스크 속성을 기술하는 OIL의 확장

OIL에서 태스크를 기술하는 항목은 기본적으로 우선순위, 선점 허용 여부, 다중 활성화 횟수, 응용 모드에 따른 자동 수행 여부, 사용하는 자원, 사용하는 이벤트, 사용하는 메시지이다. 제안된 메커니즘이 각 경우에서 필요로 하는 항목들을 살펴보고, OIL이 제공하는 기본 항목이 아니면 새로운 항목을 추가한다.

먼저 스택을 공유하는 경우부터 살펴보자. 서로 다른 응용 모드에 속한 태스크들의 경우에는 각 태스크가 어느 응용 모드에서 실행되는 지에 대한 정보가 필요하다. 이는 OIL에서 기술되는 자동 수행 여부와는 다른 것이므로 태스크가 실행되는 모든 응용 모드를 기록하는 UsedMode 항목을 추가하였다. 우선순위가 동일한 Basic 태스크의 경우에 필요한 정보인 우선순위와 태스크 타입은 기본적으로 기술된다. 비선점 태스크의 경우와 내부 자원을 공

유하는 태스크의 경우는 Schedule() API를 호출하지 않아야 한다는 제약이 있고, 이를 기술하는 Using-Schedule 이라는 항목을 추가하였다. 그리고 위의 모든 경우에 공통적으로 각 태스크가 사용할 스택의 크기를 알 필요가 있기 때문에 StackSize 항목을 추가하였다.

다음으로, CC에 따른 레디큐 구현을 위해서는 태스크의 타입과 다중 활성화 여부, 각 우선순위에 해당하는 태스크의 개수를 파악해야 한다. 이 항목들은 모두 OIL에서 기본적으로 기술된다. 마지막으로 태스크의 고정적/유동적 정보를 판단하는 것은 OIL의 기술과는 관련 없다.

Fig. 16은 확장한 OIL을 이용하여 태스크 T2를 기술한 예이다. 이를 통하여 T2 태스크는 비선점형이며, Schedule() 함수를 사용하지 않고, 응용 모드 A에서는 자동 실행되고 응용 모드 B에서는 다른 태스크나 알람에 의해서 활성화 된다는 등의 정보를 얻을 수 있다. 굵은 글씨는 새로 추가한 항목들이다.

```
TASK T2 {
  TYPE = Basic;
  SCHEDULE = NON;
  PRIORITY = 8;
  AUTOSTART = TRUE
  {
      APPMODE = A;
  };
  StackSize = 120;
  UsedMode = A;
  UsedMode = B;
  UsingSchedule = FALSE;
  //...
};
```

Fig. 16 OIL에 새로운 항목을 추가한 태스크의 기술

## 4.2 OIL파일의 해석을 통한 메모리 절약 코드 생성과 평가

본 절에서는 OIL 해석기/코드생성기가 4.1절에서 제시한 OIL 파일의 태스크 기술을 분석하여 생성할 코드를 제시하고, 이에 따라 절약되는 메모리의 양을 정량적으로 분석한다.

먼저 스택을 공유하는 경우이다. 3.1절의 분석에 따르면 T1, T2, T4가 스택을 공유하며 180 바이트의 스택을 사용하는 것을 알 수 있다. 따라서 응용 프로그램이 필요로 하는 스택의 양을 600 바이트에서 380 바이트로 절약하였다. 스택 공유를 위하여 Fig. 17

```
uint32 stack0[50];   //for idle task
uint32 stack1[180];
uint32 stack2[150];
uint32 taskStack[5] =
   {stack0,stack1,stack1,stack2, stack1};
```

Fig. 17 Table 2의 태스크 정보를 이용하여 생성된 코드

와 같은 코드가 생성된다. stack0, stack1, stack2 배열은 태스크 실행 시 사용되는 스택 공간들이다. taskStack 배열에는 각 태스크가 사용할 스택의 포인터가 저장되어있고, 각 태스크는 taskStack 배열에 태스크 ID를 인덱스로 사용하여 접근한다. Fig. 17에서 굵은 글씨로 표현했듯이, T1과 T2와 T4가 동일한 스택을 사용함을 알 수 있다.

