공 학 박 사 학 위 논 문

실시간성과 결함허용을 고려한 임베디드 태스크 스케줄러

2011년 2월

부 경 대 학 교 대 학 원

IT융합응용공학과

전 태 건

공 학 박 사 학 위 논 문

실시간성과 결함허용을 고려한 임베디드 태스크 스케줄러

지도교수 김 창 수

이 논문을 공학박사 학위논문으로 제출함.

2011년 2월

부 경 대 학 교 대 학 원

IT융합응용공학과

전 태 건

전태건의 공학박사 학위논문을 인준함.

2011 년 2 월 23 일

주 심 이학박사 박 만 곤 인

위 원 이학박사 이 경 현 인

위 원 공학박사 배 인 한 인

위 원 공학박사 정 민 수 인

위 원 공학박사 김 창 수 인

< 차 례 >

<그림 차례>

<표 차례>

An Embedded Task Scheduler considering Real Time Characteristics and Fault Tolerance

Tae Gun Jeon

Department of IT Convergence and Application Engineering, Graduate School, Pukyong National University

Abstract

Embedded systems need to ensure real-time of the task response time depending on the applied fields of it. Nevertheless, existing real-time operating systems in embedded systems perform the task scheduling based on the priority of task without considering aperiodic task. Therefore the properties of both periodic and aperiodic tasks should be considered in order to effectively schedule the real-time tasks. Task fault could be caused by various reasons in real-time system, and real-time system is required to be predictable performance despite occurrence of the fault.

In this paper, we propose a task scheduler that considers real-time and fault tolerance in embedded system with a single processor. The scheduler consists of an ECM(executability check manager), BTM(backup time manager), STM(surplus time manager), PTS(periodic task scheduler), FTM(fault tolerant manager) and APTS(aperiodic task manager). The ECM examines the feasibility of a set of periodic tasks before scheduling it and then can reducing the overhead of reconstructing a poorly designed task set. The BTM calculates and manages backup times which is used to recover a faulty task. The STM maintains the processor idle times for aperiodic tasks that can be obtained by excluding from the total processor times to the periodic tasks's execution times. The PTS schedules a set of periodic tasks and supports the execution of

aperiodic tasks. The FTM requests back up times to the BTM when a transient fault of task is occurred and recovers a faulty task by restarting it using back-up times. The APTS requests surplus times to the STM when aperiodic tasks generated and schedules aperiodic tasks by using surplus times.

We analyze and evaluate if our task scheduler can meets periodic tasks's deadlines and guarantees of aperiodic tasks's completion. And we propose an important level of periodic tasks that can controls response time of periodic and aperiodic tasks. If an important level is raised, the response time of aperiodic tasks increases and response time of periodic tasks decreases. If you want to reduce the response time of aperiodic tasks, it can be obtained by reducing the important level of periodic tasks. Therefore this important level can be used when we formulates task scheduling policy considering the response time of tasks.

In order to provide fault tolerance, the scheduler manages backup times and reexecutes a fault task to restore it's conditions. And we analyse and evaluate the proposed fault-tolerance techniques by showing if it can meets deadlines of normal periodic tasks and a faulty task.

1.서론

1.1 연구 배경

전기, 전자, 컴퓨터 등의 기술들이 눈부시게 발전하면서 이들 기술들을 이용한 다양한 제품들이 개발되었다. 예를 들면, TV, 냉장고, 세탁기, 전자레인지와 같은 전자 가전제품뿐만 아니라 일상생활에 필요한 핸드폰, PDA, 교통관리 시스템, 주차 관리 시스템, 엘리베이터 시스템, 현금 지급기(ATM), 항공관제 시스템, 군사용 제어 장치, 네비게이션 장치 등의 제품들이다. 이러한 기기들은 범용컴퓨터가 처리하는 다양한 작업과는 달리 특정한 요구 사항을 가지고 있으며, 하나 혹은 몇 개의 미리 정의된 작업(task)만을 수행한다. 이러한 특수한 기능만을 수행하는 시스템을 임베디드 시스템이라고 한다.

즉, 임베디드 시스템이란 “다른 시스템의 일부로 내장되어 시스템 내외부에 대한 센싱, 모니터링, 제어기능을 수행하는 마이크로프로세서 기반 디지털 시스템”을 의미하는 것으로 주로 특정 기기에 내장된 컴퓨팅 시스템을 일컫는다. 또한 일반적으로 임베디드 시스템은 “특정 목적을 위하여 동작하는 컴퓨팅 시스템”이라고 정의할 수 있다. 따라서 범용 컴퓨터를 제외하고 컴퓨팅 시스템이 내장된 모든 시스템이 임베디드 시스템이다.

초창기 임베디드 시스템들에서 수행해야할 작업은 많지 않았다. 그러므로 제시된 작업을 수행할 때 복잡한 스케줄링 기법을 제공하는

운영체제를 고려할 필요가 없었다.

최근 임베디드 시스템은 하드웨어 기술의 발달로 매우 집적화된 마이크로프로세서가 등장하면서 컴퓨팅 파워가 늘어나 여러 기능을 동시에 수행할 수 있게 되었으며 임베디드 시스템이 갖추어야 하는 사용자의 기능에 대한 요구 또한 다양하게 증가하였다. 그러므로 임베디드 시스템에서는 처리해야할 작업들이 많아지고 처리해야할 작업의 복잡성 또한 증가하게 되었다. 즉, 임베디드 시스템에서 운영체제에 대한 필요성이 대두되었다. 임베디드 운영체제는 항공기, 우주 왕복선, 산업 프로세스 제어와 스마트 자동차 등과 같은 작업 시간 제약을 만족해야하는 경우가 있다[1,2,3,4]. 이를 위해서는 외부 반응에 즉각적으로 응답해야하는 실시간성이 요구된다. 따라서 임베디드 운영체제는 실시간성을 보장하면서 태스크들을 관리할 필요가 있다. 즉, 태스크의 속성에 따른 스케줄링 정책이 필요하다.

현재 상용 및 공개용으로 제공되고 있는 임베디드 운영체제는 우선순위 기반의 선점형 태스크 스케줄링과 Round-Robin 기법들을 사용하고 있다. 그러나 이러한 운영체제에서 제공되는 스케줄링 정책들은 기존의 범용 스케줄링 정책들을 도입하여 사용하고 있기 때문에 임베디드 시스템에서 실시간 작업을 적용하기 위한 실시간 태스크 스케줄러가 필요하다.

실시간 태스크는 태스크가 주기적인 특성에 따라 실행되는지의 여부에 따라 크게 실행주기가 일정한 주기적(periodic) 태스크, 실행주기가 일정하지 않는 비주기적(aperiodic), 산발적(sporadic) 태스크로

구분될 수 있다. 또한 태스크들은 마감시간을 어겼을 때 결과의 심각성에 따라 임계(critical) 태스크와 비임계(non-critical) 태스크로 나눌 수 있다. 그리고 우선순위가 낮은 태스크가 수행 중에 우선순위가 높은 태스크가 도착했을 때 CPU를 양보하는지의 여부에 따라 선점(preemptive) 태스크와 비선점(nonpreemptive) 태스크로 나눌 수 있다. 태스크에 우선순위를 부여하는 방법으로 정적 우선순위(static priority) 기법과 동적 우선순위(dynamic priority) 기법이 있다.

임베디드 실시간 태스크 스케줄러는 효율적인 태스크 관리를 위해 실시간 태스크들을 주기적 태스크와 비주기적 태스크로 구분하여 스케줄링 할 필요가 있다. 일반 시스템에 비해 제한적인 시스템 자원을 가지는 기존의 임베디드 시스템에서의 태스크 스케줄러는 이러한 실시간 태스크들을 실시간 속성에 따라 구분하지 않고 모든 태스크들을 주기적 태스크로 간주하여 스케줄링 하고 있다. 즉, 비주기적 작업 속성을 가지는 태스크도 주기적 태스크와 동일한 속성으로 분류하여 스케줄링되고 있다. 이는 임베디드 실시간 시스템의 특성상 생성되는 태스크 수가 많지 않기 때문에 가능하였다. 또한 주기적 태스크에 대한 관리는 가능하지만 도착시간이 일정하지 않은 비주기적 태스크 관리를 위한 기법이 운영체제 수준에서 제공되어야한다. 또한 임베디드 프로세서 성능 향상과 더불어 임베디드 실시간 시스템의 컴퓨팅 파워가 점점 높아짐에 따라 임베디드 시스템이 처리해야할 작업 프로세스의 수가 증가하여 시스템 내의 태스크 수도 크게 증가하였기 때문에 빠른 CPU 응답시간과 효율적인 메모리 및 전력 관리를 위해서는 태

스크 속성에 따른 스케줄링 정책이 필요하다.

임베디드 운영체제에서는 여러 요인에 의해 태스크 결함이 발생할 수 있다. 태스크의 결함(fault)은 결함의 지속시간에 따라 영구적 결함(permanent fault), 일시적 결함(transient fault), 간헐적 결함(intermittent fault)으로 분류할 수 있다. 기존의 결함허용 연구들은 주로 여분(redundancy) 기법들을 사용하고 있으며 여분 기법들은 크게 공간 여분(space redundancy)과 시간 여분(time redundancy) 기법으로 나눌 수 있다[5].

임베디드 운영체제가 적용되는 시스템 중에는 수행 중에 발생하는 태스크의 결함이 전체 시스템에 치명적인 결과를 초래할 수 있다. 예를 들면, 의료장비나 자동차 엔진 제어, 항공기 운행 제어, 고속전철 신호 제어와 같이 안전함에 민감(safety-critical)한 임베디드 실시간 시스템에서 발생하는 시스템 결함(fault)은 인간의 생명에 직접적인 영향을 미칠 수 있다. 또한, 임베디드 시스템의 특성상 시스템이 설치된 후 시스템의 결함이 발생한 경우 사람이 즉각적으로 직접 제어하기 어려운 경우가 있다. 따라서 이러한 임베디드 실시간 시스템을 위한 운영체제는 반드시 작업 결함으로 인한 시스템 고장 혹은 시스템 재시작을 방지하는 기능을 포함하여야 한다. 즉, 임베디드 시스템에서 태스크 결함으로 인해 발생할 수 있는 문제점들을 해결하여 시스템의 정상적인 동작을 보장하기 위해서는 결함 허용 기법이 제공되어야 한다. 그러므로 기존의 결함 허용 기법들을 응용하여 이를 임베디드 실시간 시스템에 적용하기 위한 연구가 필요하다.

1.2 연구 목적

본 논문은 단일 프로세서를 가지는 임베디드 시스템에서 실시간성과 결함 허용을 고려한 태스크 스케줄러를 설계하고 구현한다. 구현된 스케줄러는 실시간성을 제공하기 위해서 태스크의 특성인 태스크의 주기성, 비주기성, 태스크의 실행시간, 태스크의 마감시간 등을 고려하여 태스크를 스케줄링한다. 또한 태스크 수행 중 발생할 수 있는 일시적인 결함에도 태스크를 재실행하는 복구 허용 기법을 제공한다. 그리고 결함이 발생하지 않은 태스크 및 결함이 발생한 태스크의 실행 마감 시간을 보장한다. 이를 위한 세부적인 연구 내용은 다음과 같다.

첫째, 우선순위 기반의 선점형 실시간 태스크 스케줄링 알고리즘을 분석하여 임베디드 실시간 운영체제의 태스크 스케줄러에 적용한다. 결함이 발생하지 않을 경우 정적 우선순위를 기반으로 태스크들을 스케줄링한다. 고정 우선순위의 주기적 태스크 스케줄링 기법의 경우 Rate Monotonic Scheduling(RMS)[6][7]이 최적의 알고리즘이므로, 본 논문에서의 주기적 태스크들의 우선순위는 RMS에 의해 결정된다.

둘째, 태스크 실행 중 발생할 수 있는 태스크 결함 허용을 위한 방법을 제시하고 이를 구현하고자 한다. 태스크에서 일시적인 결함이 발생할 경우, 미리 확보된 백업 타임을 이용하여 결함이 발생한 태스

크의 재실행을 통하여 복구 작업을 수행한다. 또한 결함이 발생한 태스크 및 결함이 발생하지 않은 태스크의 마감을 보장한다. 결함이 발생하였을 경우 결함이 발생한 태스크의 마감시간을 보장하기 위하여 동적 우선순위를 적용한다.

셋째, 주기적 태스크 집합의 마감시한을 넘기지 않으면서 비주기적 태스크를 스케줄링한다. 이를 위해 주기적 태스크들의 실행 가능 시간을 분석하고 비주기적 태스크를 위한 잉여시간을 관리한다. 본 논문에서는 잉여시간 관리자를 이용하여 잉여시간을 파악한 후, 비주기적 태스크에 할당하여 비주기적 태스크의 실행을 보장한다.

넷째, 주기적 태스크의 실행에 있어서 중요도라는 개념을 도입한다. 중요도란 주기적 태스크 중에서 빠른 응답시간이 필요한 경우 이 태스크에 대해서 선택적으로 빠른 응답 시간을 제공할 수 있도록 한다. 중요도가 적용된 주기적 태스크는 비주기적 태스크 보다 먼저 실행함으로써 빠른 응답시간을 나타낼 수 있다.

1.3 논문 구성

본 논문의 구성은 다음과 같다. 2장에서 기존의 임베디드 시스템에서 사용되고 있는 임베디드 실시간 운영체제에 대해서 기술한다. 3장에서는 실시간 태스크 관리와 결함 허용에 대한 관련 연구를 기술한다. 4장에서는 본 논문에서 제안하는 임베디드 실시간 태스크 스케줄러를 설계한다. 이를 위해 스케줄러의 구성 요소를 포함하는 구성

도를 제시하고 각 구성 요소의 기능에 대해 기술한다. 5장에서는 4장에서 설계한 스케줄러를 구현하고 그 결과에 대해 평가한다. 마지막으로 6장에서 결론을 기술한다.


PUKYONG NATIONAL UNIVERSITY
부경대학교

2. 임베디드 실시간 운영체제

임베디드 시스템은 유연한 구조를 갖춘 하드웨어 및 소프트웨어 결합 플랫폼으로 실시간 동작 수행을 통해 성능을 구현한다. 과거 임베디드 시스템은 작고 단순한 플랫폼에서 연산을 수행했기 때문에 각각의 시스템의 응답성에 최적화된 RTOS 운영체제가 주로 사용되어 왔다. 하지만 임베디드 시스템의 성능이 향상되고 운영체제에 필요한 커널수가 급속도로 증가함에 따라 특화된 기능에 맞추어 구현되는 RTOS(Real-Time Operation System)보다 다양한 인터페이스와 개발 환경을 지원하는 OS 시스템의 필요성을 인식하게 되었다.

임베디드 시스템은 군사 무기체계, 로봇, 인공위성 등과 같이 제한된 자원을 가지고 특정한 작업에서만 수행하던 전통적인 방식에서 휴대폰, 디지털 캠코더, PMP, MP3 플레이어와 같은 보다 복잡한 응용프로그램 구동을 필요로 하는 형태로 그 적용 범위가 확장되고 있다. 이러한 작업들을 효율적으로 수행하고 제어하기 위해서는 초기의 펌웨어(firmware) 수준의 시스템 제어프로그램으로는 한계가 있으므로 좀 더 복잡하고 신뢰성 있는 제어 프로그램인 임베디드 운영체제가 필요하게 되었다. 특히 우주항공, 자동차, 교통관리 시스템과 같은 분야에서 적용되는 임베디드 운영체제는 실시간성이 중요시 되므로 한정된 자원을 효율적으로 관리하고 시간적·논리적으로 정확성을 보장하기 위해서 운영체제 자체에서 실시간 시스템의 특성을 제공하는 임베디드 실시간 운영체제가 요구된다[35].

임베디드 시스템에 적용되는 실시간 운영체제는 다음과 같은 기능 및 특성이 요구된다. 임베디드 시스템이 가질 수 있는 자원이 한정되어 있으므로 처리기 및 메모리 관리에 있어서 효율적인 기법을 지원할 수 있어야 한다. 다양한 내장형 기기에 탑재되어야 하므로 각 기기들이 가지는 자원의 제약 환경에 따라 적합하게 구성할 수 있도록 하기 위해서는 운영체제를 기능별로 모듈화하여 적절하게 조립할 수 있어야 한다. 정의된 시간동안에 특정 동작이 수행되는 특성을 나타내는 결정성이 제공되어야 한다. 결정성은 완전 결정성과 완전 비결정성으로 구분된다. 완전 결정성은 항상 일정한 시간 내의 동작을 완료할 수 있음을 의미하며, 완전 비결정성은 동작 완료에 대한 일정 시간이 존재하지 않음을 의미한다. 임베디드 실시간 운영체제가 가지는 마지막 특징으로는 신뢰성이다. 이는 장기간동안 시스템 오류 없이 동작을 진행할 수 있음을 의미한다.

임베디드 실시간 운영체제는 시스템에서 태스크를 운영하는 방식에 따라 스레드(thread) 기반의 임베디드 운영체제와 프로세스(process) 기반의 임베디드 운영체제로 구분할 수 있다[36][37].

스레드 기반의 임베디드 운영체제란 시스템(커널) 프로세스와 응용(사용자) 프로세스가 구분되지 않고 동일한 주소 공간을 공유하여 실행되므로 문맥교환이 빠르고 커널 자원의 사용에 제한이 없으므로 실시간적인 태스크 수행이 요구되는 시스템에 적용이 가능하며 작은 규모의 시스템에서 운영되어 구현이 쉽고 빠르다는 장점이 있다. 그러나 동일한 주소 공간을 커널과 응용 프로세스가 구분없이 사용하기

때문에 응용 프로그램의 오류가 시스템 전체를 멈추게 할 수 있다는 단점이 있다. VxWorks[8], VRTX, pSOS, Nucleus Plus[9], SuperTask, uC/OS-II[10] 등이 이에 속한다.

프로세스 기반의 임베디드 운영체제란 시스템 프로세스와 응용 프로세스가 별도의 기준으로 구분되어 다른 주소공간에서 실행된다. 이러한 구조의 프로세스 기반 임베디드 운영체제에서는 응용 프로세스의 오류가 시스템 프로세스에 심각한 영향을 미치지 않으며 커널과 독립적인 모듈 단위의 응용 프로그램 개발이 가능하고 모듈의 추가, 변경이 용이하다는 장점이 있다. 그러나 스레드 기반의 운영체제에 비해 사이즈가 크기 때문에 소형 시스템 개발에는 부담이 되는 단점이 있다[34][37]. LunxOS, QNX[11], OS-9, RTLinux[12], Windows CE[13] 등이 있다.

그리고 임베디드 실시간 운영체제는 운영체제 커널 소스의 공개 여부에 따라 상용 또는 공개용으로 구분할 수 있다. 상용 임베디드 운영체제는 개발환경과 디버깅 툴(Tool) 등을 지원하므로 개발자들이 시스템을 개발하기에 편리하다. 또한 대부분이 POSIX API를 지원하기 때문에 호환성이 높은 제품을 개발할 수 있다. 그러나 상용으로 인해 라이선스 비용을 지불해야하고 운영체제의 소스코드를 수정 및 배포할 수 없다는 단점이 있다. VxWorks[8], Velos[14], pSOS+[15], QNX[11], VRTX, Nucleus Plus[9], Windows CE[13], BeOS[16] 등이 상용 운영체제에 속한다.

공개용 임베디드 운영체제는 오픈소스이므로 개발을 위한 운영체

제 소스코드 수정이 용이하고 운영체제에 대한 로열티를 지불할 필요가 없다는 장점이 있으나, 표준화되지 않은 소스코드로 인해 코드에 대한 안정성을 보증할 수 없다는 단점이 있다. uC/OS-II[10], eCOS[17], RTLinux[12], Tynux, Linuette, Velos[14] 등이 이에 속한다.

2.1 임베디드 실시간 운영체제의 종류

Velos[14]는 멀티태스킹(multi-tasking) 커널로, 다수의 쓰레드를 스케줄러를 통해 수행할 수 있으며 POSIX 표준 쓰레드를 지원한다. 또한 POSIX 쓰레드와 함께 확장된 형태의 실시간 쓰레드를 새로 정의하여, 사용자는 주기 쓰레드(periodic thread)를 구성할 수 있다. 각각의 쓰레드는 뮤텍스(mutex), 세마포어(semaphore) 등에 의해 동기화 되며, 주기 쓰레드를 위한 시간적인 동기화도 가능하다. 또한 쓰레드 간 통신을 위해 메시지 큐(message queue) 및 메시지 박스(message box) 등을 제공한다. 선점형 실시간 성을 보장하기 위해 256단계의 고정 우선순위 선점형 스케줄링(fixed-priority preemptive scheduling) 방식을 지원하며 동일 우선순위 내의 쓰레드 간에 FIFO(first-in first-out) 및 라운드 로빈(round-robin) 스케줄링을 지원한다. 사용자 수준에서 특정한 목적에 맞게 인터럽트 처리를 할 수 있도록 다양한 방법을 제공하기 위해서 하나의 인터럽트 소스에 대해 독립적으로 여러 개의 인터럽트 처리기를 등록 할 수 있는 구조를 제

공하며, 인터럽트가 쓰레드보다 높은 우선순위에서 수행되어 발생하는 우선순위 역전 현상을 방지하기 위해, 인터럽트 처리를 인터럽트 문맥상에서 뿐만 아니라 쓰레드 문맥에서도 처리할 수 있도록 지원한다. 응용 프로그램 또는 장치 드라이버 등의 모듈을 커널 수행 중에 동적으로 불러서 사용할 수 있는 확장 구조를 제공한다. 빠르고 안정적인 네트워크 스택을 사용하여 내장형 장비의 네트워크 기능을 완벽하게 지원한다.

VxWorks[8]은 WindRiver사에서 개발한 상용 임베디드 실시간 운영체제로서 확장 가능한 마이크로 커널을 제공한다. 즉, 선택적으로 커널 컴포넌트를 포함할 수 있으므로 개발 시스템의 규모에 맞게 커널 이미지의 크기 조정이 가능하다. 멀티태스킹, 우선순위 스케줄링, 태스크간 동기화와 통신, 로컬 파일 시스템 지원, 가상 메모리 관리 기능을 제공한다. VxWorks는 POSIX 인터페이스와 호환이 가능하며 다중 프로세스를 지원한다. 256단계의 태스크 우선순위 레벨을 가질 수 있으며 선점형 우선순위 기반으로 스케줄링한다. 또한 동일한 우선순위를 가지는 태스크들의 경우 선택적으로 Round-Robin 방식을 사용하여 태스크들을 스케줄링한다. 화성 탐사선 스피리트호와 오퍼튜니티호의 PowerPC 플랫폼에 탑재되었다.

Integrated Systems 사에서 개발한 pSOS+[15]은 VxWorks와 같이 클라이언트/서버 커널 구조를 가지며, 소프트웨어 버스를 이용하여 운영체제내의 각 모듈간 통신을 수행한다. [15]는 삼성전자가 개발한 휴대폰 핵심 칩인 Scom3000에 포팅되어 있으며 여러 통신 장비와 네

트워크 장비 등에서 사용되고 있는 실시간 운영체제이다. 스레드들은 하나의 어드레스 공간을 공유하기 때문에 문맥교환 시간은 짧으나 스레드간 메모리 보호가 이루어지지 않으므로 프로그래밍 기법이 시스템의 안정성에 영향을 줄 수 있다. pSOS+는 이러한 문제들을 메모리 보호 라이브러리를 통해 해결하고 있다. 태스크의 우선순위는 256단계까지 지원하며 레벨 240에서 255까지는 커널에 의해서 사용된다. 그리고 태스크의 우선순위 중복이 가능하다. 태스크 스케줄링은 우선순위를 가지는 FIFO와 Round-Robin 방식으로 수행되고, 우선순위가 중복된 태스크가 존재하면 그 태스크에 대해 time slicing에 의한 라운드로빈 스케줄링이 사용된다. pSOS+의 구성요소에는 pSOS+, pSOS+m, pHILE+, pNA+, pRPC+, pREPC+, pROBE+, pX11+, pSOS+Query 라이브러리, pRISM+가 있다. pSOS+는 단일 프로세서용 실시간 멀티태스킹 커널이고, pSOS+m은 다중 프로세서용 멀티태스킹 커널이다. pHILE+는 실시간 파일 시스템 관리자이며, pNA+는 TCP/IP 네트워크 관리자이다. pRPC+는 ONC Compliant Remote Procedure Calls 라이브러리이고 pREPC+는 ANSI C/C++ 표준 라이브러리이며, pROBE+는 Target Debugger이다. pX11+와 pRISM+는 그래픽 지원 런타임 코어와 통합개발환경이다. pSOS+Query 라이브러리는 pSOS+ 객체에 대한 상제 정보제공을 위해 사용된다.

QNX[11]은 마이크로 커널과 커널 계층 위의 시스템 서비스 담당 프로세스들로 이루어진 클라이언트/서버 구조를 가진다. 마이크로 커널은 스케줄링, 프로세스 디스패칭, 인터럽트 핸들링, IPC 등의 핵심

적인 서비스들만을 가지며 사용자 프로세스는 파일 시스템 관리, 프로세스 관리, 디바이스 관리, 네트워크 관리를 수행한다. 마이크로 커널과 시스템 서비스들은 서로 다른 CPU 모드에서 동작하기 때문에 시스템의 구조적 안정성을 보장할 수 있다. 프로세스는 자신의 메모리 영역을 가지고 있으며 32단계 우선순위 레벨을 가진다. 그리고 프로세스의 수는 2000개로 제한되고, 태스크 스케줄링은 우선순위를 가지는 FIFO와 Round-Robin, adaptive 스케줄링 기법을 사용하여 이루어진다.

VRTX는 VRTXoc(on-chip)와 VRTXsa(sacalable architecture)라는 두 종류의 커널을 가지는 임베디드 실시간 운영체제이다. VRTXoc는 극히 제한된 리소스를 가진 간단한 임베디드 시스템을 위한 커널이고, VRTXsa는 메모리 보호가 필요한 복잡한 임베디드 시스템을 위한 커널이다. 커널 상의 시스템 서비스가 모듈화 되어 있기 때문에 상황에 적절한 커널 크기로 줄일 수 있다. VRTX는 통신장비, 네트워크 장비, 모바일 기기와 산업용 모니터링 시스템 등의 분야에서 사용되고 있다. 선점형 멀티태스킹 커널을 가지고 있으며, 운영체제는 모듈 형태로 되어 있어 사용자들은 선택적으로 사용하여 운영체계를 구성할 수 있다. 이런 모듈들에는 멀티태스킹 커널 이외에 메모리가 매우 작은 임베디드 시스템을 위해 커널의 양을 최적화한 커널도 있으며 TCP/IP 프로토콜 스택 이외에 OSI 프로토콜 스택도 지원하고 있다.

Nucleus PLUS[9]는 Accelerated Technology사의 실시간 운영체

제로서 저작권이 없으며 그로 인해 널리 사용되고 있다. 인공위성 및 엘리베이터, 휴대전화, 기지국, 네트워크분야 등에서 많이 적용되고 있다. 또한 ANSI C 코드로 작성되어 있기 때문에 포팅이 용이하고 태스크간 통신, 태스크 동기화, 메모리 관리, 타이머 관리 기능과 다수의 디바이스 드라이버를 제공한다. 태스크는 자신의 메모리 포인터 리스트를 관리하기 때문에 프로세스간 메모리 영역침범이 발생하지 않는다. 그러나 모든 프로세스가 하나의 어드레스상에서 CPU의 슈퍼바이저 모드로 돌아가기 때문에 안정성에 문제가 있을 수 있지만 태스크 교환이나 시스템 콜의 오버헤드는 작아진다.

Windows CE .NET은 Microsoft 사에서 실시간 운영체제 및 임베디드 시스템을 목적으로 개발하여 제공하는 Windows CE 3.0의 후속 버전이다. 모바일 장치 및 메모리를 적게 차지하는 장치를 신속하게 개발할 수 있는 환경을 제공한다. Windows CE .NET은 풍부한 네트워킹, 경성실시간, 적은 메모리 점유율, 풍부한 멀티미디어, 웹브라우징 기능 등을 제공하고 있다. Windows CE는 32개의 독립된 멀티프로세스 기능을 제공하고 256개의 멀티 스레드 우선순위 수준을 지원한다. 또한 중첩 인터럽트 지원을 통해 시스템 자원 사용량이 많은 주요 응용프로그램에 대한 실시간처리를 지원한다. 또한 기존의 MS Windows 시스템 및 애플리케이션과 호환성이 우수하다는 장점이 있다. Win32 API를 지원하며 메모리, 프로세서, 스레드 관리 기능을 제공하며 커널은 프로세스와 스레드 함수를 이용하여 프로세스와 스레드를 생성 및 종료시키거나 동기화시킬 수 있고, 스케줄링 및 일시

정지시킬수도 있다.

uC/OS-II는 JEAN J. LABROSSE에 의해 공개용으로 만들어진 우선순위 기반의 선점형 실시간 커널이다[10][18][38]. 커널 코드의 대부분이 이식 가능한 ANSI C를 기반으로 하며, 일부 마이크로프로세서에 의존적인 부분은 어셈블리어로 작성되어 있다. 그리고 응용 프로그램에서 필요한 기능만을 이미지에 포함할 수 있도록 설계되어 있으므로 코드 공간이나 데이터 공간의 낭비를 줄일 수 있다. 최대 지원 가능한 태스크 수는 64개까지 가능하며 각 태스크는 중복되지 않는 유일한 우선순위 값을 가진다. 태스크의 우선순위는 64레벨을 가질 수 있으며, 이 중에서 최상위 4개와 최하위 4개는 시스템 용도로 예약되어 있고 255레벨의 Nest Interrupt를 지원하고 있다. uC/OS-II는 태스크간 통신과 동기화를 위해 메시지 큐와 메일박스, 세마포어를 지원하며 메모리 사용량을 줄일 수 있도록 태스크 스택의 크기를 서로 다르게 설정할 수 있다. 대표적인 공개용 커널로써 사용이 자유롭고 사이즈가 작아 많은 시스템에 적용되고 있다.

eCOS[17]는 매우 제한된 임베디드 시스템을 위한 운영체제로서 자체 C 라이브러리를 가지고 응용 프로그램을 개발할 수 있으며 EL/IX라는 리눅스 호환 API가 제공되어 리눅스용 응용 프로그램을 수정 없이 수행시킬 수 있다. 커널은 공개되어 있지만 개발 툴과 다양한 지원을 받기 위해서는 비용을 지불해야 한다.

Finite State Machine Labs 사에서 개발한 RTLinux[12]는 기존의 리눅스 상에서 경성 실시간 처리를 위해 시작된 프로젝트로서 공장

자동화, 로보틱스, 통신기기, 제어기기, 방위산업 등 실시간 임베디드 시스템이 필요한 산업 부분에 제공되는 RTOS이다. 또한 기존의 리눅스 커널을 재설계하지 않고 실시간 기능을 제공한다. 듀얼커널 기술(duel kernel technology)을 제공하여 Linux, BSD 등의 OS를 하나의 애플리케이션으로 생각하여 실시간이 필요한 부분은 RT CoreOS로 처리하며 동시에 실시간이 필요 없는 상황에서는 Linux나 BSD가 해결하도록 하여 하드웨어가 보장하는 내에서 빠르고 정확한 프로세싱 시간을 제공하고 있다. 상위 애플리케이션에 대한 부분도 리눅스나 BSD가 해결하게 함으로서 작은 사이즈를 유지하게 해 준다. RTLinux 커널은 실시간 프로세스의 생성 및 스케줄링, 인터럽트 처리, 리눅스와의 통신을 담당하며, 리눅스는 자신의 프로세스를 관리하고 인터럽트를 처리한다. RTLinux는 우선순위 기반 선점가능 스케줄링과 EDF(Earliest Deadline First) 스케줄링 기법을 사용한다.

iOS는 Mac OS X 10.5 기반으로 애플사에서 출시한 모바일 운영체제이다. 아이폰 SDK를 배포함으로써 아이폰용 애플리케이션을 쉽게 개발할 수 있다. iOS3은 통화와 MP3에 대해서만 멀티태스킹 기능을 지원하고 있으나 기본적으로는 단일 태스크 기반의 태스크 스케줄링 정책을 제공하였다. iOS4로 버전업되면서 백그라운드 작업을 이용한 멀티태스킹을 제공한다. 응용프로그램이 실행하고 있는 중에 다른 응용프로그램을 실행시킬 경우 기존의 응용프로그램은 background 작업(suspended state)으로 전환된다. 프로그램을 background 작업으로 전환함으로써 전원 사용을 최소화하여 전체적인 시스템 수행 능력

을 증가시키고 foreground 응용프로그램에 좀 더 많은 실행 시간을 제공할 수 있다. 그러나 백그라운드 작업으로 전환된 프로그램일지라도 중요한 태스크를 완료하기 위해서 일정한 실행시간을 요청할 수 있다.

안드로이드(Android)는 OHA(Open Handset Alliance)에서 개발된 운영체제와 미들웨어 및 핵심 모바일 어플리케이션을 포함한 모바일 기기를 위한 소프트웨어 스택이다. 리눅스 커널(Linux Kernel) 2.6을 기반으로 하는 개방형 운영체제이며 메모리 및 프로세스 관리, permission(권한) 기반의 보안 모델, 검증된 드라이버 모델, 공유 라이브러리 지원, 오픈 소스 기반의 장점을 가진다. 또한 멀티태스킹을 지원하므로 다양한 응용프로그램을 실행시킬 수 있다. 그러나 애플리케이션에 대한 보안성 및 정당성에 대한 검증 절차가 없으며 오픈소스라는 특징으로 인해 악성코드나 바이러스의 위험과 같은 보안 문제에 취약하고 한국형 앱스토어의 콘텐츠가 부족하며 제조사마다 다른 성능을 제공한다는 단점이 있다.

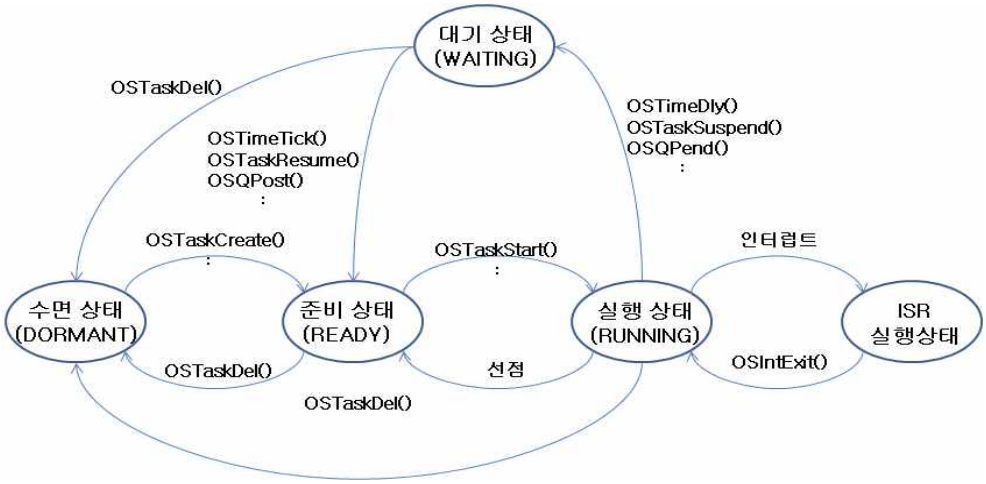
2.2 uCOS-II 태스크 스케줄러

본 논문에서 제안한 임베디드 실시간 스케줄러를 구현하기 위해서 uC/OS-II 실시간 운영체제를 이용하였으며 본 절에서는 uC/OS-II 실시간 운영체제의 커널에 대해 기술한다.

uC/OS-II는 우선순위 기반의 임베디드 실시간 운영체제로서

uC/OS-II에서 실행되는 태스크는 무한루프 함수이며 실행이 완료된 후에 OSTaskDel() 함수를 호출해서스스로를 삭제할 수 있다. 전체 64개의 태스크 우선순위를 지원하고 있으며 예약을 위한 우선순위를 제외한 56개의 우선순위를 태스크에 할당할 수 있다. 또한 스케줄링 중에 OSTaskChangePrio() 함수를 호출하여 우선순위를 제어할 수 있다.

<그림 2.1>은 uC/OS-II에서의 태스크 상태 전이도로서, 태스크는 DORMANT, READY, RUNNING, WAITING, ISR(Interrupt Service Routine)의 5가지 상태전이를 가진다[18]. DORMANT는 수면상태로서 프로그램 영역에 존재하지만 아직 태스크로서 생성되지 않은 상태이다. READY는 준비상태로서 OSTaskCreate() 또는 OSTaskCreateExt() 함수에 의해 생성되거나 대기 상태로부터 실행 준비가 된 태스크 상태이다. RUNNING은 처리기를 할당받아 태스크를 실행하는 상태이며 READY 상태의 태스크들 중 가장 높은 우선순위를 가지는 태스크가 선정된다. WAITING은 대기 상태로서 OSTimeDly() 함수 호출이나 이벤트 대기 함수 호출에 의해 이 상태로 전이된다. ISR은 인터럽트가 발생하면 전이되는 상태로서, 실행 중인 태스크가 ISR 상태가 되면 ISR이 처리기 제어권을 받아 실행하게 된다.



<그림 2.1> uC/OS-II의 태스크 상태 전이도

OSTaskCreate() 함수에 의해 태스크가 생성될 때, uC/OS-II는 <그림 2.2>와 같이 태스크 상태 관리를 위해 OS_TCB라는 태스크 제어 블록(Task Control Block)을 할당받는다. 할당된 태스크들은 메모리에 위치하며 커널에 의해 유지 관리된다. 라인 2~3은 태스크가 사용할 스택과 스택 크기 저장을 위한 변수이며, 라인 7은 태스크에서 스스로 실행을 지연시킬 경우 OSTimeDly() 함수에 의해 저장되는 타임아웃 값을 저장하기 위한 변수이다. 라인 10은 태스크의 우선순위를 저장하기 위한 변수이고, 라인 11~14는 태스크 준비 테이블(Task Ready Table)에서의 태스크 위치를 빠르게 접근하기 위한 마스크 변수들이다.

```
1 :  typedef struct os_tcb {
2 :   OS_STK        *OSTCBStkPtr;
3 :   INT32U          OSTCBStkSize;
4 :            :
5 :   OS_EVENT     *OSTCBEventPtr; // Pointer to event control block

6 :            :
7 :   INT16U OSTCBDly;// N ticks to delay task or, timeout waiting for event
8 :   INT8U  OSTCBStat; // Task status
9 :            :
10:   INT8U  OSTCBPrio; // Task priority (0 == highest, 63 == lowest)
11:   INT8U OSTCBX;//Bit position in group corresponding to task priority (0..7)
12:   INT8U OSTCBY;// Index into ready table corresponding to task priority
13:   INT8U OSTCBBitX; // Bit mask to access bit position in ready table
14:   INT8U OSTCBBitY; // Bit mask to access bit position in ready group
15: } OS_TCB;
```

<그림 2.2> uC/OS-II의 태스크 제어 블록(OS_TCB)

커널은 실행시킬 다음의 태스크를 결정하기 위해 OS_Sched( )를 이용하여 태스크 레벨의 스케쥴링을 수행하는데 이를 위해서는 생성된 태스크를 준비테이블에 할당한 후, 이 중에서 가장 높은 우선순위를 가지는 태스크를 선정할 수 있어야 한다.