Fig. 18은 문맥 전환 과정에서 태스크의 스택을 이용하는 코드이다. 스택을 공유하지 않는 태스크는 OS 초기화 과정에서 스택을 할당받아 문맥을 초기화 하는 반면, 스택을 공유하는 태스크는 태스크가 처음으로 실행되기 직전에 스택을 할당받아 문맥을 초기화해야 한다. Fig. 18에서 굵게 표시한 부분은 태스크의 스택 주소를 이용해서 문맥을 초기화 하는 과정이다.

```
// in Schedule() API

uint32_t hID =
          getHighestPriorityTaskID();
if (dynamicTCB[hID].stackAlloc == FALSE)
{
 dynamicTCB[hID].stackPointer =
      createContext(taskStack[hID], ...);
 dynamicTCB[hID].stackAlloc = TRUE;
}
// make this task run

restoreContext(dynamicTCB[hID].stackPointe
r[hID]);
```

Fig. 18 스택을 공유하는 태스크의 문맥 초기화

다음으로 CC에 따른 레디큐 설계의 구현을 살펴본다. CC에 따른 레디큐 타입의 선언과 레디큐의 선언은 OS의 코드에 Fig. 19와 같이 작성되어 있어 CC에 따라 필요한 부분이 컴파일 된다. CC1인 경우 레디큐는 단순한 배열이고 CC2의 경우는 head와 tail 포인터로 이루어진 구조체의 배열임을 알 수 있다. 그리고 레디큐의 크기는 3.2절에서 살펴본 것처럼

서로 다른 우선순위를 가지는 태스크의 개수와 자원의 개수를 합한 값이 된다.

```
#if defined (BCC1) || defined (ECC1)
typedef readyQ_t uint8;
#elif defined (BCC2) || defined (ECC2)
typedef struct readyQueueType {
    uint8 head;
    uint8 tail;
} readyQ_t;
#endif

readyQ_t
readyQueue[numOfDifferentPriority+
         numOfDifferentCeilingPriority];
```

Fig. 19 CC에 따른 레디큐 설계

Table 4의 예제에서는 모든 태스크가 다른 우선순위를 가지고 있고, 다중 활성화를 하지 않기 때문에 CC1의 레디큐를 사용한다. 이 경우에는 OIL 해석기/코드 생성기가 생성할 코드가 없다. 하지만 같은 경우 CC2의 레디큐를 사용한다면 Fig. 20과 같은 코드를 생성해야 한다. 즉, 각 우선순위마다 원형큐를 생성하기 위하여 priorityQueue# 배열을 선언하고, 원형큐의 주소를 저장하기 위하여 priorityQAddr 배열을 선언한다.

```
//CC2의 레디큐를 사용하기 위해 생성되는 코드

uint8 priorityQueue0[1];
uint8 priorityQueue1[1];
uint8 priorityQueue2[1];
uint8 priorityQueue3[1];
uint8 priorityQueue4[1];
uint8* priorityQAddr[numOfDifferentPriority
       +numOfDifferentCeilingPriority] =
    {priorityQueue0, priorityQueue1,...};
```

Fig. 20 Table 5의 태스크 정보를 이용하여 생성된 코드

3.2절에서 제시된 레디큐의 효율적 설계 기법을 적용하기 전/후의 메모리 사용량을 비교해보자. 적용 전에는 64개의 우선순위가 존재하고 각 우선순위마다 7바이트 (head, tail, 원형큐 각각 1 바이트, priorityQAddr 4바이트)가 필요하므로 448바이트가 필요하다. 하지만 효율적 설계 기법을 적용하면 크기가 5이고 각 요소가 1바이트인 배열로 충분하므로 5바이트가 필요하다. 즉 443 바이트를 절약하였다.