<그림 2.3>은 생성된 태스크를 준비테이블에 할당하는 코드이다. 태스크 스케줄링을 위해 OSRdyGrp, OSRdyTbl[]라는 두개의 변수를 이용한다. 64개의 태스크를 8개의 그룹으로 묶고, 각 그룹마다 8개의 태스크를 관리하므로, 태스크의 우선순위 값의 하위 3비트는

OSRdyTbl[]안의 비트 위치를 나타내며, 다음 하위 3비트는 OSRdyTbl[]에 대한 인덱스가 된다. OSRdyGrp를 이용하여 가장 높은 우선순위를 가지는 태스크가 속한 그룹을 찾고, OSRdyTbl[]를 이용하여 준비 상태로 상태 전환할 가장 높은 우선순위의 태스크를 찾게 된다. 또한 준비테이블로 태스크의 상태를 전환할 경우 변수 OSMapTbl[]을 추가로 이용하며 이 변수는 이진 바이너리값을 저장하는 배열이다.

```
1 : OSRdyGrp  |= OSMapTbl[prio >> 3];
2 : OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

<그림 2.3> 준비테이블로 태스크 상태 전환

<그림 2.4>는 준비테이블에 있는 태스크 중에서 가장 높은 우선순위를 가지는 태스크를 선정하는 코드이다. 변수 OSRdyGrp, OSRdyTbl[] 그리고 OSUnMapTbl[]를 이용한다. 배열 OSUnMapTbl[]은 준비상태의 태스크 중에서 가장 높은 우선순위를 찾기 위해서 사용되는 배열이다.