마지막으로 태스크의 정보를 ROM에서 관리할 때 생성되는 코드를 제시한다. 고정 정보를 담고 있는 구조체를 사용할 때에는 Fig. 21과 같이 const로 선언하여 read-only data (.rodata) 섹션으로 지정하고, 링커 스트립트에서 .rodata 섹션은 ROM에 위치하도록 설정한다. Fig. 22는 MPC5554 평가 보드에 사용한 링커 스크립트를 보여준다.

```
const  STCB_t  staticTCB[numOfAllTasks]  =
{
  {
        BASIC_TASK, //type
        0,          //priority
        //...
  },
  //...
}

DTCB_t dynamicTCB[numOfAllTasks] = {
  {
        SUSPENDED,    //state
        0,            //current priority
        //...
  },
  //...
}
```

Fig. 21 코드생성기에 의해 생성된 태스크 정보

```
MEMORY {
   rom : org = 0x00000008, len = 0x001FFFF8
  sram : org = 0x40000000, len = 0x1000
   //...
}
SECTIONS{
  .text : { *(.text) } > rom
  .rodata :{*(.rodata)} > rom
  //...
}
```

Fig. 22 MPC5554 평가 보드의 링커 스크립트

우리의 구현에 따르면 하나의 태스크를 기술하기 위해서는 총 21 바이트가 필요한데 그 중 13 바이트는 고정적인 정보로 ROM에 저장되고, 8바이트는 RAM에 저장된다. Table 4의 응용 프로그램은 5개의 태스크를 가지고 있으므로 총 105 바이트 중 65바이트를 절약하게 된다.

위의 세 가지 기법을 종합해 보면 제안한 메커니즘을 사용하기 전에는 총 1153 바이트가 필요했던 반면에 메커니즘 적용 후에는 425바이트가 필요했다. 총 63%의 메모리를 절약한 것이다. Fig. 23은 각 메커니즘의 적용 전후의 메모리 사용량을 보여준다.

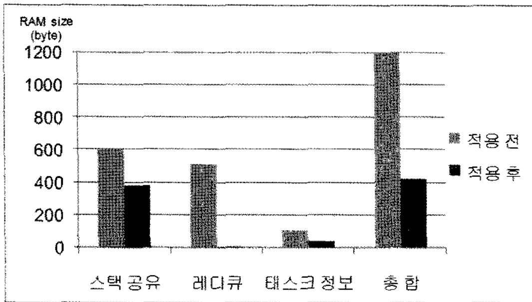



Fig. 23 정량적인 메모리 절약 분석

본 논문에서 제시된 기법들이 Fig. 23에서 분석된 특정 응용 프로그램 뿐만 아니라 다양한 종류의 응용 프로그램에 적용되었을 때에도 메모리의 사용량을 감소시킨다는 것을 보이기 위하여 추가적인 실험을 수행한다. 그리고 실험 결과를 토대로, 메모리 사용량을 더욱 효과적으로 줄이기 위해서 태스크의 우선순위를 어떻게 부여해야 하는지에 대한 가이드라인을 제시한다.

Table 5 실험의 조작 변인과 통제 변인

| ■조작 변인 |
|---|
| ① 태스크의 우선순위: 1~k까지 무작위로 부여<br>(k = 1, 0.2*n, 0.5*n, 0.8*n, n. n은 태스크의 수) |
| ② 비선점 태스크의 비율: 0, 30, 60% |
| ③ 태스크의 개수 : 10개~40개 |
| ■통제 변인 |
| ① Basic 태스크의 비율: 60% |
| ② 내부 자원을 사용하는 태스크의 비율: 0% |
| ③ 응용 모드의 개수: 1개 |
| ④ 태스크 스택의 크기: 100 바이트 |