```
1 : y = OSUnMapTbl[OSRdyGrp];
2 : x = OSUnMapTbl[OSRdyTbl[y]];
3 : prio = (y << 3) + x;
```

<그림 2.4> 가장 높은 우선순위 태스크 선정


PUKYONG NATIONAL UNIVERSITY
부경대학교

3. 태스크 스케줄링 알고리즘

3.1 주기적 태스크 스케줄링 알고리즘

고정우선순위를 부여하는 대표적인 태스크 스케줄링 방법은 Liu와 Layland에 의해 제안된 RMS(Rate Monotonic Scheduling)기법이다[7]. RMS에서는 주기 태스크를 태스크 모델로 사용하였으며, 모든 태스크의 마감시간은 태스크의 주기와 일치한다고 가정하였다. 태스크들 간에 선행 관계가 없다는 조건을 만족할 때 각 태스크 주기의 역수인 빈도율(rate)이 높을수록, 즉 주기가 짧은 태스크에게 높은 우선순위를 부여하는 방법을 제안하였다. 그러므로 각 태스크의 주기가 주어지면 태스크들간의 우선순위는 정적으로 결정될 수 있으며 이 우선순위는 시스템이 수행되는 동안 고정된다. 또한 RMS는 선점형 스케줄링 방법이므로 어떤 태스크가 실행 중일 때 더 높은 우선순위를 가지는 태스크가 도착하면 선점에 의해 우선순위가 높은 태스크가 실행 권한을 가진다.

RMS는 고정 우선순위 기반 선점형 스케줄링 방식 하에서는 최적의 스케줄링 방법임을 증명하였으며 태스크 집합의 실행 가능 여부를 검사하여 태스크의 실행 가능성을 판단하였다. 즉, N이 증가함에 따라 전체 처리기 이용률은 $U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\left(2^{\frac{1}{N}} - 1\right) \fallingdotseq 0.69$이며, 이는 U의 값이 0.69보다 작은 태스크 집합은 RM 스케줄링으로 항상 실행

가능함을 의미한다. 그러나 전체 처리기 이용률 U를 만족하지 않는 태스크 집합이라 할지라도 RMS에 의해 실행이 가능함을 Lehoczky[6] 등이 보여 주었다. 즉, Liu와 Layland[7]의 실행 가능성 분석 방법의 부정확성을 보완하여 실행 가능성 검사를 위한 필요충분 조건을 제시하였다.

Joseph 등[19]과 Audsley 등[20]은 식 $r_i^{n+1} = C_i + \sum_{j=1}^{i-1} \frac{r_i^n}{T_j} \cdot C_j,\ n = 0,1,2...$를 이용하여 태스크의 최악 반응시간(worst case response time)을 계산함으로써 태스크 스케줄 가능성을 분석하는 방법을 제안하였다. 위 식에서 태스크 t_i 의 최악 반응 시간 r_i는 자신의 실행시간 C_i와 함께 $[0,r_i]$구간 내에서 릴리스되는 자신보다 우선순위가 높은 태스크 인스턴스들의 실행시간 합계로 계산된다.

Tindell 등은 임의의 마감시한을 가지는 태스크들에 대해 최악 반응 시간 계산을 적용하여 스케줄 가능성 분석을 위한 필요충분조건을 제시하였다. 이 기법은 릴리스 지터(release jitter)가 존재하는 경우를 고려하여 이러한 지터를 줄임으로써 응답시간에 대한 최악 반응 시간을 감소시켜 스케줄 가능성을 최대한 증가시킬 수 있는 스케줄 가능성 분석식을 제안하였다[21].

Liu와 Layland는 EDF 기법에 대해서도 스케줄 가능성 분석을 위한 조건을 제시하였는데, RM 기법에서처럼 전체 프로세서 이용률 $U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$를 이용한다. 이 조건은 필요충분조건으로써 정확

한 스케줄 가능성 분석 방법을 제공한다. 또한 EDF 기법이 동적 우선순위 기반의 선점형 스케줄링 방식 중에서는 최적의 스케줄링 방법임을 의미한다. 즉, EDF 기법에 의해서 스케줄링될 수 없는 태스크 집합은 다른 어떤 우선순위 기반 스케줄링 방법을 사용하더라도 스케줄링될 수 없음을 의미한다.

EDF 기법과 유사한 동적 우선순위 기반 스케줄링 기법으로 MLF(Minimum Laxity First) 스케줄링 기법이 있다[24]. 이 기법에서는 어떤 시점 t에서 현재 실행중인 각각의 태스크에 대한 유휴시간을 식 $laxity = D_{i,k} - t - (C_i - C_{consumed})$을 이용하여 계산한다. $C_{consumed}$는 시점 t 까지 태스크 r_i의 k번째 실행 태스크 $r_{i,k}$ 가 소비한 시간이며, $D_{i,k}$는 $r_{i,k}$의 마감시간이다. 위 식에서 계산된 유휴시간이 적을수록 태스크는 더 높은 우선순위를 가진다. MLF 기법도 EDF 기법과 같이 최적의 우선순위 기반 선점형 스케줄링 방법이다. 그러나 태스크 실행 중에 $C_{consumed}$ 값을 계산하기 어렵고 그러므로 연속적인 t에 대해서 각 태스크의 유휴시간을 계산하기 어렵다는 단점이 있다.

3.2 비주기적 태스크 스케줄링 알고리즘

Lehoczky와 Ramos-Theul[22]은 고정 우선순위 시스템에서 비주기적 태스크 스케줄링을 위해서 슬랙 스틸링(slack stealing) 알고리즘을 제안하였다. 이 알고리즘은 비주기적 태스크 처리를 위한 passive

태스크를 생성하고 모든 주기적 태스크들의 마감시간을 만족함과 동시에 비주기적 태스크에 할당 가능한 처리기의 최대 유휴 시간을 계산한다. 그러므로 주기적 태스크의 마감시간을 보장함과 동시에 비주기 태스크의 응답시간 예측과 실행 시간을 보장한다. 태스크 집합을 스케줄링하기 전에 최대 유휴 시간을 계산하여 테이블에 저장해 둔 다음 태스크 집합을 실행할 때 이용하기 때문에 정적 유휴 시간 획득 기법이다. 그러나 이 기법은 비주기적 태스크의 실행가능 시간을 계산하기 위해 모든 주기적 태스크들의 스케줄 가능성을 검사하기 때문에 시간알고리즘의 계산복잡도로 인한 시간 오버헤드가 크다는 단점이 있다.

김형일 등[42]은 연성 비주기적 태스크 스케줄링 기법인 CTI 알고리즘을 제공하여 비주기적 태스크를 위한 slack stealing 알고리즘[22]을 개선하였다. CTI 알고리즘은 주어진 주기적 태스크에 대하여 오프라인에서 고정 우선순위를 바탕으로 마감시간지향 사전할당 테이블을 만들고, 고정 우선순위 스케줄링과 생성된 테이블을 참조하여 주기적 태스크의 마감시간을 보장하는 범위 내에서 비주기적 태스크가 가장 빠른 반응시간을 얻도록 온라인상에서 동적으로 스케줄링하는 방법이다.

동적 우선순위 시스템으로 제안된 Tia[23]의 알고리즘은 고정 우선순위의 슬랙 스틸링 개념을 동적 우선순위 시스템으로 확장하였다.

동적 우선순위 기반 스케줄링 방법 중에서 가장 널리 알려져 있는 것은 EDF(Earliest Deadline First) 스케줄링 기법으로 Liu와

Layland에 의해서 처음으로 제시되었다[7]. EDF 기법에서는 새로운 태스크가 도착할 때마다 현재 실행 중인 태스크들과 새로운 태스크의 마감시한을 비교하여 새로운 태스크의 우선순위를 결정한다. 이 때 마감 시한이 빠를수록 더 높은 우선순위를 부여받게 된다.

정경훈 등[39]과 김병훈 등[40]은 멀티 태스크 기반의 확장성과 주기 및 비주기 태스크 관리 기법을 효율적으로 제공할 수 있는 실시간 센서 노드 플랫폼을 제시하였다. 실시간성을 제공하기 위하여 주기적 태스크의 마감시한을 보장하고 비주기적 태스크의 응답시간을 최소화하는 방법을 제공하였으나 태스크 실행 중에 발생할 수 있는 결함을 복구할 수 있는 방법은 제공하지 못하고 있다.

김희헌 등[41]은 마감시간이 있는 주기적 태스크와 마감시간이 없는 비주기 태스크를 고려한 실시간 시스템에서 주기 태스크의 마감시간과 비주기 태스크의 빠른 응답시간을 보장한다. 비주기 태스크 처리에 효율적인 알고리즘인 Total Bandwidth Server(TBS) 보다 향상된 알고리즘인 Enhanced TBS(ETBS)를 제시하였다. EDF 스케줄링 알고리즘을 사용하는 단일처리기 시스템에서 주기 작업의 단위 수행시간마다 확보할 수 있는 잉여 여유시간을 이용해 온라인으로 비주기 태스크에 마감시간을 부여하는 알고리즘이다. 주기 및 비주기 태스크들이 처리기의 이용률을 모두 이용할 수 있게 하며 주어진 주기 태스크들의 마감시간을 보장하였다.

김진석 등[43]은 동적 우선순위에서 경성 비주기적 태스크를 스케줄링을 위해 Chetto and Chetto의 알고리즘이 가지는 비주기적 태

스크의 요청에 대한 제약 조건을 제거함으로써 비주기적 특성을 고려한 알고리즘을 제안하였다. EDL(Earliest Deadline as Late as possible)테이블을 만든 후, 온라인에서 경성 비주기적 태스크가 요청되면 EDL 테이블(유휴시간 테이블)을 이용하여 최대 유휴시간을 계산한다. 계산된 유휴시간에서 현재 수용할 비주기적 태스크의 실행시간을 제외하고 나머지를 다른 비주기적 태스크의 요청에 할당하는 가능하다면 위임하는 방법인 여분 슬랙 위임(MSC: Marginal Slack Commission) 알고리즘을 제안하였다.

3.3 결함 허용 알고리즘

실시간 시스템은 태스크의 실행 완료 시간의 결정성뿐만 아니라 지속적인 운영 가능성을 예측할 수 있어야 한다. 이를 위해 실시간 시스템의 결함 발생률은 최대한 낮아야 할 것이며, 비록 결함이 발생하더라도 이를 복구하여 시스템의 지속적인 실행이 보장되어야 한다.

결함은 결함이 하는 자원의 종류에 따라 하드웨어 결함(Hardware Fault)과 소프트웨어 결함(Software Fault)으로 분류할 수 있다. 하드웨어 결함은 시스템을 구성하는 물리적인 유닛(unit)에서 발생하는 결함으로써 끊어진 와이어, 지속적인 불분명한 결과를 나타내는 로직 게이트의 출력 등이 이에 속한다. 소프트웨어 결함은 처리기를 이용하여 실행하는 프로그램의 버그로 인해 프로그램 실행이 중지되는 결함이다.

태스크의 결함(fault)은 결함의 지속시간에 따라 영구적 결함(permanent fault), 일시적 결함(transient fault), 간헐적 결함(intermittent fault)으로 분류할 수 있다. 영구적인 결함은 전체적인 컴퓨팅 유닛(unit)의 실패에 의해 발생하며 정정 작업을 하지 않을 경우 발생한 결함이 무한정 지속된다. 일시적인 결함은 컴퓨팅 유닛(unit)이나 다른 관련된 콤포넌트의 일시적인 오동작에 의해 발생되며 부정확한 결과를 초래하고 그 영향 또한 지속적이지 않은 일시적이다. 간헐적인 결함은 일시적인 결함이 반복해서 나타나는 결함이다. 일시적인 결함의 발생률이 영구적인 결함의 발생률보다 30배 이상 높으며 또한 전체 발생하는 결함의 83%가 일시적인 결함이거나 간헐적인 결함이다.

결함에 대한 기존의 결함허용 연구들은 주로 여분(redundancy) 기법들을 사용하고 있다. 여분 기법들은 크게 공간 여분(space redundancy)과 시간 여분(time redundancy) 기법으로 나눌 수 있다[5]. 공간 여분 기법은 태스크를 서로 다른 처리기에서 동시에 수행하도록 하여 태스크 수행 중에 한 처리기에서 결함이 발생할 경우 다른 처리기에서 결함이 발생한 태스크를 계속 수행할 수 있도록 하거나 태스크 수행 시 필요한 데이터를 적어도 두 개 이상의 프로세서가 각각 사용하는 공간에 복제하여 한 프로세서에서 오류가 발생하더라도 다른 프로세스에서 보존된 데이터를 이용하여 태스크를 계속해서 수행될 수 있도록 하는 기법이다. 시간 여분 기법은 일정 간격으로 태스크의 상태를 저장하여, 태스크 수행 중에 결함이 발생할 경우 가장

최근의 작업 성공 시점부터 작업을 재시작하는 기법이다.

Oh와 Son[25]은 다중 처리기에서 결함 허용을 고려한 RMS(Rate Monotonic Scheduling) 알고리즘을 제안하였다. [25]에서 각 태스크는 자신과 동일한 여러 버전의 태스크를 가지며, 각 버전의 태스크는 사용되는 처리기의 수는 최소화하면서 서로 다른 처리기에 할당되는 것을 가정한다. 할당 알고리즘은 first-fit bin-packing 휴리스틱 방법이 사용된다. 즉, 동일한 두 개의 태스크는 동일한 처리기에 할당되지 않도록 한다.

Ghosh 등은 태스크를 스케줄링하는 동안에 발생하는 결함을 허용하는 기법을 제안하였다[26]. 이 기법은 같은 처리기 상에서 결함 태스크의 재실행을 통해 일시적인 결함을 허용하는 RMS 알고리즘이며 두 개의 태스크 요청들 사이에 결함이 발생한 태스크의 재실행에 이용될 수 있는 슬랙 타임(slack time)을 확보함으로써 다중 결함 허용이 가능하도록 하였다.

Pandya와 Malek[27]은 RM 기법으로 스케줄되는 주기적 태스크 집합의 실행 가능성을 분석하고 단일 결함을 허용하는 방법을 제공하였다. [27]은 결함 발생 이후의 태스크들 중 완료하지 않은 모든 태스크들을 재실행함으로써 복구 작업을 수행하였다. 또한, 처리기 이용률이 0.5보다 작거나 같으면 하나의 일시적 결함발생에 대해 어떠한 태스크도 마감시한을 넘기지 않는다는 것을 증명하였다.

Ghosh 등[28]은 primary/backup 방법을 사용하여 비선점, 독립적이며 비주기적 실시간 태스크들의 결함 허용 스케줄링을 연구하였다.

태스크가 시스템에 도착하면 primary는 가능한 한 빨리 스케줄이 되며 backup은 primary 스케줄링 시간보다 늦고 태스크의 마감시한 보다 빠른 시간에 스케줄된다. Primary가 성공적으로 수행을 종료하면, 그에 대응하는 스케줄링된 backup은 할당 해제된다.

Chen 등[29]은 단일 프로세서 상에서 태스크들 간의 의존성이 없다는 가정 하에 주기적 실시간 태스크들의 실행시간을 높일 수 있는 결함허용 실시간 스케줄링 기법(ICFTRM: Imprecise Computation Fault-Tolerant Rate-Monotonic)을 제안하였다. 각 태스크들의 특성을 파악하여 태스크가 수행하는 작업을 필수적으로 반드시 수행해야 하는 부분(Mandatory Part, 실행시간: M_i)과 선택적으로 수행할 수 있는 부분(Optional Part, 실행시간: O_i)으로 분류하였다. 또한 태스크의 실행 중 결함이 발생할 경우 M_i는 O_i에 할당된 프로세서 시간과 $M_i > O_i$인 경우를 위해 미리 확보한 여유시간을 이용하여 재실행한다.

Hong 등[30]은 분산 실시간 시스템에서 처리기의 이용률을 높이면서 주기적 태스크들을 위한 결함 허용 스케줄링 방법을 제안하였다. 태스크의 처리기 이용률에 따라 태스크를 분류하여 primary/backup 기반으로 결함이 발생한 primary 태스크의 재실행을 backup 태스크에서 재실행함으로써 결함을 복구하였다.


PUKYONG NATIONAL UNIVERSITY
부경대학교

4. 실시간 태스크 스케줄러 설계

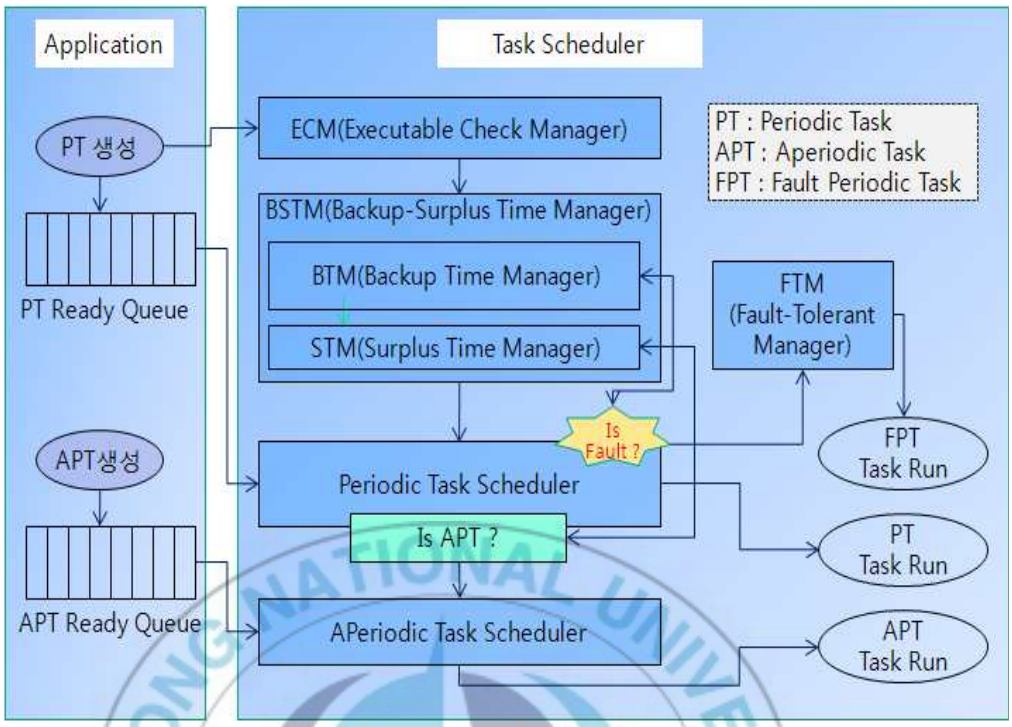
4.1 시스템 구성

본 절에서는 본 논문에서 제안하는 실시간성과 결함허용을 고려한 임베디드 스케줄러(RTFT-ETS: Real-Time Fault-Tolerant Embedded Task Scheduler)의 전체 구조와 구성 요소에 대해 기술한다.

임베디드 시스템에서 태스크들을 스케줄링할 경우, 실시간성을 제공하기 위해서는 시스템에서 실행하는 태스크들의 특성을 고려하여야 한다. 즉, 태스크의 우선순위, 태스크의 주기성 및 비주기성, 태스크의 마감시간과 같은 특징들이 고려되어야 실시간성을 제공할 수 있다. 또한 이러한 특징이 고려된 태스크들을 스케줄링할 수 있는 스케줄러가 필요하다. 즉, 운영체제 레벨에서 실시간 태스크를 관리하는 기법이 제공되어야 한다.

또한 실시간 시스템에서 고려해야 할 것은 시스템 운용의 지속성이다. 시스템은 실행하는 중간에 발생할 수 있는 예기치 않은 상황에서도 지속적인 태스크들의 실행을 보장하여야한다. 이를 위해서는 태스크의 실행 중 발생할 수 있는 태스크의 결함을 허용할 수 있는 기법을 제공할 필요가 있다.

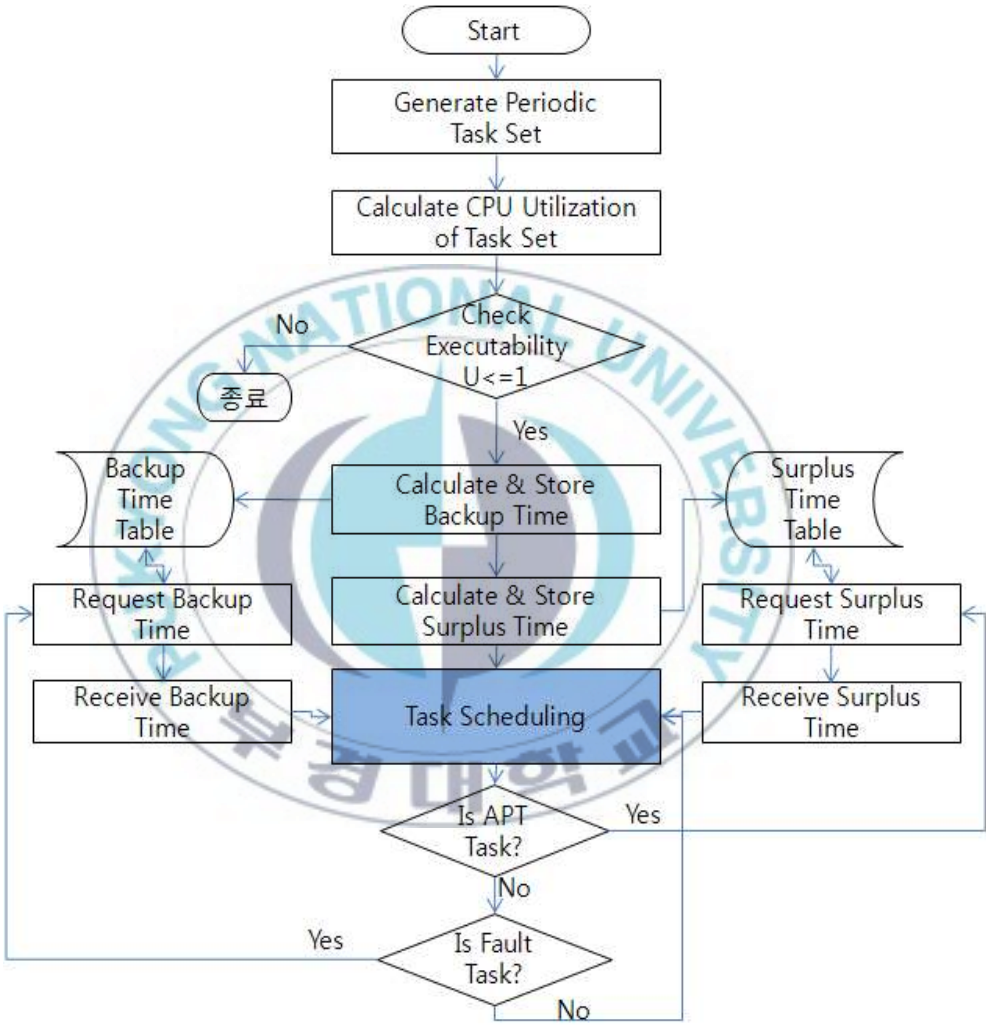
<그림 4.1>은 본 논문에서 제안하는 실시간 결함허용 태스크 스케줄러에 대한 전체 구성도를 나타낸다.



<그림 4.1> RTFT-ETS 구성도

본 논문에서 제안한 스케줄러는 실행 가능성 검사 관리자(ECM: Executability Check Manager), 타임 관리자(BSTM: Backup Surplus Time Manager), 결함 허용 관리자(FTM: Fault-Tolerant Manager), 비주기적 태스크 스케줄러(APTS: Aperiodic Task Scheduler)와 주기적 태스크 스케줄러(PTS: Periodic Task Scheduler)로 구성되어 있다. ECM은 결함이 발생할 경우 태스크의 결함 허용을 위한 백업 시간과 주기적 태스크들에서 고려되어야 할 시간관련 속성들인 실행 시간과 마감시간을 고려하여 실행 가능성 여부를 검사한다. BSTM은 결함 허용을 위한 백업 시간과 비주기적 태스크를 위한 잉여시간을

관리한다. BSTM의 구성 요소 중 하나인 BTM은 결함이 발생한 태스크를 재실행하기 위한 백업 시간을 관리하고 재실행 요청에 대해 백업 시간을 할당하며 STM은 비주기적 태스크에 할당할 수 있는 잉여시간을 관리한다.



<그림 4.2> RTFT-ETS 실행 흐름도

<그림 4.2>는 RTFT-ETS의 전체 실행 흐름도를 나타낸다. 먼저 주기적 태스크 집합을 생성한다. 그리고 생성된 집합을 이용하여 태스크 집합이 이용할 처리기 이용률을 계산한다. 계산된 처리기 이용률을 이용하여 실행 가능성을 검사한다. 만약 계산된 전체 처리기 이용률 $U \leq 1$이면 생성된 태스크 집합은 실행 가능한 집합이며 그렇지 않을 경우 실행 가능하지 않은 집합이므로 스케줄러는 종료한다. 실행 가능한 태스크 집합으로 판단될 경우 결함 태스크를 위한 백업시간을 계산하여 백업시간 테이블에 저장하고 비주기적 태스크의 실행을 보장하기 위한 잉여시간을 계산한 후 잉여시간 테이블에 저장한다. 주기적 태스크의 실행 중에 비주기적 태스크의 요구가 발생할 경우 비주기적 태스크의 실행을 위한 잉여시간을 검사하여 할당이 가능하면 비주기적 태스크에 잉여시간을 할당한다. 또한, 태스크 실행 중 결함이 발생할 경우 백업시간을 요청하고 이 요청에 의해 할당 받은 백업시간을 이용하여 결함이 발생한 태스크의 재실행을 수행한다.

4.2 ECM(Executability Check Manager)

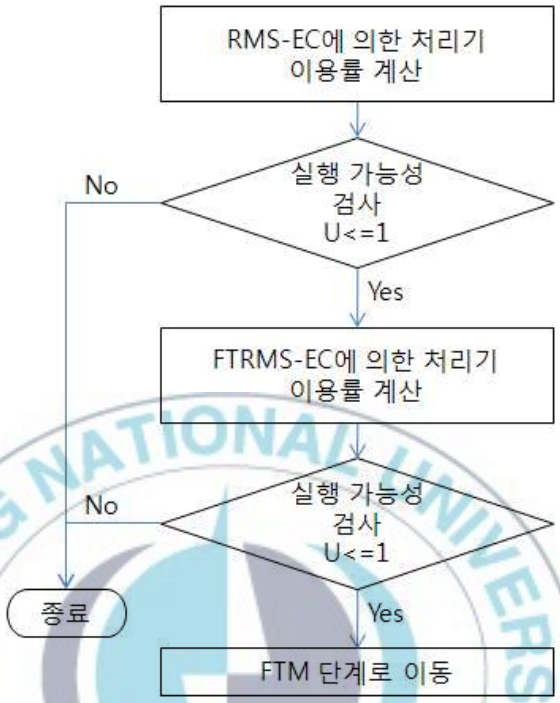
ECM은 주기적 태스크의 실행시간과 마감시간, 그리고 결함 허용을 위한 태스크 백업시간을 고려하여 주기적 태스크들의 실행 가능성을 검사한다. 이를 위해 주기적 태스크들이 실행 중에 처리기를 사용하는 처리기 이용률과 결함이 발생한 태스크를 위한 백업 시간 이용률을 기반으로 하여 전체 태스크의 처리기 이용률을 계산한다.

스케줄 가능성 검사를 위해 본 논문에서는 [6]에서 제안한 스케줄 가능성 검사 방법을 사용하며, 결함이 발생하였을 경우 결함을 복구하기 위하여 [26]에서 제시한 백업 시간을 이용한다. n개의 태스크로 구성된 태스크 집합에 대해서 다음과 같은 가정을 한다.

- 주기적 태스크를 τ_i라고 하고, τ_i의 실행시간과 주기를 각각 C_i와 T_i라고 하면 n개의 태스크를 가지는 태스크 집합 $\tau=\{\tau_1,\tau_2,\cdots,\tau_n\}$에 대해 태스크 τ_i와 τ_j의 실행 주기가 각각 T_i, T_j이고 $T_i<T_j$라면 태스크 τ_i의 우선순위가 태스크 τ_j의 우선순위 보다 높다. 또한 우선순위가 높을수록 낮은 태스크에 비해 먼저 수행된다. 즉, 우선순위 i를 가지는 태스크 τ_i는 우선순위 i보다 큰 태스크들의 수행이 끝난 후에 태스크 실행이 가능하다.
- 주기적 태스크 집합은 모든 태스크들의 초기 시작시간이 동일한 태스크 집합이다.
- 모든 태스크들은 하나의 처리기상에서 실행된다.
- 선점의 비용과 스케줄링 오버헤드는 고려하지 않는다.
- 태스크들은 모두 독립적이며 스스로 수행을 중단하지 않는다.
- 시간은 양의 정수로 나타낸다.

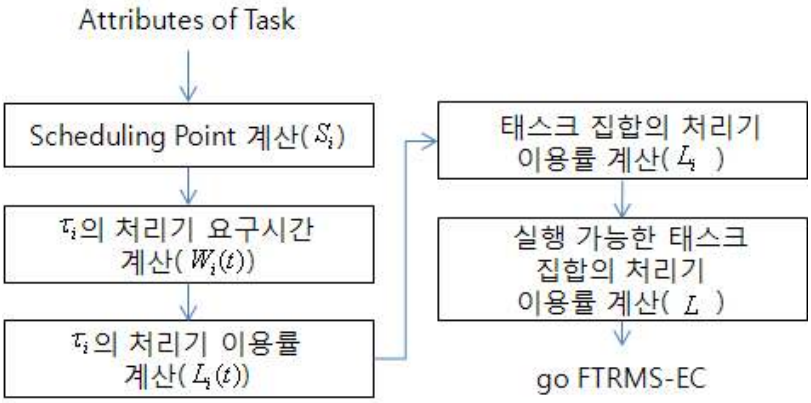
<그림 4.3>은 ECM에서 수행하는 태스크 집합의 실행가능성 검사 흐름도를 나타낸다. 효율적인 실행 가능성 검사를 위해 먼저 RMS-EC에 의한 실행 가능성을 검사한 후, 실행 가능이 판단될 경우

FTRMS-EC에 의한 실행 가능성을 검사를 수행한다.



<그림 4.3> ECM 수행 흐름도

RMS-EC(Rate Monotonic Scheduling Executability Checker)에서는 태스크의 결함을 고려하지 않았을 경우의 태스크 집합의 처리기 이용률을 계산한다. 그리고 FTRMS-EC(Fault-Tolerant Rate Monotonic Scheduling Executability Checker)에서는 태스크의 결함을 고려하여 추가적인 처리기 사용 시간을 할당하였을 경우의 처리기 이용률을 계산한다.



<그림 4.4> RMS-EC 흐름도

태스크의 결함을 고려하지 않았을 경우 태스크 집합의 실행 가능성은 RMS(Rate Monotonic Scheduling)를 따르며 다음 수식 (1)~(5)에 의해 결정되며 <그림 4.4>는 RMS-EC의 수행 흐름도를 나타낸다.

$$S_i = \{k \cdot T_j \mid j = 1,\ldots,i\,;\, k = 1,\ldots, \lfloor T_i/T_j \rfloor\} \quad (1)$$

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil t/T_j \rceil \quad (2)$$

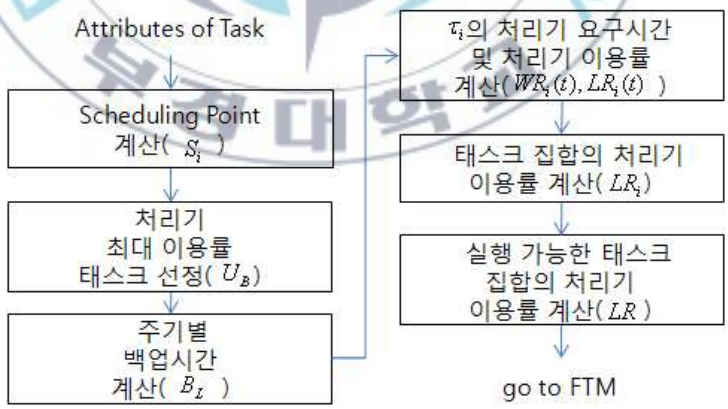
$$L_i(t) = W_i(t)/t \quad (3)$$

$$L_i = \min(L_i(t)),\ 0 < t \le T_i \quad (4)$$

$$L = \max(L_i),\ \ 1 \le i \le n \quad (5)$$

$t(=S_i)$는 τ_i에 대한 스케줄링 포인트(scheduling point)이다. 또한

τ_i의 마감 시간이며 τ_i의 마감 시간 전에 τ_i보다 높은 우선순위를 가진 태스크의 도착 시간이다. 즉, 주기 T_j의 배수 중의 하나가 되며 처리기 이용률을 계산하는 기준 시간이 된다. 수식 (2)는 시간 t에 대한 태스크 τ_i가 수행완료를 위해 요구되는 처리기의 시간요구량이다. 즉, 주기 t까지 추가된 태스크 집합이 실행하기 위해서 필요로 하는 처리기 이용 시간이 된다. 수식 (3)은 시간 t에 대한 태스크들의 처리기 이용률이다. 이 값은 우선순위가 높은 태스크부터 낮은 태스크들을 추가하면서 계산된 처리기 이용률이다. 그리고 L_i(수식 4)는 주기 t까지 추가된 태스크 집합이 실행하기 위해서 필요로 하는 최소한의 처리기 이용시간이다. 수식 (3)에서 계산된 처리기 이용률 중에서 최소값을 나타내며, 이 값은 실제 처리기 이용률이 된다. 수식 (5)는 수식 (4)에서 구한 처리기 이용률 중에서 최대값이다. 이 값은 전체 주기 태스크 집합의 처리기 이용률이다.



<그림 4.5> FTRMS-EC 수행 흐름도

결함 허용을 고려할 경우는 그렇지 않을 경우에 비해서 추가적인 프로세서 이용시간이 고려되어야 한다. 이를 위한 프로세서 이용률은 FTRMS-EC에서 계산하며 수행 흐름도는 <그림 4.5>에 나타난다.

즉, 결함이 발생한 태스크의 재실행을 위해서 추가적인 프로세서 사용 시간을 요구하게 된다. 결함이 발생할 경우 주기적 태스크 집합의 마감시간을 보장하면서 스케줄링시 제공된 백업 시간을 이용하여 결함이 발생한 태스크의 마감시간 이전에 태스크의 재실행 완료를 보장하는 복구 기법이 필요하며 다음과 같은 조건이 만족되어야 한다.

1. 모든 태스크 τ_i에 대해, kTi와 (k+1)Ti 사이에 적어도 Ci 만큼의 여유 시간(slack time)이 존재한다.
2. 태스크 τ_r의 실행 중 결함이 발생하였을 경우, 데드라인이 되기 전에 태스크 τ_r을 재실행 및 완료할 수 있어야 한다.
3. 결함이 발생한 태스크가 재실행 할 때, 복구 태스크에 의해 다른 태스크들의 실행 종료는 마감시간을 넘지 않아야 한다.

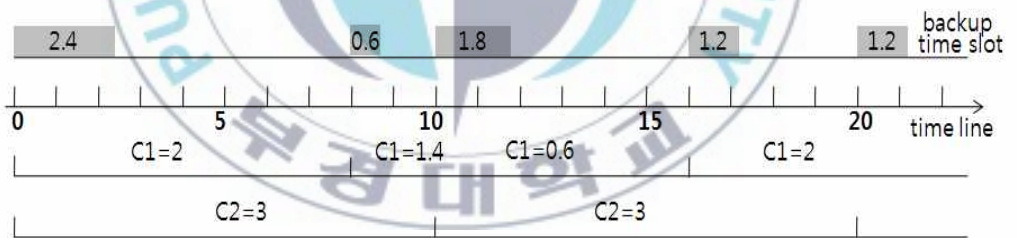
결함이 발생한 태스크의 재실행을 위한 백업 시간은 다음 수식 (6)~(7)에 의해 결정된다.

$$U_B = \max(U_i),\ U_i = C_i / T_i \qquad (6)$$

$$B_L = U_B^*(lT_j - kT_i) \qquad (7)$$

수식 (6)은 태스크들의 처리기 이용률 중에서 최대의 처리기 이용률을 의미하며, 백업 시간을 위해 계산된 수식 (7)은 태스크들의 주기와 주기 사이에 결함 허용을 위한 백업 시간을 할당한다. 즉, 태스크들의 최대 처리기 이용률과 τ_j의 l번째 주기 값인 lT_j값과 lT_j보다 작은 값 중에서 최대값인 τ_i의 k번째 주기 값인 kT_i의 차를 이용하여 결함 허용을 위한 백업 시간을 계산한다.

예를 들어, τ={(2, 8), (3, 10)}의 태스크 집합이 있을 경우, 태스크의 처리기 이용률은 각각 25%, 30% 이므로 U_B=30%가 되며 각 태스크들 주기 사이의 백업 시간은 시간 0에서 T_1사이는 2.4, T_1과 T_2 사이는 0.6, T_2와 $2T_1$ 사이는 1.8, $2T_1$에서 $2T_2$ 사이는 1.2의 백업 시간을 할당한다. 결함 허용을 위해 각 주기마다 할당된 백업 시간은 <그림 4.6>에서 회색 박스로 표시된다.



<그림 4.6> 백업 시간 할당

τ_i보다 더 높은 우선순위를 가진 태스크의 재실행으로 인해 τ_i를 실행하기 위해서 추가적인 프로세서 요구(HR_i)가 필요하고 τ_i보다 같

거나 낮은 우선순위를 가진 태스크의 재실행으로 인한 τ_i의 지연 시간(LR_i)이 필요하며 그 각각은 다음 수식 (8)~(9)와 같다.

$$HR_i = \max(C_1, ..., C_i) \quad (8)$$

$$LR_i = \min(T_i \cdot U_B, \max(C_{i+1}, ..., C_n)) \quad (9)$$

그러므로 결함 허용을 고려한 실시간 스케줄러의 실행 가능성은 수식 (10)~(13)에 의해 결정되며, $LR \leq 1$인 경우 태스크 집합은 실행이 가능함을 나타낸다.

$$WR_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil t / T_j \rceil + \max(HR_i, LR_i) \quad (10)$$

$$LR_i(t) = WR_i(t)/t \quad (11)$$

$$LR_i = \min_{\{0 < t \leq T_i\}} LR_i(t) \quad (12)$$

$$LR = \max_{\{1 \leq i \leq n\}} LR_i \quad (13)$$

<표 4.1>은 주기적 태스크 집합 τ={(2, 10), (3, 15), (5, 30)}가 주어진다고 가정할 경우, RMS에 의한 전체 처리기 이용률을 보여 준다. RMS에 따른 주기 태스크 집합 τ의 처리기 이용률은 L_1=0.20, L_2 =min(0.50, 0.47)=0.47, 그리고 L_3=min(1.00, 0.80, 0.75, 0.57)=0.57이 된다. 따라서 태스크 집합 τ의 전체 처리기 이용률은 L=max(0.20, 0.47, 0.57)=0.57이 되며 $L < 1$이므로 주기 태스크 집합 τ는 실행이 가능하

다.

<표 4.1> RMS-EC 처리기 이용률

i	τ_i	*t*	*Wi(t)*	*Li(t)*	*Li*	*L*
1	τ_1	10	*C1 = 2*	*0.20*	*0.20*	
2	$\tau_1 \sim \tau_2$	10	*C1+C2 = 5*	*0.50*		
		15	*2C1+C2 = 7*	*0.47*	*0.47*	
3	$\tau_1 \sim \tau_3$	10	*C1+C2+C3 = 10*	*1.00*		
		15	*2C1+C2+C3 = 12*	*0.80*		
		20	*2C1+2C2+C3 = 15*	*0.75*		
		30	*3C1+2C2+C3 = 17*	*0.57*	*0.57*	***0.57***

<표 4.2>는 주기적 태스크 집합 τ={(2, 10), (3, 15), (5, 30)}가 주어진다고 가정할 경우, 결함 허용에 따른 처리기 이용률을 보여 준다.

<표 4.2> FTRMS-EC 처리기 이용률

i	τ_i	*t*	*WRi(t)*	*LRi(t)*	*LRi*	*LR*
1	τ_1	10	*B1+C1 = 4*	*0.40*	*0.40*	
2	$\tau_1 \sim \tau_2$	10	*B1+C1+C2 = 7*	*0.70*		
		15	*B1+B2+2C1+C2 = 10*	*0.70*	*0.70*	
3	$\tau_1 \sim \tau_3$	10	*B1+C1+C2+C3 = 12*	*1.20*		
		15	*B1+B2+2C1+C2+C3 = 15*	*1.00*		
		20	*B1+B2+B3+ 2C1+2C2+C3 = 19*	*0.95*		
		30	*B1+B2+B3+ B4+3C1+2C2+C3 = 23*	*0.77*	*0.77*	***0.77***

결함 허용을 위해 필요한 백업 시간을 고려한 처리기 이용률을 살펴보면 LR_1=0.40, LR_2=min(0.70, 0.70)=0.70, 그리고 LR_3=min(1.20, 1.00, 0.95, 0.77)=0.77이 된다. 따라서 태스크 집합 τ의 전체 처리기 이용률은 LR=max(0.40, 0.70, 0.77)=0.77이 되며 LR<1이므로 주기 태스크 집합 τ는 실행이 가능하다.

4.3 BSTM(Backup-Surplus Time Manager)

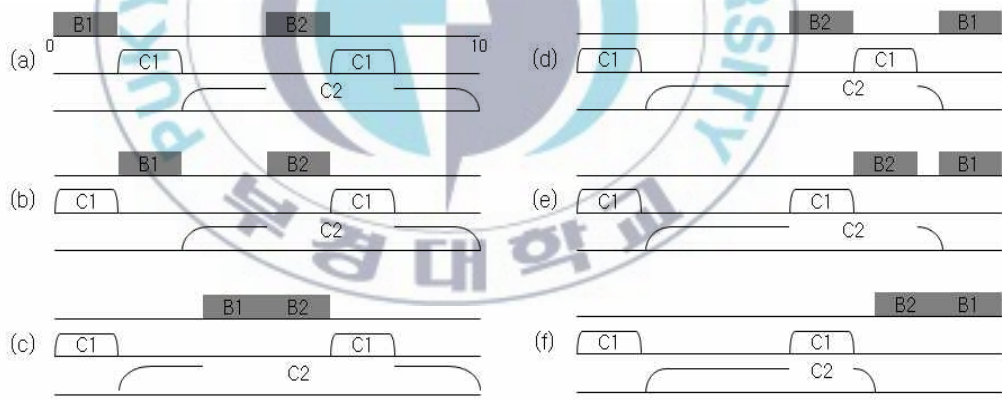
4.3.1 BTM(Backup Time Manager)

BTM은 결함 허용을 위한 백업으로 할당된 타임 슬롯(time slot)을 관리하며 결함 태스크의 요청에 백업 시간을 할당한다. 또한 모든 태스크의 결함 허용을 위해서 Overloading 기법[31]을 이용한다. 이 기법은 결함이 발생한 어떠한 태스크도 백업 시간을 복구에 이용할 수 있다. 이 기법을 이용하지 않을 경우, 각 태스크에 대한 별도의 백업 시간을 제공해 주어야 하며, 태스크들의 처리기 이용률은 전체 태스크의 실행 시간에 대해 두 배로 증가할 것이며 그로 인해 태스크 집합의 실행 가능성은 절반으로 감소할 수 있기 때문이다.

백업 시간은 4.2절의 수식 (7)에 의해서 설정된다. 복구 태스크의 마감시간을 보장하면서 재실행에 충분한 실행 시간을 확보하기 위해서 교체(Swapping) 방법을 이용한다. 즉, 결함이 발생하지 않는 상태에서 태크스들이 실행할 경우 이미 할당된 백업 타임은 실행되는 태

스크의 시간 뒤로 이동한다. 그러므로 향후 태스크 결함이 발생할 경우 복구에 필요한 시간을 확보할 수 있다. 즉, RMS 기법에 결함 허용을 적용하기 위해서 결함이 발생한 태스크에 대한 재실행이 마감시간 내에서 보장되어야 한다.

예를 들어, τ = {(2, 8), (4, 16)}라는 태스크가 있을 경우, 초기 스케줄 타임 테이블은 (a)와 같다. 결함이 발생하지 않을 경우 RMS에 의해 수행된다. 먼저, τ_1이 수행하기 위해서 B1과 C1을 교체하고 첫 번째 주기의 τ_1이 작업을 수행한다(b). 이 후, C2가 수행할 때, B1은 다시 C2와 교체된다((c), (d)). 그리고 두 번째 주기의 C1을 실행하기 위해서 B2와 C1이 C2의 수행하기 위해서 B2와 C2가 교체된다((e), (f)).



<그림 4.7> Swapping(교체) 기법

4.3.2 STM(Surplus Time Manager)

STM은 주기적 태스크 집합의 마감시간을 만족하고 비주기적 태스크의 실행을 보장하기 위해서 처리기 잉여시간을 관리한다. 먼저 주기적 태스크의 실행 시간과 BTM에 의해 백업 시간을 할당 한 후, 처리기 잉여시간을 계산하고 확보된 잉여시간을 비주기적 태스크의 요청에 할당함으로써 비주기적 태스크를 실행할 수 있다.

본 논문에서는 [32]에서 제안한 잉여시간(slack time) 계산 알고리즘을 사용하여 최대 잉여시간을 계산하며 이를 위해 [32]는 다음의 가정을 하고 있다.

- 문맥교환, 태스크 스케줄링을 위한 모든 오버헤드는 0이다.
- 태스크는 태스크간 의존관계가 존재하지 않는 독립적인 태스크이며 주기의 시작시점에 준비상태가 된다.
- 태스크는 보다 높은 우선순위를 가지는 태스크에 의해 선점될 수 있다.
- 비주기적 태스크를 위한 버퍼공간은 무한대이다.

잉여시간을 계산하는 과정은 다음과 같다. 주기 태스크 τ_i의 j번째 요구를 τ_{ij}라고 하면, τ_{ij}의 도착시간은 (j-1)T_i이고, 마감시한은 D_{ij}, 그리고 실행시간은 c_{ij}가 된다. $B(t)$는 시간 [0, t]에서 결함 허용을 위해

할당된 백업시간이며 $P_i(t)$는 시간 [0, t]에서 우선순위가 i인 주기 태스크가 실행되는 시간이 된다. $A_i(t)$는 시간 [0, t]에 우선순위가 i보다 크거나 같은 비주기 태스크들의 전체 실행시간이 된다. $I_i(t)$는 시간 [0, t]에서 우선순위 i보다 낮은 태스크들이 수행되거나 처리기가 유휴 상태인 시간이 된다. 태스크들의 전체 처리기 이용시간은 다음과 같다.

$$W_i(t) = B(t) + A_i(t) + P_i(t) + I_i(t) \qquad (14)$$

$A_{ij}(t)$를 시간 $[0, C_{ij}]$에서 우선순위가 i보다 크거나 같은 비주기 태스크를 처리하기 위한 최대 잉여시간이라고 하면, 수식 (15)의 조건이 성립된다.

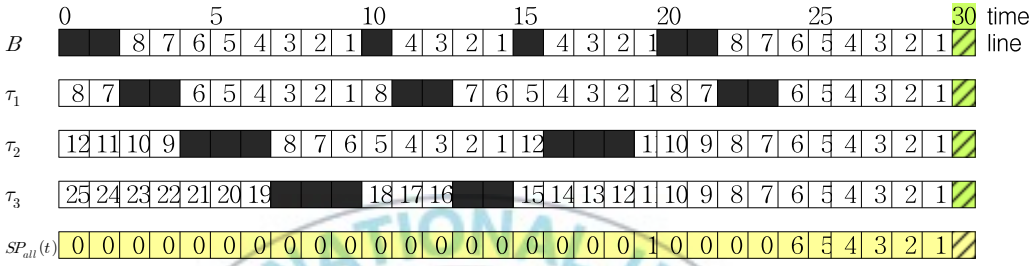
$$A_{ij}(t) = t - B(t) - P_i(t) \qquad (15)$$

실행 시간 t에 따른 처리기의 최대 잉여시간(Surplus time of Processor)은 다음과 같다.

$$SP_{all}(t) = \min_{\{1 \le i \le n\}} A_{ij}(t),\ C_{ij-1} \le t < C_{ij},\ j \ge 1 \qquad (16)$$

예를 들어 주기 태스크 집합 τ={(2, 10), (3, 15), (5 ,30)}가 주어진

다고 가정할 경우, 모든 태스크 τ_i에서 실행 주기 T_i의 최소공배수가 되는 하이퍼 주기(Hyperperiod)는 30이 된다. 그림 4는 수식 (14)~(16)을 통해 계산된 주기 태스크 집합 τ에 대한 잉여시간 테이블을 나타낸다.



<그림 4.8> 잉여시간 할당 테이블

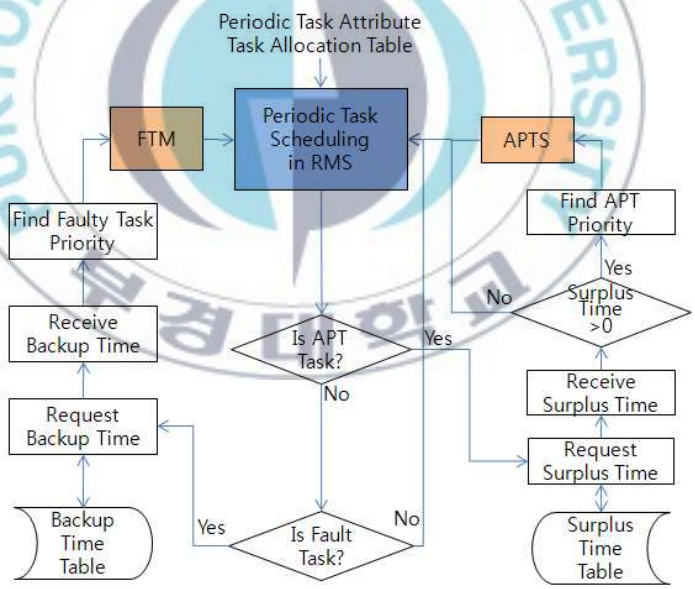
<그림 4.8>에서 검은 상자는 처리기의 이용 상태를 의미한다. 즉, 결함이 발생한 태스크의 복구를 위한 백업 시간과 태스크 실행으로 인한 처리기 이용 상태를 나타낸다. <표 4.2>로부터 τ의 전체 처리기 이용률은 77%이고 하이퍼 주기는 30이므로 비주기 태스크를 실행할 수 있는 처리기 잉여시간은 7(23%)임을 알 수 있다.

4.4 PTS(Periodic Task Scheduler)

주기 태스크 스케줄러인 PTS는 주기적 태스크에 대해 우선순위 기반의 스케줄링을 수행한다. RMS[7]에 의해 부여되는 태스크의 우선순위를 고려하며 태스크 주기가 짧을수록 높은 우선순위를 가진다.

본 논문에서는 태스크의 주기적 특성에 따라 우선순위 테이블을 구분하여 관리한다. 즉, 주기적 태스크와 비주기적 태스크 집합에 대해 각각 별도의 우순순위 테이블을 관리한다.

<그림 4.9>는 PTS의 알고리즘을 나타낸다. 비주기적 태스크와 결함 태스크가 존재하지 않을 경우, RMS에 의해 주기적 태스크 집합을 스케줄링한다. 비주기적 태스크가 실행을 위해 준비큐에 도착하고 비주기적 태스크 실행을 위한 잉여시간이 존재하는 경우, 비주기적 태스크 스케줄러인 APTS를 호출하여 비주기적 태스크를 실행한다. 또한 주기적 태스크가 실행 중에 결함이 발생할 경우 이를 결함 허용 매니저인 FTM에 통보하여 결함 태스크를 재실행함으로써 복구를 수행한다.



<그림 4.9> PTS 수행 흐름도

주기적 태스크 집합이 실행하고 있는 중간에 비주기적 태스크와 결함 복구 요청이 발생할 수 있으므로 이러한 태스크들의 처리 작업은 주기적 태스크의 실행과 상호 동작하도록 처리한다. 또한 비주기적 태스크가 발생하였으나 잉여시간이 존재하지 않을 경우는 비주기적 태스크는 실행할 수 없으므로 결함허용을 위해 FTM에 의해 결함 복구 모드로 전환한다.

4.5 FTM(Fault-Tolerant Manager)

본 논문에서 고려한 결함은 여타 결함들 보다 상대적으로 발생 빈도가 높은 일시적인 결함을 고려하였으며 제안한 결함 허용 기법은 시간 여분 기법을 사용하여 결함 허용 기능을 제공한다. 왜냐하면 임베디드시스템의 특성상 활용할 수 있는 자원이 제한되어 있으며 시간 여분 기법은 추가적인 하드웨어를 요구하지 않기 때문에 비교적 비용이 적기 때문이다. 그리고 태스크의 결함 발생 판단에 필요한 오버헤드는 고려하지 않으며 단일 태스크에 의해 발생하는 결함은 다른 태스크에 영향을 미치지 않는다고 가정한다. 그러므로 이러한 결함은 태스크를 재실행함으로써 복구될 수 있다. 복수의 결함에 대한 허용은 단일 결함 허용에 비해 상대적으로 많은 비용이 필요하며 이로 인해 프로세서의 태스크 처리 응답 시간이 늦어 질 수 있고 그러므로 처리기의 이용률 측면에서 비효율적이기 때문에 단일 결함에 대한 복구만을 고려한다.

FTM은 결함이 발생하였을 경우 BSTM의 구성요소 중 하나인 BTM에서 제공하는 백업 시간을 이용하여 결함이 발생한 태스크를 재실행한다.

<그림 4.10>과 <그림 4.11>은 복구 중인 태스크 τ_r이 재실행되고 있을 때 수행하는 알고리즘을 나타낸다.

<그림 4.10>은 τ_r이 복구 모드에서 새로운 태스크 τ_i가 도착했을 경우 수행하는 알고리즘이다. 태스크들의 정상적인 수행모드뿐만 아니라 복구 모드에서 태스크의 스케줄링은 우선순위에 의한 선점 방식에 의해 이루어진다. 그러나 다음의 경우는 예외이다. 새로운 태스크 τ_i의 우선순위가 복구 태스크 τ_r의 우선순위보다 높고 마감시간이 큰 경우(①)는 우선순위에 의해 스케줄링 되지 않고 복구 태스크의 수행이 완료될 때까지 기다렸다가 실행한다. 즉, 비록 τ_i의 우선순위가 τ_r보다 높더라도 τ_r의 마감시간을 보장하기 위해서 τ_i를 블록 태스크 큐로 전환(②)한 후 기존의 τ_r을 실행한다.