이 실험에서는 다양한 응용 프로그램을 일정한 조건하에 생성시키고, 이 때 본 논문에서 제시된 기법을 사용하여 절약된 메모리의 양을 측정한다. 실험을 위한 응용 프로그램을 생성하기 위해서 Table 5와 같이 조작 변인과 통제 변인을 설정한다. 먼저, 몇 개의 태스크가 평균적으로 동일한 우선순위를 가지느냐를 나타내는 변수 k에 따라 5가지의 응용 프로그램들을 패턴을 정의한다. 변수 k가 1일 때는 모든 태스크가 같은 우선순위를 가지고, 0.2*n일 때는 평균 5개, 0.5*n일 때는 평균 2개의 태스크가 동일한 우선순위를 가지는 것을 의미한다. 그리고 응용 프로그램을 구성하는 전체 태스크 중 비선점 태스크의 비율을 0, 30, 60%로 변화시켰고, 응용 프로그램을 구성하는 태스크의 개수는 10개부터 40개까지 변화시켰다. 우선순위와 비선점 태스크의 비율 그리고 태스크의 개수를 제외한 나머지 속성들은 최대한 일반적인 응용 프로그램과 유사한 값으로 고정하였다.

이렇게 생성된 응용 프로그램들에 대해 본 논문이 제시하는 기법을 사용했을 때 응용 프로그램에 의해 사용되는 메모리의 양을 측정하였다. 그리고 이 결과를 본 논문의 기법을 사용하지 않았을 때의 응용 프로그램 메모리 사용량(대조군)과 비교하였다.

Fig. 24는 이 실험의 결과를 나타낸다. 비선점 태스크의 비율이 일정할 때에는 동일한 우선순위를 가지는 태스크가 많은 패턴일수록 여러 개의 태스크가 하나의 스택을 공유하므로 메모리 사용량이 줄어드는 것을 확인할 수 있다. 비선점 태스크의 비율을 증가시키면 비선점 태스크간의 스택 공유가 일어나 대부분의 패턴에서 메모리 사용량이 감소한다. 하지만 패턴 1과 2의 메모리 사용량은 거의 변하지 않는데, 이는 패턴 1과 2의 경우에 비선점 속성을 가진 태스크들보다 더 많은 수의 태스크들이 동일한 우선순위를 가지고 스택을 공유하기 때문이다.

메모리의 사용량을 정량적으로 살펴보면 태스크의 개수가 40개일 때 대조군은 5288 바이트가 필요한 반면, 절약 기법을 적용하였을 때는 최대 4433 바이트, 최소 2288 바이트가 필요하였다. 이는 각각 16%와 57%를 절약한 것으로 본 논문에서 제시된 기법들의 효용성을 증명할 수 있다.

실험 결과에서 주목할 만한 점은 패턴 5의 메모리 사용량이 비선점 태스크의 비율에 관계없이 패턴 4보다 항상 적다는 사실이다. 이는 패턴 5와 패턴 4가 동일한 우선순위를 가지는 태스크 사이의 스택 공유를 통하여 얻을 수 있는 메모리 절감 효과는 거의 동일한 반면에 패턴 5는 CC1의 레디큐를 사용하여 CC2의 레디큐를 사용하는 패턴 4보다 적은 메모리를 사용하기 때문이다. 이로부터 태스크의 속성을 부여하는 대략의 가이드라인을 도출할 수 있다. 즉,