```
New task τi arrives
if (priority−τi>priority−τr) and
  (deadline−τi>deadline−τr) then     ----- ①
     move τi to block_task_queue     ----- ②
else
     add τi to priority_task_queue
execute task as RMS
```

<그림 4.10> τ_r 복구 모드(새로운 태스크 도착)

<그림 4.11>은 τ_r의 복구 실행이 완료(①)되었는지를 판단하여 복구가 완료되면 블록 상태에 있던 태스크 τ_i를 우선순위 태스크 큐로 이동(②)한 후 우선순위 방식에 따라 태스크들을 스케줄링(③)한다. 복구가 완료되지 않았을 경우, 기존의 복구 태스크는 복구 모드에서의 실행을 유지하고 높은 우선순위 태스크는 복구가 완료되기 전까지는 블록 태스크 큐에 저장된 상태를 유지한다.

```
Is recovery task τr finished?
if StatusOfRecovery_r==false then   --- ①
  move blocked task τis
        to priority_task_queue      --- ②
else
  continue Recovery Mode
execute task as RMS                 --- ③
```

<그림 4.11> τ_r 복구 모드(복구 완료 여부 판단)

4.6 APTS(Aperiodic Task Scheduler)

비주기 태스크 스케줄러인 APTS는 비주기적 태스크에 대한 요청이 발생할 경우 이 태스크를 실행하기 위해서 잉여시간 관리자인 STM에 필요한 처리기 시간을 요청하며, STM으로부터 할당받은 처리기 잉여시간에 따라 비주기적 태스크의 생성 및 스케줄 여부를 결

정한다. 비주기적 태스크의 특성에 대해 다음과 같은 가정을 한다.

- 비주기적 태스크의 도착시간은 태스크를 스케줄링하기 전에는 알 수 없다.
- 비주기적 태스크의 실행시간 및 마감시간을 알 수 없다.
- 비주기적 태스크의 실행은 마감시간을 고려하지 않은 즉, 연성 비주기적 태스크를 고려한다.
- 비주기 태스크들은 FIFO(First-In First-Out)순으로 스케줄된다.

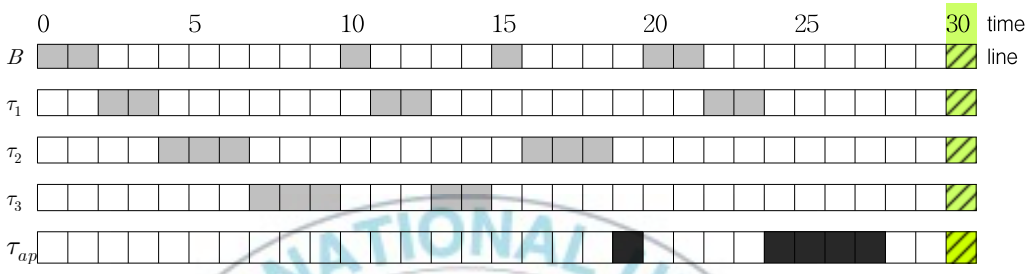
비주기적 태스크를 실행함에 있어서 고려하여야 할 것은 먼저 비주기적 태스크가 실행되더라도 주기적 태스크의 마감시간을 보장하여야 한다는 것이다. 또한 비주기적 태스크의 응답 시간을 최소로 줄일 필요가 있다는 것이다. 마지막으로 주기적 태스크의 응답시간이다. 비주기적 태스크의 짧은 응답시간을 보장할 경우 상대적으로 주기적 태스크의 응답시간은 더 길어진다.

비주기적 태스크를 주기적 태스크로 고려한 우선순위 방법에 따라 스케줄링할 때와 본 논문에서 제안한 RTFT-RTS에서의 방법인 잉여시간을 이용한 비주기적 태스크를 스케줄링하는 경우에 대해 기술한다.

4.3.1절의 주기 태스크 집합 τ=(C, T)={(2, 10), (3, 15), (5, 30)}와 실행시간이 5인 비주기 태스크 τ_{ap}=APT=(C, D)={(5, 25)}를 고려하자.

비주기적 태스크를 비주기적 태스크로 실행하는 경우 비주기 태

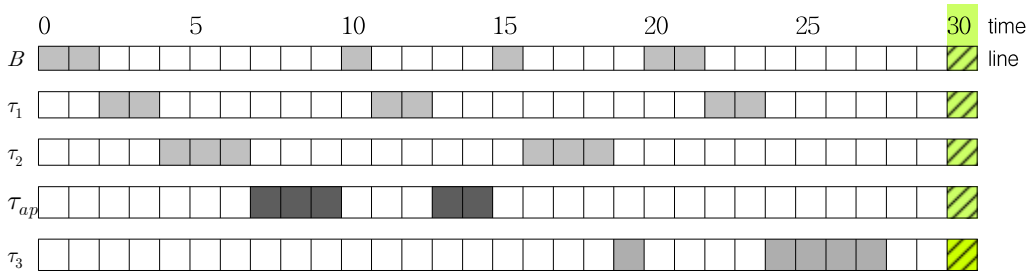
스크를 실행할 수 있는 시간 중에서 가장 빠른 잉여시간은 19가 되며 이 시간부터 비주기 태스크는 스케줄된다. <그림 4.12>와 같이 본 논문에서 제안한 주기 및 비주기 태스크 관리 기법은 비주기 태스크인 τ_{ap}의 실행을 보장함과 동시에 모든 주기적 태스크들의 실행을 마감시간 내에 완료한다.



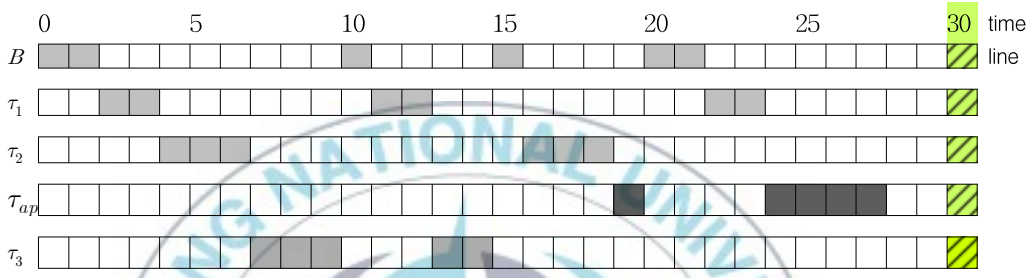
<그림 4.12> 비주기적 태스크를 비주기적 태스크로 실행하는 경우 태스크 할당 테이블

비주기적 태스크를 주기적 태스크로 간주하여 스케줄링할 때의 태스크 할당 테이블을 고려하자. 비주기적 태스크인 APT는 태스크의 마감시간(주기)의 크기에 의해 3번째 우선순위를 가진다. 그리고 비주기적 태스크는 태스크의 특성상 주기적 태스크와 달리 태스크의 도착시간이 일정하지 않다. 그러므로 태스크가 t=7 이전에 도착할 경우 태스크 할당 테이블은 <그림 4.13>이며 태스크가 시간 t=15에서 t=19 사이에 도착하였을 경우의 태스크 할당 테이블은 <그림 4.14>이다. 그러나 비주기적 태스크를 주기적 태스크로 간주하여 우선순위를 부여할 경우 비주기적 태스크로 인해 주기적 태스크의 마감시간을 보장

하지 못하는 경우가 발생할 수 있다는 단점을 지닌다.



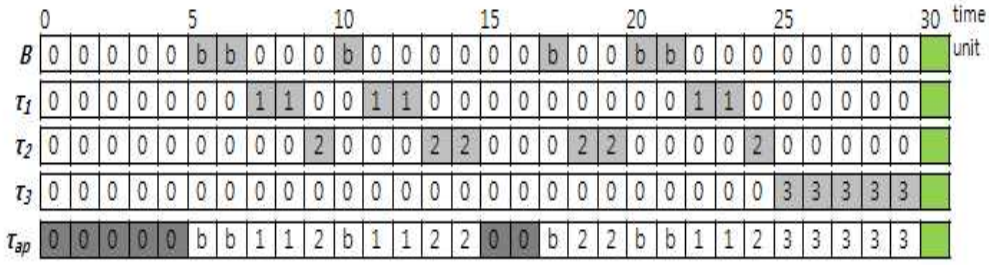
<그림 4.13> 태스크 할당 테이블(APT의 도착시간 : 0~7)



<그림 4.14> 태스크 할당 테이블(APT의 도착시간 : 15~19)

비주기적 태스크를 스케줄링할 경우 주기적 태스크의 실행시간과 마감시간을 이용하여 생성되는 처리기 실행시간을 주기적 태스크에 제공한 후 처리기의 잉여시간을 비주기적 태스크에 할당한다. 또한 주기적 태스크의 마감시간을 보장하면서 최악 마감시간을 적용함으로써 비주기적 태스크의 빠른 응답시간을 제공할 수 있다.

<그림 4.15>는 비주기적 태스크의 빠른 응답시간을 제공하기 위해 적용된 Slack Stealing 알고리즘에 의한 태스크 할당 테이블을 나타낸다.



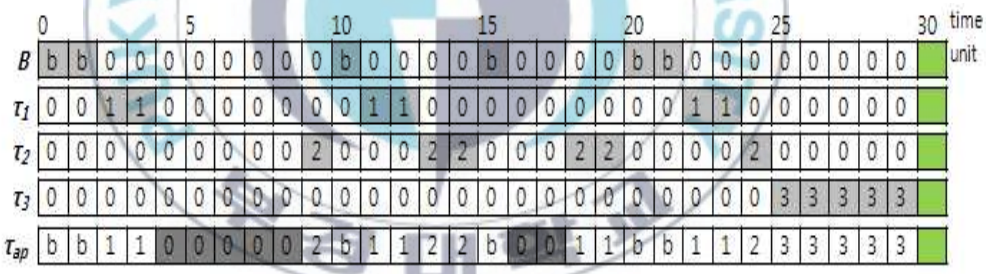
<그림 4.15> 태스크 할당 테이블(Slack Stealing 알고리즘)

τ_2는 첫 번째 실행 주기에서 자신의 마감시간을 보장하기 위해서 시간 11에서 실행하도록 배치하여 최악 마감시간인 시간 15에서 수행 완료를 하게 된다. 위에 제시된 기법에서 주기적 태스크 집합의 태스크들은 각 태스크들의 주기별 마감시간을 보장하면서 실행을 완료한다.

본 논문에서는 주기적 태스크 집합의 태스크 중에서 빠른 응답시간을 필요로 하는 태스크가 있을 경우 이를 제공할 수 있는 방법을 제시한다. 빠른 응답시간을 필요로 하는 주기적 태스크의 경우 중요도를 도입하여 빠른 응답시간을 제공한다. 중요도란 주기적 태스크의 응답시간이며 응답시간이 짧을수록 중요도가 높고 그렇지 않은 경우 중요도가 낮다고 가정한다. 또한 중요도는 태스크 집합을 스케줄링하기 전에 고정적으로 제공된다고 가정한다. 주기적 태스크 집합 중에서 빠른 응답시간을 요구하는 태스크의 경우는 그렇지 않은 주기적 태스크에 비해 빠른 응답시간을 제공하도록 하였다. 빠른 응답시간을 요구하지 않는 주기적 태스크의 경우 최악의 마감시간을 보장하면서

최대한 마감시간 가까이에서 실행하도록 하였다. 그 결과 미리 확보되는 처리기의 잉여시간을 비주기적 태스크에 할당할 수 있다. 즉, 주기적 태스크 집합 중에서 빠른 응답시간을 필요로 하는 태스크에 대한 최선 실행시간을 보장하면서 비주기적 태스크의 응답시간 또한 향상시킬 수 있다.

주기적 태스크 집합 τ=(C, T)={(2, 10), (3, 15), (5, 30)}와 실행시간이 5인 비주기 태스크 τ_{ap}=APT=(C, D)={(5, 25)}를 고려하자. <그림 4.16>은 주기적 태스크 집합 중에서 태스크 τ_1에 대해서 빠른 응답시간을 요구한다라고 가정할 경우의 주기적 태스크의 중요도를 고려한 태스크 할당 테이블이다. 즉, 태스크 τ_1은 가장 빠른 응답시간을 필요로 하며 그렇지 않은 주기적 태스크들은 최악 마감시간 내에 실행을 완료하게 된다.



<그림 4.16> 태스크 할당 테이블(중요도 : 주기적 태스크 τ_1)

백업시간은 주기적 태스크들의 결함이 발생할 경우 결함 태스크를 재실행하는데 필요한 시간으로 주기적 태스크 보다 먼저 할당한다. 그리고 중요도를 가진 태스크 τ_1을 할당한 후 나머지 태스크들은

최악 마감시간을 보장하면서 실행하게 된다. 그러므로 비주기적 태스크를 위한 잉여시간은 시간 4~9와 시간 16~18이며, 이 때 주기적 태스크들의 응답시간은 수식 (17)에 의해 계산된다.

$$ResponseTime\_\tau_i = \sum_{j=1}^{k}(FinishedTime\_\tau_{ij} - ArrivedTime\_\tau_{ij}) \quad (17)$$

$$, \ k = 1,...,Hyperperiod/Period\_\tau_i$$

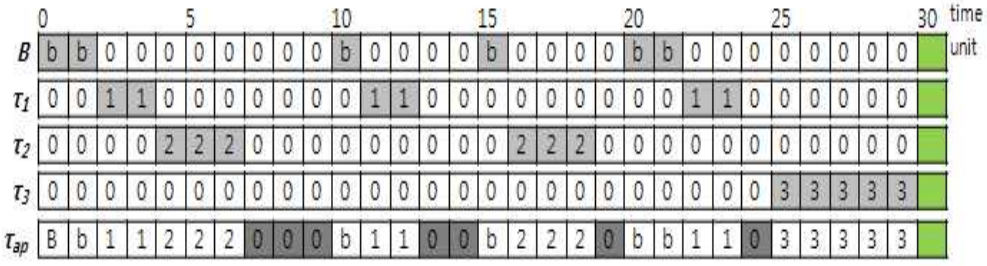
주기적 태스크의 중요도를 τ_1으로 하였을 경우 주기적 태스크들의 응답시간은 다음과 같다.

$$ResponseTime\_\tau_1 = (4-0)+(13-10)+(24-20) = 12$$

$$ResponseTime\_\tau_2 = (15-0)+(25-15) = 25$$

$$ResponseTime\_\tau_3 = (30-0) = 30$$

<그림 4.17>은 주기적 태스크 집합 중에서 태스크 τ_1, τ_2에 대해서 빠른 응답시간을 요구한다라고 가정할 경우의 주기적 태스크의 중요도를 고려한 태스크 할당 테이블이다. 즉, 태스크 τ_1과 τ_2는 가장 빠른 응답시간을 필요로 하며 그렇지 않은 주기적 태스크인 τ_3는 최악 마감시간 내에 실행을 완료하게 된다.



<그림 4.17> 태스크 할당 테이블(중요도 : 주기적 태스크 τ_1, τ_2)

백업시간과 태스크 τ_1과 τ_2의 실행시간을 할당하며 τ_3는 최악 마감시간에 할당한 후 최종 잉여시간을 비주기적 태스크에 할당한다.

<표 4.3>은 주기적 태스크의 중요도를 고려하였을 경우 주기적 태스크들의 응답시간을 비교한 결과이다.

<표 4.3> 주기적 태스크의 중요도에 따른 응답시간 비교

태스크 / 항목	응답시간		
	τ_1	τ_2	τ_3
중요도 τ_1	12	25	30
중요도 τ_2	12	11	30
중요도 τ_3	12	11	15

5. 실시간 태스크 스케줄러 구현

본 절에서는 RTFT-ETS의 구성 요소인 결함 허용을 고려한 주기적 태스크 집합의 실행 가능성 검사, 백업 시간 및 잉여시간 관리자, 주기적 태스크 스케줄러, 태스크의 복구를 수행할 결함 허용 관리자 및 비주기적 태스크 스케줄러를 구현하고 실행 결과를 기술한다.

5.1 구현 환경

RTFT-ETS를 구현하고 테스트하기 위한 하드웨어와 소프트웨어 구현 환경은 <표 5.1>과 같다. 임베디드 하드웨어 시스템은 ARM920T ARM 프로세서를 장착한 CLABSYS사의 의 LN2410SBC 임베디드 보드를 사용하였다.

<표 5.1> 구현환경 - H/W 및 S/W

구분	종류	제품명
하드웨어	임베디드 보드	LN2410SBC
소프트웨어	운영체제	uC/OS-II ver. 2.76
	컴파일러	Borland C 3.1 ADS v1.2
	이미지 전송 툴	ArmDown

본 논문에서 구현한 RTFT-ETS는 uC/OS-II ver. 2.76 운영체제

를 사용하였으며, 이 운영체제의 내부 태스크 스케줄러를 수정 및 추가하였다. 컴파일링을 위해 PC에서는 Borland C 3.1 컴파일러를, 그리고 임베디드 보드에 사용할 이미지 생성을 위해 Metrowerks사의 ADS v1.2(Arm Developer Suite v1.2)를 사용하여 소스코드를 컴파일하였다. ADS v1.2를 이용하여 이미지를 생성하였으며 이미지를 임베디드 보드로 전송하기 위해서 ArmDown 프로그램을 사용하였다.

5.2 ECM(Executability Check Manager)

본 절에서는 4.2절에서 설계한 실행 가능성 검사 모듈을 구현하고 그 결과를 분석한다. 주기적 태스크 집합의 실행 가능성 검사를 위해 Olympus 위성의 고도 및 궤도 제어 시스템(AOCS: Attitude and Orbital Control System)[33]에 적용된 태스크 집합을 사용하였다.

<표 5.2>는 본 논문에서 제안한 실시간 태스크 스케줄러를 시뮬레이션하기 위해서 사용하는 태스크 집합을 나타낸다. AOCS에 적용된 태스크들은 주기와 마감시간이 동일하지 않다. 그러나 본 논문에서는 주기와 마감시간은 동일하다고 가정하였으므로 AOCS에 적용된 태스크들의 속성 중에서 태스크 주기와 최악의 수행 시간만을 고려하여 실행 가능성 검사를 실시하였다. 본 논문에서는 구현 결과를 시뮬레이션할 때, 타임 틱(tick) 발생 시간을 10ms단위로 적용하여 테스트하였다. 즉, 초당 100번의 타임 틱이 발생하도록 설정하였다. 따라서 시뮬레이션을 위한 주기적 태스크의 실행 시간을 정수값으로 처리할

필요가 있으므로 AOCS에서 제시한 실행 시간보다 큰 정수 중에서 최소값을 태스크의 실행시간으로 선정하였다. 또한 태스크의 우선순위는 RMS에 의해 부여하였으며 우선순위가 동일한 태스크의 경우 임의의 우선순위를 부여하여 태스크 집합을 재구성하였다.

<표 5.2> AOCS 실시간 태스크 집합(단위: ms)

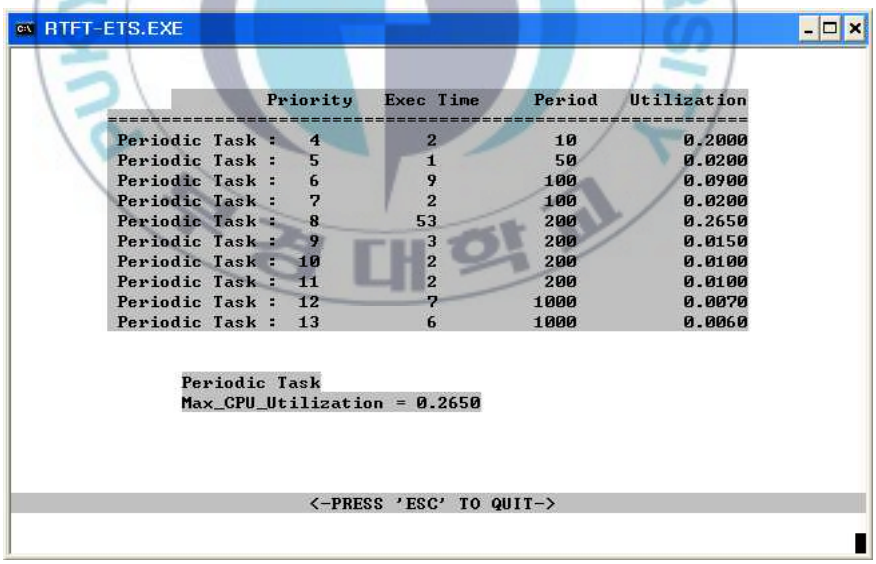
우선순위	태스크 이름	최악의 경우의 수행시간	주 기
4	Read_Bus_IP	2	10
5	Real_Time_Clock	1	50
6	Process_IRES_data	9	100
7	Request_IRES_data	2	100
8	Control_Law	53	200
9	Command_Actuators	3	200
10	Request_DSS_Data	2	200
11	Request_Wheel_Speeds	2	200
12	Calibrate_Gyro	7	1000
13	Process_DSS_data	6	1000

주기적 태스크 집합의 실행 가능성을 검사하기 위해서 백업 시간 및 태스크 집합의 처리기 시간을 계산한다. 태스크 결함을 위한 백업 시간을 계산하기 위해서 태스크 집합 중에서 최대 처리기 이용률을 가진 처리기 이용률을 계산하며 알고리즘을 구현한 내용은 <그림

5.1>과 같다. 라인 6에서 각 처리기의 처리기 이용률을 계산하고 라인 9에서 최대 처리기 이용률을 계산한다.

```
1 : void Max_CPU_Utilization_Task(void)
2 : {
3 :  FP32 UB = 0.0;
4 :  while(Task[i]!=0)
5 :  {
6 :   UB = (FP32)OSTCBPrioTbl[Task[i]]->ExecTime
7 :         /(FP32)OSTCBPrioTbl[Task[i]]->TPeriod;
8 :   if (UB > Max_CPU_TASK_U)
9 :    Max_CPU_TASK_U = UB;
10:   i++;
11:  }
```

<그림 5.1> 주기적 태스크 집합에서 최대 처리기 이용률을 가지는 태스크의 처리기 이용률 계산



<그림 5.2> 최대 처리기 이용률 계산 결과 화면

<그림 5.2>는 각 태스크의 처리기 이용률과 최대 처리기 이용률을 계산하는 구현 결과다. <표 5.2>에서 제시된 태스크 중에서 최대 처리기 이용률을 가진 태스크는 우선순위가 8인 태스크이며 처리기 이용률은 26.5%이다.

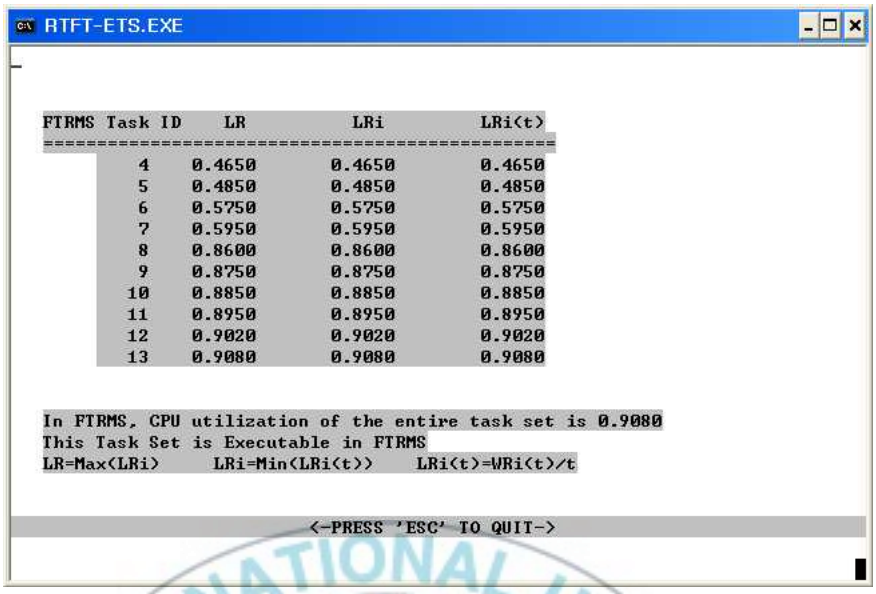
<그림 5.3>은 <그림 5.1>에서 계산된 최대 처리기 이용률과 주기적 태스크의 처리기 이용률을 이용하여 태스크 집합의 실행 가능성을 검사하기 위한 알고리즘을 구현한 것이다. 라인 3~7에서 태스크 i와 태스크 j의 주기값을 이용하여 k값을 계산하고 0~k값을 이용하여 스케줄링 포인트인 S_i값을 계산한다. 라인 19에서 시간 t에 대해 태스크의 처리기 이용 시간($W_i(t)$)을 계산하고 라인 20에서 처리기 이용 시간 $W_i(t)$에 백업 시간을 포함한 태스크의 처리기 이용 시간($WR_i(t)$)과 처리기 이용률인 $WRI_t(LR_i(t) = WR_i(t)/t)$을 계산한다. 라인 27~28에서 각 스케줄링 포인트 S_i에서 처리기 이용률이 최소값인 LRi ($LR_i = \min(LR_i(t))$)를 계산하고 라인 21~22에서 전체 주기적 태스크 집합의 처리기 이용률인 $LR(LR = \max(LR_i))$을 계산한다. 계산된 LR값을 이용하여 태스크 집합의 실행 가능성을 검사한다.

LR값이 1.0보다 작거나 같으면($\leq 100\%$) 주기적 태스크 집합 내의 태스크들은 각 태스크 자신의 마감시한을 만족하면서 스케줄이 가능함을 의미하므로 TRUE 값을 반환하고, 1.0보다 크면(>100%) 태스크 집합의 실행이 불가능하므로 FALSE 값을 반환한다.

```
1 :  BOOLEAN Exec_Check_FTRMS(void)
2 :  { for (i = 0; i < OSTaskCtr-2; i++)
3 :    {   for (j = 0; j <= i; j++)
4 :      {     t_index=(OSTCBPrioTbl[Task[i]]->TPeriod/
5 :                    OSTCBPrioTbl[Task[j]]->TPeriod);
6 :       for (k = 1; k <= t_index; k++)
7 :         S[index++] = k * OSTCBPrioTbl[Task[j]]->TPeriod; }
8 :      for (k = 0; k < index; k++)
9 :      { WRI_t= 0.0;
10:       sealing_distance = 0.0;
11:       for (j = 0; j <= i; j++)
12:       { ExecTime = OSTCBPrioTbl[Task[j]]->ExecTime;
13:         TPeriod = OSTCBPrioTbl[Task[j]]->TPeriod;
14:         sealing_float = (S[k]/TPeriod);
15:         sealing_int = S[k]/OSTCBPrioTbl[Task[j]]->TPeriod;
16:         sealing_distance = sealing_float - sealing_int;
17:         if (sealing_distance > 0.0)
18:             sealing_float += 1.0 - sealing_distance;
19:       WRI_t+= (ExecTime * sealing_float / S[k]);}
20:       WRI_t+= Max_CPU_TASK_U;
21:       if (WRI_t < LRi)
22:           LRi= WRI_t
23:      }
24:      if (LRi > LR)
25:        LR= LRi
26:    }
27:    if LR<= 1.0)    return TRUE;
28:    else return FALSE;
29:  }
```

<그림 5.3> FTRMS-EC 구현

<그림 5.4>는 <표 5.2>에서 제시한 주기적 태스크 집합을 이용하여 FTRMS-EC를 실행한 결과로써 태스크 집합의 처리기 이용률은 0.908이며 이는 이 태스크 집합이 실행 가능함을 나타내고 있다.



<그림 5.4> 태스크 집합의 실행 가능성 판단 결과(실행 가능)

<그림 5.5> 태스크 집합의 실행 가능성 판단 결과(실행 불가능)

<그림 5.5>는 <표 5.2>에서 제시한 주기적 태스크 집합 중에서 우선순위 5번 태스크의 실행시간을 1에서 10으로, 7번 태스크의 실행시간을 2에서 20으로 변경하여 시뮬레이션한 결과 태스크 집합의 전체 처리기 이용률이 1.01로 나타나 실행이 불가능한 태스크 집합임을 보여준다.

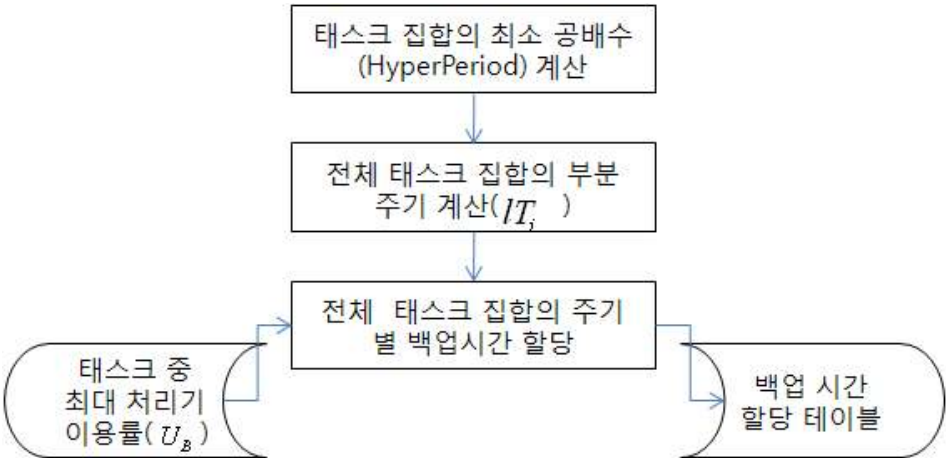
5.3 BSTM(Backup-Surplus Time Manager)

5.3.1 BTM(Backup Time Manager)

BTM은 백업시간을 생성한 후 태스크의 결함이 발생할 경우 태스크 스케줄러로부터 요청되는 백업시간을 관리한다. 백업 시간을 관리하기 위해서 백업 시간이 할당된 테이블을 생성하며 수행 흐름도는 <그림 5.6>과 같다.

백업 시간 할당 테이블을 생성하기 위해서 먼저 주기적 태스크 집합에 포함된 모든 태스크들의 주기에 대한 최소 공배수를 계산한다. 이 값은 각 태스크들의 속성 중에서 마감시간을 이용하여 계산되며 하이퍼 주기(Hyper Period)라고 한다. 이는 주기적 태스크 집합에 포함된 모든 태스크들의 최소 공통 주기가 된다.

하이퍼 주기가 생성되면, 이 주기값에 대하여 각 주기적 태스크들의 주기를 반영한 전체 태스크 집합의 부분 주기값을 계산한다.



<그림 5.6> 백업 시간 테이블 생성 흐름도

마지막으로 5.2절에서 계산된 태스크 중에서 최대 처리기 이용률을 각 부분 주기 범위에 할당함으로써 백업 시간 할당 테이블을 생성한다.

```
1 : void Calculate_HyperPeriod()
2 : {
3 :     arg1 = OSTCBPrioTbl[Task[0]]->TPeriod;
4 :     arg2 = OSTCBPrioTbl[Task[1]]->TPeriod;
5 :     gcd = Compute_GCD(arg1, arg2);
6 :     lcm = (arg1 * arg2) / gcd;
7 :     for (i = 2; i < N_PERIODIC_TASKS; i++)
8 :     {
9 :             arg1 = OSTCBPrioTbl[Task[i]]->TPeriod;
10:             gcd = Compute_GCD(lcm, arg1);
11:             lcm = lcm * (arg1 / gcd);
12:     }
13:     HYPER_PERIOD = lcm;
14: }
```

<그림 5.7> 태스크 집합의 하이퍼 주기(Hyper Period) 구현

<그림 5.7>은 모든 주기적 태스크들에 대한 하이퍼 주기(Hyper Period)를 계산하는 알고리즘을 구현한 것이다.

라인 5에서 주기적 태스크 집합에서 우선순위가 가장 높은 두 개의 태스크의 마감시간을 이용하여 최대 공약수를 계산(<그림 5.8>)한 후, 이 값을 이용하여 태스크 집합의 나머지 모든 태스크들의 마감시간을 차례대로 적용하여 라인 7~12에서 하이퍼 주기를 계산한다. <표 5.2>에서 제시한 주기적 태스크 집합을 이용한 하이퍼 주기는 1000이 된다.

```
1 : INT32U Compute_GCD(INT32U u, INT32U v)
2 : {
3 :     INT32U tmp;
4 :     if (u>v){
5 :             tmp = u; u=v; v=tmp; }
6 :     while(v)
7 :     {
8 :         tmp = u%v; u = v; v = tmp; }
9 :     return u;
10: }
```

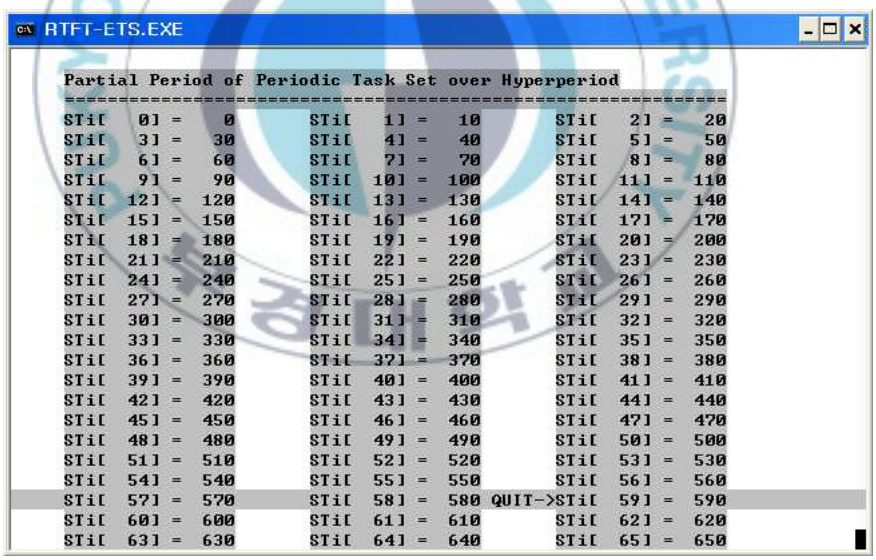
<그림 5.8> 최대 공약수(GCD) 구현

<그림 5.9>는 하이퍼 주기에 대한 전체 태스크 집합의 부분 주기 값(lTi)을 계산하는 알고리즘을 구현한 것이다. 라인 8에서 주기적 태스크의 주기 값을 추출한 후, 라인 15에서 주기 값을 할당하고 향후 백업시간을 계산하는데 활용한다.

```
1 : void All_PTask_Period()
2 : {
3 :  for (j = 0; j < OSTaskCtr-2; j++)
4 :  {
5 :   for(t=1; t<= HYPER_PERIOD; t++)
6 :   {if((t % OSTCBPrioTbl[Task[j]]->TPeriod)==0)
7 :    {if(S_tmp[t]==0)
8 :     {S_tmp[t] = k*OSTCBPrioTbl[Task[j]]->TPeriod; }
9 :     k=k+1;
10:    }  } }
11:  STi[0] = 0;
12:  for(t=1; t<= HYPER_PERIOD; t++)
13:  {
14:   if(S_tmp[t]!=0)
15:    STi[S_Index++]=S_tmp[t];
16:  } }
```

<그림 5.9> 태스크 집합의 부분 주기(ITi) 계산 알고리즘 구현

<그림 5.10>은 주기적 태스크 집합의 전체 부분 주기를 계산한 결과다.



<그림 5.10> 전체 태스크 집합의 부분 주기 계산 결과

<그림 5.11>은 백업 시간 할당 테이블을 생성하는 알고리즘을 구

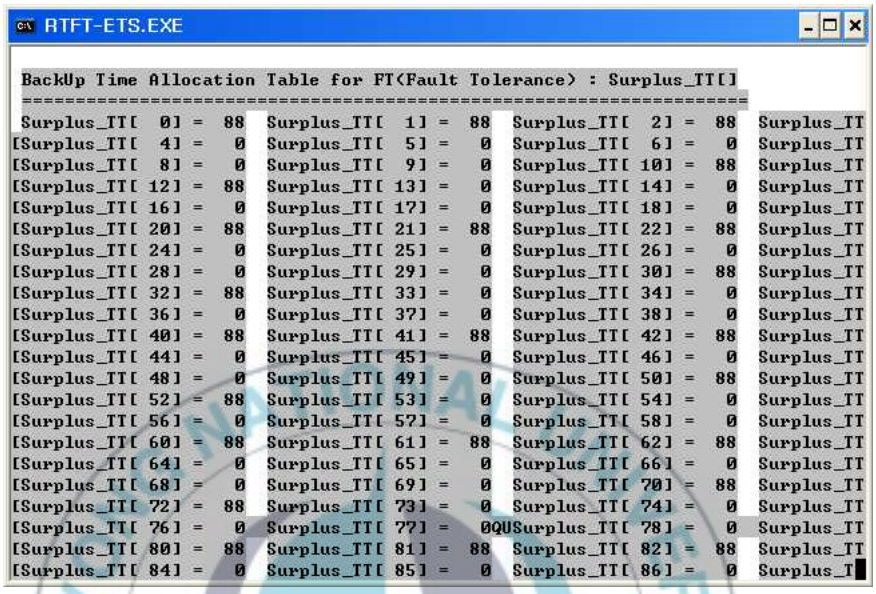
현한 것이다. 라인 6에서 하이퍼 주기에 대해 모든 두 주기 사이의 시간 간격($lT_j - kT_i$)를 계산한 후, 이 값과 최대 처리기 이용률을 이용하여 라인 7에서 백업 시간을 계산한다(수식 (7)). 계산된 백업 시간을 라인 13에서 하이퍼 주기의 크기와 동일한 시간 할당 테이블에 할당함으로써 백업 시간 할당 테이블을 생성한다.

```
1 : void Make_Backup_TimeTable()
2 : { for (t = 0; t < HYPER_PERIOD; t++)
3 :    { Surplus_TimeTable[t] = 0; }
4 :    for(i=0;i<S_Index-1;i++) {
5 :      k = i+1; t = STi[i];
6 :      Length_Two_Tasks = STi[k]-STi[i];
7 :      FP_Backup_Length_Two_Tasks=Length_Two_Tasks*Max_CPU_TASK_U;
8 :      Int_Backup_Length_Two_Tasks=(INT32U) FP_Backup_Length_Two_Tasks;
9 :      if((FP_Backup_Length_Two_Tasks - Int_Backup_Length_Two_Tasks)!=0)
10:       Int_Backup_Length_Two_Tasks++;
11:      for(j=0;j<Int_Backup_Length_Two_Tasks;j++)
12:      { tt = t+j;
13:        Surplus_TimeTable[tt] = 88; }}
14: }
```

<그림 5.11> 백업 시간 할당 테이블 생성 구현

<그림 5.12>는 생성된 백업 시간 할당 테이블을 시뮬레이션 한 결과이다. 하이퍼 주기에 대한 타임 슬롯을 출력한 결과의 일부를 나타낸다. 그림에서 나타나는 값 중에서 “0”으로 표기된 곳은 주기적 태스크와 비주기적 태스크의 처리에 할당될 타임 슬롯이며, “88”로 표기

된 곳은 결함이 발생한 태스크를 재실행하기 위해 확보되는 백업 시간을 나타내는 타임 슬롯이다.



<그림 5.12> 백업 시간 할당 테이블 생성 결과

5.3.2 STM(Surplus Time Manager)

비주기적 태스크의 실행과 완료를 보장하기 위해서 처리기 잉여 시간을 계산한다. 5.3.1절에서 생성된 백업 시간 할당 테이블을 이용하여 주기적 태스크 집합의 실행 시간을 할당한 후, 처리기 잉여시간을 계산한다.

<그림 5.13>은 비주기적 태스크의 실행을 위해 생성되는 처리기

잉여시간 생성 알고리즘을 구현한 것이다.

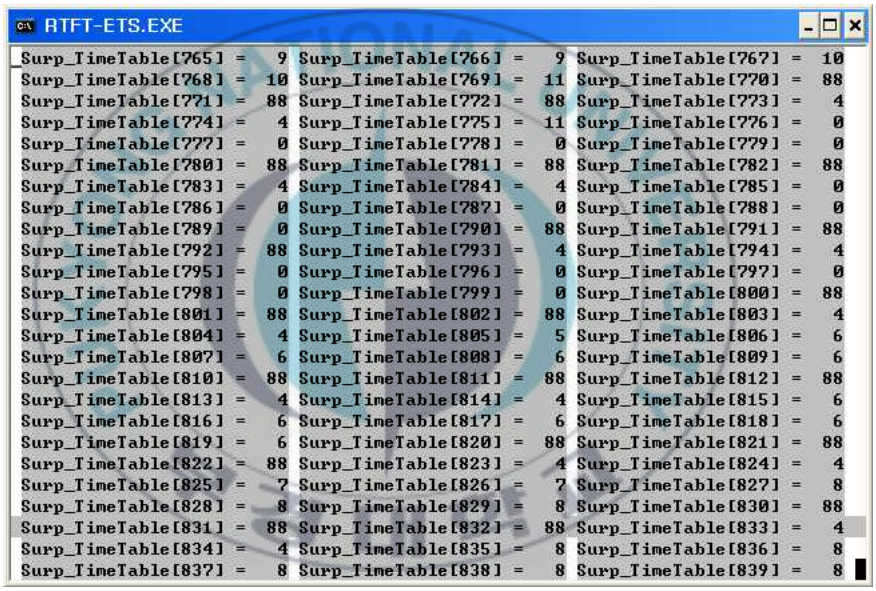
```
1 :  void Calculate_Surplus_TimeTable_FTRMS()
2 :  { for (i = 0; i < OSTaskCtr-2 ; i++) {
3 :    for (t = 0; t < HYPER_PERIOD; t++) {
4 :    if ((t % OSTCBPrioTbl[Task[i]]->TPeriod) == 0) {
5 :     for (j = 0; j <= i; j++) {
6 :      for (k = t; k < (t + OSTCBPrioTbl[Task[i]]->TPeriod); k++) {
7 :       if ((k % OSTCBPrioTbl[Task[j]]->TPeriod) == 0) {
8 :        if ((OSTCBPrioTbl[Task[j]]->ExecTime + k)
9 :            > (t + OSTCBPrioTbl[Task[i]]->TPeriod))
10:          task_time+=OSTCBPrioTbl[Task[j]]->ExecTime
11:           -((k + OSTCBPrioTbl[Task[j]]->ExecTime)
12:           -(t + OSTCBPrioTbl[Task[i]]->TPeriod));
13:        else task_time += OSTCBPrioTbl[Task[i]]->ExecTime;  }}}
14:        comp_time = OSTCBPrioTbl[Task[i]]->ExecTime; }
15:      if (Surplus_TimeTable[t] == 0) {
16:      if (comp_time != 0) {
17:       Surplus_TimeTable[t] = OSTCBPrioTbl[Task[i]]->OSTCBPrio;
18:       comp_time--; }}}}
19:  }
```

<그림 5.13> 잉여시간 생성 알고리즘 구현

백업 시간이 할당된 백업 시간 할당 테이블에서 우선순위가 높은 태스크에서 우선순위가 낮은 태스크로 각 태스크별 처리기 실행 시간을 할당하게 된다. 라인 2에서 우선순위가 높은 태스크부터 선택한 후, 각 태스크의 마감시간을 라인 4에서 판단한다. 현재 시간이 각 주기의 마감시간과 동일하다면, 라인 5~14에서 각 태스크의 실행 시간

과 잉여시간을 계산하고 라인 17~18에 잉여시간 할당 테이블에 실행 시간과 잉여시간을 할당하게 되며 이러한 과정은 모든 주기적 태스크 집합의 모든 태스크들에 대해서 반복 수행함으로써 최종 잉여시간 할당 테이블을 생성한다.

<그림 5.14>는 비주기적 태스크의 실행을 위해 생성되는 처리기 잉여시간 생성 알고리즘을 구현한 결과로 생성되는 잉여시간 할당 테이블 화면이다.

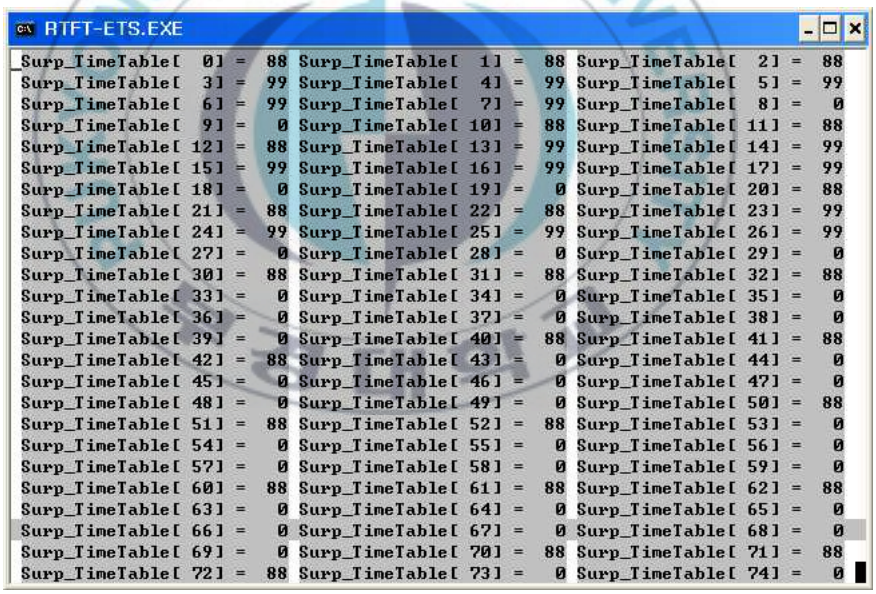


<그림 5.14> 잉여시간 할당 테이블 생성 결과

그림에서 나타난 “4”에서 “11”로 할당된 값은 주기적 태스크 집합의 우선순위 4에서 11까지의 태스크를 할당한 타임 슬롯을 나타내며,

“88”은 백업 타임 슬롯, 그리고 “0”으로 표시되는 부분이 잉여시간을 나타내는 타임 슬롯이다.

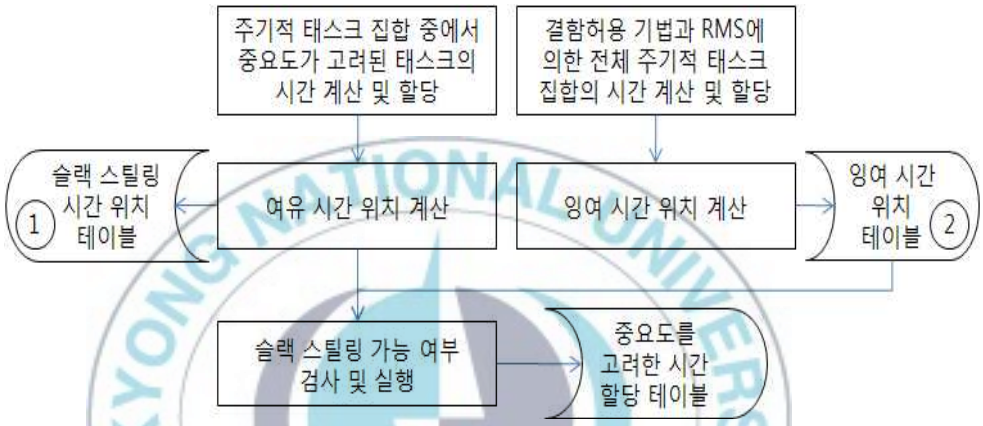
<그림 5.15>는 비주기적 태스크의 빠른 응답시간을 보장하기 위해 슬랙 스틸링(Slack Stealing) 기법을 적용하였을 경우의 잉여시간 할당 테이블 생성 결과를 나타낸다. 타임 슬롯 8부터 9까지의 “0”으로 표시된 부분은 주기적 태스크의 처리를 위해 사용되는 처리기 실행 시간이며, “99”라는 값으로 표시된 부분이 비주기적 태스크의 실행을 위한 타임 슬롯이 된다. 즉, 비주기적 태스크는 타임 슬롯 3부터 실행이 가능하다. 비주기 태스크는 주기적 태스크 보다 먼저 수행함으로써 빠른 응답시간을 제공할 수 있다.



<그림 5.15> 잉여시간 할당 테이블(슬랙 스틸링 기법 적용)

4.6절에서 주기적 태스크의 중요도에 대해서 기술하였다. RMS에

서 제안한 우선순위 기법을 기본 알고리즘으로 적용하며 또한 주기적 태스크 집합 중에서 빠른 응답시간을 필요로 하는 태스크에 대해 비주기적 태스크보다 먼저 실행을 보장함으로써 빠른 응답 시간을 제공한다. <그림 5.16>은 주기적 태스크의 중요도를 고려한 태스크 할당 테이블 생성 흐름도를 나타낸다.



<그림 5.16> 태스크의 중요도를 고려한 태스크 할당 테이블 생성 흐름도

먼저 결함 허용을 위해 확보된 백업 시간과 주기적 태스크 집합 중에서 중요도가 고려된 태스크의 실행 시간을 이용한 타임 슬롯을 계산함으로써 하이퍼 주기에서 슬랙 스틸링 검사에 필요한 타임 슬롯을 확보한다. 다음으로 결함 허용 기법과 RMS에 의한 전체 주기적 태스크 집합의 타임 슬롯을 계산한다. 이 과정은 <그림 5.13>의 잉여 시간 생성 알고리즘 구현을 통해 획득할 수 있으며 이 결과를 이용하여 잉여시간 위치 테이블을 생성한다. ①의 값과 ②의 값을 이용하여

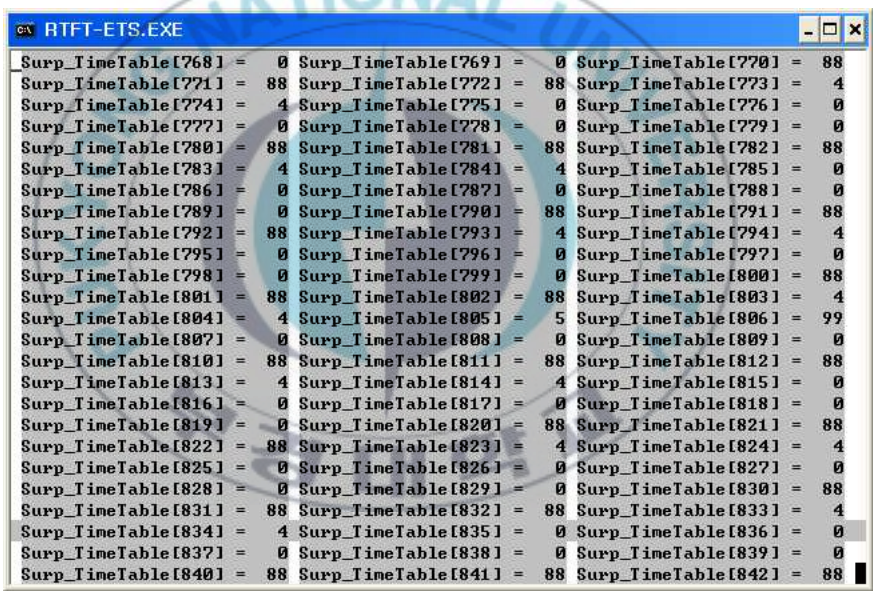
슬랙 스틸링이 가능한지를 검사하게 된다.

<그림 5.17>은 주기적 태스크의 중요도를 고려한 태스크 할당 테이블을 생성하는 알고리즘을 구현한 것이다.