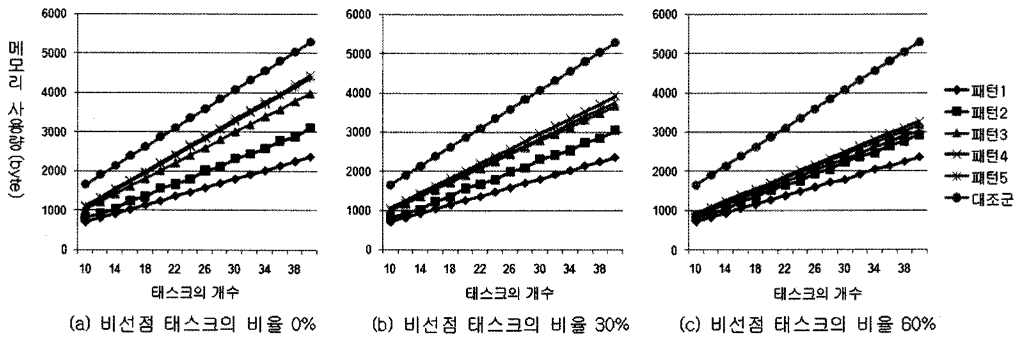



Fig. 24 비선점 태스크의 비율이 0, 30, 60%일 때 응용 프로그램의 메모리 사용량

응용 프로그램에 동일한 우선순위를 사용하는 태스크가 평균 2개 이하일 때에는 아예 서로 다른 우선순위를 부여하여 CC1 레디큐를 사용하는 것이 더 많은 메모리를 절약할 수 있다는 것이다.

위의 실험에서 통제 변인이었던 basic 태스크의 비율을 30%로 변화 시키거나 내부 자원을 사용하는 태스크의 비율을 0, 30, 60%로 변화시켜가며 추가적인 실험을 수행하였으며, 위의 실험과 비슷한 결과를 보였으므로 논문에 모두 제시하지 않는다.

## 5. 관련 연구

OSEK OS 표준이 발표 된 이후로 OSEK OS 구현의 성능을 향상시키려는 많은 노력이 있었다. 이러한 연구들은 주로 속도 향상과 메모리 사용 절약에 중점을 두고 있다.

8)에서는 태스크의 문맥 교환 시간을 단축시키기 위해 처음으로 실행되는 태스크의 문맥을 복구하지 않는 기법과 종료 상태로 변하는 태스크의 문맥을 저장하지 않는 기법을 사용하였다. 그리고 실험을 통하여 상용 OSEK OS인 OSEKTurbo 보다 향상된 성능을 보여주었다. 9)에서 제시한 Trampoline OS는 레디큐를 배열이 아닌 리스트로 구현한 특징을 가진다. 리스트를 사용하면 우선순위가 가장 높은 태스크를 찾는 시간은 빨라지지만 레디큐에 태스크를 삽입하는데 O(n) 시간이 걸리는 단점이 있다. 하지만 9)에서는 10)을 참조하여 태스크의 개수가 10개 이하일 때는 O(1) 시간이 필요한 비트맵 알고리즘을 사용하는 것 보다 리스트를 사용하는 것이 속도가 더 빠르고, 태스크의 개수가 30~40개이면 속도가 거의 비슷하다고 주장하였다.

11)에서 구현한 AlphaOS는 태스크 스택을 정적/동적으로 할당하는 것을 모두 허용한다. 동적으로 스택을 할당 할 때에는 외부 단편화를 막기 위하여 고정된 크기의 메모리 블록을 할당하고 반납하는 메커니즘을 사용하나 이는 내부 단편화로 인한 메모리 낭비를 막지 못하는 단점이 있다. 12,13)에서는 스택 영역을 절감하는 것에 초점을 맞춘 smartOSEK을 구현하였다. 먼저 문맥을 저장할 때 정적으로 결정된 일부 GPR(General Purpose Register)만을 저장하여 각 태스크의 스택에 저장할 문맥의 크기를 감소시켰다. 또한 인터럽트 서비스 루틴을 실행할 때 각 태스크의 스택을 사용하지 않고 별도의 스택을 사용하도록 하여 각 태스크의 스택 요구량을 절감하였다. 본 논문에서 제시한 것과 유사하게 비선점형 Basic 태스크 간의 스택 공유를 제시하기도 했으나 Schedule() API를 호출하지 않아야 하는 점을 간과하였다.

## 6. 결 론

본 논문에서는 OSEK OS와 응용 프로그램이 사용하는 메모리를 효율적으로 절약 할 수 있는 커널 메커니즘을 제시하였다. 이를 위해 먼저 OSEK OS의 태스크, 리소스 서브시스템과 구성 요소의 정적 설정, Conformance Class의 사용, 응용 모드의 사용 등 OSEK OS가 갖는 타 RTOS와의 차이점을 분석하였다. 분석 결과를 바탕으로 메모리를 효율적으로 사용할 수 있는 세 가지 메커니즘을 제시하였다. 이는 태스크의 속성을 이용한 태스크 간의 스택 공유

기법, CC에 따른 레디큐의 설계 기법, TCB의 고정적 정보 관리 기법이다. 제시한 기법을 사용하는데 필요한 태스크의 정보를 얻기 위하여 OIL에 새로운 항목을 추가하고, 이에 따라 생성될 코드를 제시하였다. 마지막으로 실험을 통하여 본 논문에서 제시한 기법이 다양한 응용 프로그램에 대해서 메모리를 절약하는 효과가 있다는 것을 검증하였다.

향후에는 스택 공유 기법을 확장하여 서로 다른 응용 모드에 속한 태스크가 일대일 관계가 아닌 일대다, 다대다 관계로 스택을 공유할 수 있는 메커니즘을 연구 할 계획이다.

현재 본 연구실에서는 2009년 1월을 목표로 자동차용 MCU인 MPC5554를 탑재한 평가보드에 OSEK OS를 구현하고 있고, OIL 해석기/코드 생성기도 작성 중이다. 차후 구현이 완료 되면 연구실 홈페이지[14]를 통하여 오픈 소스로 공개할 계획이다.

## 후 기

본 연구는 산업자원부에서 시행하는 자동차 부품기반기술개발사업(사업명: 네트워크 기반 분산형 실시간 제어 시스템 설계 기반 기술 개발)으로 지원받아 수행한 연구결과이다. 연구비를 지원해주신 관계자 여러분께 감사를 표한다.

## References

1) M. Broy, “Challenges in Automotive Software Engineering,” Proc. of the 28th International Conference on Software Engineering, pp.33-42, 2006.
2) J. Cooling, “Real-Time Operating Systems for the Embedded World,” Open Control Systems - The Importance of Industrial Standards, pp.4/0-4/10, 2004.
3) J. Youn, M. Sunwoo, J. Ma and W. Lee, “Model Based Design and Validation of Control Systems using Real-time Operating System,” Transactions of KSAE, Vol.16, No.2, pp.8-17, 2008.
4) J. Dirk, “OSEK/VDX History and Structure,” OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar, pp.2/1-2/4, 1998.
5) OSEK/VDX, http://portal.osek-vdx.org
6) Freescale, MPC5553/5554 Microcontroller Reference Manual, http://www.freescale.com, 2007.
7) P. Abraham and Greg, Operating System Principles 7th Edition, Wiley, New Jersey, pp.81-82, 2006.
8) Z. Wu, H. Li, Z. Gao, J. Sun and J. Li, “An Improved Method of Task Context Switching in OSEK Operating System,” Proc. of the 20th International Conference on Advanced Information Networking and Applications, pp.217-222, 2006.
9) J.-L. Bechennec, M. Briday, S. Faucou and Y. Trinquet, “Trampoline an Open Source Implementation of the OSEK/VDX RTOS Specification,” Conference on Emerging Technologies and Factory Automation, pp.62-69, 2006.
10) D. W. Jones, “An Empirical Comparison of Priority-queue and Event-set Implementations,” Commun. ACM, Vol.29, Issue 4, pp.300-311, 1986.
11) Y. Gu, Z. Wu and L. Yue, “AlphaOS, An Automotive RTOS Based on OSEK/VDX: Design and Test,” Proc. of Networking, Sensing and Control, pp.174-179, 2005.
12) W. Chen, Z. Wu and X. Wang, “Minimizing Memory Utilization of Task Sets in Smart-OSEK,” Proc. of the 19th International Conference on Advanced Information Networking and Applications, Vol.2, pp.552-558, 2005.
13) T. Chen, W. Chen, X. Wang and W. Hu, “Implementing and Evaluation of an OSEK/VDX-Compliant Configurable Real-time Kernel,” Proc. of Networking, Sensing and Control, pp.555-559, 2005.
14) SNU RTOSLab, http://redwood.snu.ac.kr