```
1 :   void Surplus_Time_Table_FTRMS_For_Important()
2 :   { INT32U ETS;
3 :     INT32U N_Surplus_Time_Slot;
4 :     INT32S Position_Surplus_Time_Slot[MAX_HYPERIOD];
5 :     INT32U Position_Empty_Time_Slot[MAX_HYPERIOD];
6 :     INT32U N_Empty_Time_Slot;
7 :     for(i = 0; i <  N_Surplus_Time_Slot; i++){
8 :     for(j = ETS; j < Position_Surplus_Time_Slot[i]; j++){
9 :      Imp_Surplus_TimeTable[Position_Surplus_Time_Slot[i]] = 0;
10:      Imp_Surplus_TimeTable[Position_Empty_Time_Slot[j]] = 77;
11:       for (k = Not_Important_Task; k < OSTaskCtr-2 ; k++) {
12:       for(m = 0; m < HYPER_PERIOD; m++){
13:        if ((m % OSTCBPrioTbl[Task[k]]->TPeriod) == 0){
14:         Task_i_ExecTime = OSTCBPrioTbl[Task[k]]->ExecTime;
15:         for(p = m; p < (m + OSTCBPrioTbl[Task[k]]->TPeriod); p++){
16:          if((Imp_Surplus_TimeTable[p] == 0) && (Task_i_ExecTime !=0)){
17:           Imp_Surplus_TimeTable[p]
18:                    = OSTCBPrioTbl[Task[k]]->OSTCBPrio;
19:           Task_i_ExecTime--;}}
20:          if(Task_i_ExecTime!=0) Check_Slack_Stealing = FALSE;}}}
21:        ETS++;
22:        if(Check_Slack_Stealing){
23:         Surplus_TimeTable[Position_Empty_Time_Slot[j]] = 99;
24:         Surplus_TimeTable[Position_Surplus_Time_Slot[i]] = 0;}}}}
```

<그림 5.17> 태스크의 중요도를 고려한 태스크 할당 테이블 생성 알고리즘 구현

라인 7에서 기존의 백업 시간과 주기적 태스크의 중요도를 고려한 태스크가 할당된 타임 슬롯을 제외한 모든 시간에 대하여 슬랙 스틸링이 가능한지를 검사하기 위해 반복 수행한다. 라인 7~9에서 각 타임 슬롯에 대해 슬랙 스틸링이 되었다고 가정하고 중요도를 고려하지 않는 주기적 태스크들에 대해 마감시간 내에서 실행 완료가 되는지를 검사한다. 라인 22~24에서 만약 실행이 완료된다면 선택된 타임 슬롯은 슬랙 스틸링이 가능하다고 판단하여 비주기적 태스크를 위한 잉여시간으로 할당하게 된다.

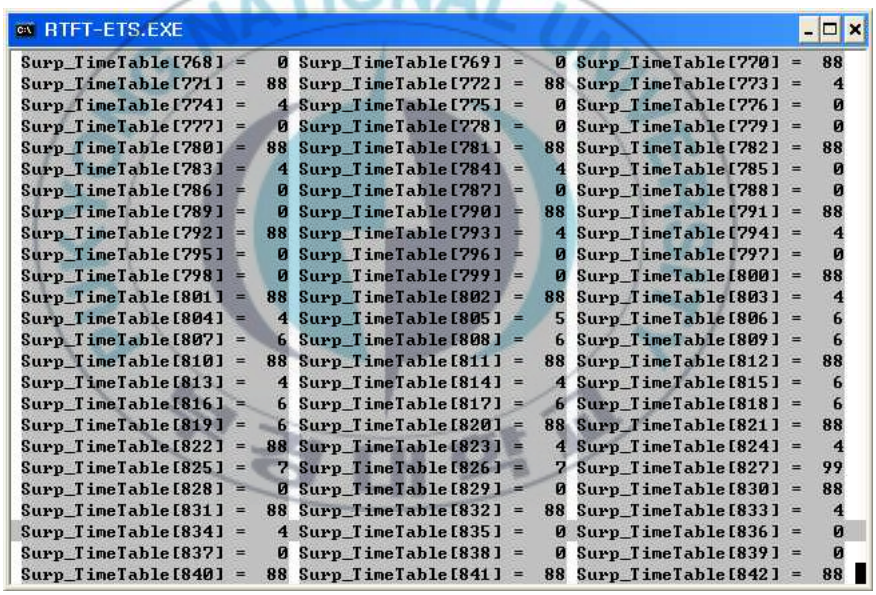


<그림 5.18> 중요도를 고려한 태스크 할당 테이블(중요도 2)

<그림 5.18>은 주기적 태스크의 중요도를 2로 설정하였을 경우의

결과이다. 예를 들어, 타임 슬롯 803에서 805까지 나타난 값은 우선순위 4와 5에 해당하는 주기적 태스크의 실행 시간을 의미하며, “99”로 나타난 값인 타임 슬롯 806은 비주기적 태스크의 실행을 위해 확보되는 타임 슬롯이다. 타임 슬롯 807에서 809까지 나타난 값인 “0”은 비주기적 태스크를 먼저 실행한 후, 우선순위 6이상의 태스크들이 실행하게 될 타임 슬롯이 된다.

<그림 5.19>는 주기적 태스크의 중요도를 4로 설정하였을 경우의 결과 화면이다.

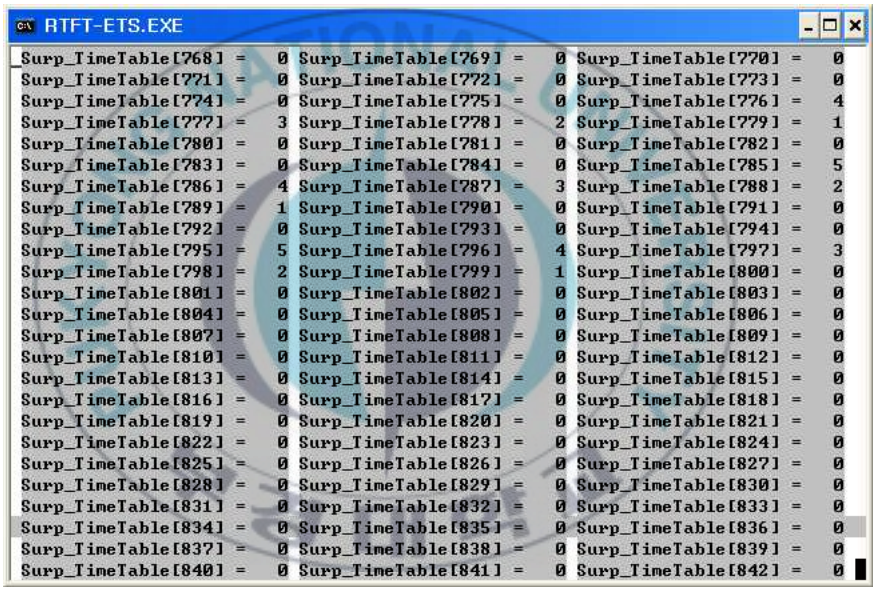


<그림 5.19> 중요도를 고려한 태스크 할당 테이블(중요도 4)

빠른 응답시간을 요구하지 않는 주기적 태스크의 경우 최악의 마감시간을 보장함으로써 그로 인해 생성된 처리기의 잉여시간을 비주

기적 태스크에 할당한다. 즉, 주기적 태스크 집합의 마감시간을 보장하면서 주기적 태스크 집합 중에서 빠른 응답시간을 필요로 하는 태스크에 대한 최선 실행시간을 보장하면서 비주기적 태스크의 응답시간 또한 향상시킬 수 있다.

<그림 5.20>은 주기적 태스크의 중요도를 8로 설정하였을 경우 비주기적 태스크를 실행할 수 있는 잉여시간을 파악하는데 사용되는 최종 잉여시간 할당 테이블이다.



<그림 5.20> 중요도를 고려한 잉여시간 할당 테이블(중요도 8)

예를 들어, 타임 슬롯 776에서 779까지 4개의 타임 슬롯은 비주기적 태스크에 할당할 수 있는 잉여 시간이다. 즉, 비주기적 태스크의

처리기 요청이 776에서 발생할 경우 시간 4만큼을 할당받아 수행하고 이후에 잉여 시간이 발생할 때까지 블록 상태가 된다.

5.4 PTS(Periodic Task Scheduler)

선점형 우선순위 기반의 임베디드 실시간 운영체제인 u/COS-II는 태스크의 실행 주기를 관리하는 방법을 제공하지 못하고 있다. 본 논문에서 설계한 스케줄러를 시뮬레이션하기 위해서는 태스크 속성에 대한 추가적인 자료구조가 포함되어야 한다. 이를 위해서 TCB(Task Control Block)에 태스크 주기와 실행시간 관리를 위한 변수들을 추가하였다. u/COS-II의 TCB 구조에 추가된 태스크 속성 정보를 <그림 5.21>의 라인 5~8에서 나타내고 있다.

```
1 :  typedef struct os_tcb {
2 :   /* Pointer to current top of stack*/
3 :     OS_STK   *OSTCBStkPtr;
4 :                    :
5 :     INT32U   TPeriod;          // 태스크의 주기
6 :     INT32U   RemainTPeriod; // 태스크의 실행 후 남은 주기
7 :     INT32U   ExecTime;         // 태스크 실행 시간
8 :     INT32U   RemainExecTime; // 실행시간 후의 남은 시간
9 :                    :
10:  } OS_TCB;
```

<그림 5.21> TCB 구조

TPeriod는 태스크의 주기를 나타내며 RemainTPeriod 변수는 태스크의 실행이 진행되면서 얻게 되는 실행 후 남은 주기를 나타낸다. ExecTime는 태스크의 실행 시간을 가지는 변수이며 RemainExecTime 변수는 태스크의 실행이 진행되면서 주기 내에서 실행해야할 남은 실행 시간을 나타낸다.

<그림 5.22>는 uC/OS-II의 태스크 생성 함수다. 본 논문에서는 생성되는 태스크의 주기와 실행시간 정보를 전달하기 위해서 두 번째 인자인 *pdata를 이용한다. 이 포인터는 태스크에게 전달될 인자값을 가리킨다.

```
OSTaskCreate (void (*task)(void *pd),  void *pdata,
                    OS_STK *ptos,  INT8U prio);
```

<그림 5.22> 태스크 생성 함수 선언

<그림 5.23>에서 태스크의 주기와 실행 시간을 설정하기 위해서 uC/OS-II에서 태스크 초기화를 수행하는 OS_TCBInit() 함수를 이용하여 태스크의 주기와 실행시간 설정을 나타낸다. OS_TCBInit() 함수는 OSTaskCreate() 함수에 의해 태스크 생성시 호출되는 TCB 초기화 함수이다. TCB의 생성과 초기화를 수행한 후, 준비 큐로 태스크를 이동한다. 함수의 인자 중에서 *pdata는 OSTaskCreate() 함수로부터 전달되며, 이 값을 이용하여 라인 12~13은 태스크의 주기와 잔여 주기를 설정하고 라인 14~15는 태스크의 실행시간과 잔여실행시간을

설정한다.

```
1 : INT8U   OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos,
2 : INT16U id, INT32U stk_size, void *pext, INT16U opt, INT16U *pdata)
3 : {
4 :    OS_TCB     *ptcb;
5 :    ptcb = OSTCBFreeList; // Get a free TCB
6 :                 :
7 :    if (ptcb != (OS_TCB *)0) {
8 :        // Load Stack pointer in TCB
9 :        ptcb->OSTCBStkPtr     = ptos;
10:                 :
11:       // 태스크의 주기와 실행시간 설정 및 잔여 주기와 실행시간계산
12:       ptcb->TPeriod = pdata[0];
13:       ptcb->RemainTPeriod = pdata[0];
14:       ptcb->ExecTime = pdata[1];
15:       ptcb->RemainExecTime = pdata[1];
16:                 :
17:    return (OS_NO_MORE_TCB); }
```

<그림 5.23> TCB 초기화

태스크의 잔여 주기와 잔여 실행 시간을 계산하기 위해서 지속적인 타임 추적이 필요하다. 이를 위해서 타이머 인터럽트가 발생할 때마다 실행되는 OSTimeTickHook()함수를 이용하였다. 이 함수는 기존 스케줄러의 내용 수정을 최소화할 수 있도록 사용자가 정의해서 사용할 수 있도록 uC/OS-II에서 제공하는 함수이며, OSTimeTick() 함수의 내부에 있다(<그림 5.24>). 라인 4~6은 실행 준비 태스크들 중에서 가장 높은 우선순위를 가지는 태스크를 선정하여 잔여 실행시

간을 계산한다.

```
1 : void  OSTimeTickHook (void)
2 : { OS_TCB *ptcb;
3 :   INT8U x, y, HighestPriority;
4 :   y = OSUnMapTbl[OSRdyGrp];//Finding the highest priority task
5 :   x = OSUnMapTbl[OSRdyTbl[y]];
6 :   HighestPriority = (y << 3) + x;
7 :   ptcb = OSTCBList;
8 :   if (OSTCBPrioTbl[HighestPriority]->RemainExecTime > 0)
9 :     OSTCBPrioTbl[HighestPriority]->RemainExecTime--;
10:   while (ptcb->OSTCBPrio != OS_IDLE_PRIO)
11:   { OS_ENTER_CRITICAL();
12:     ptcb->RemainTPeriod--;
13:     if (ptcb->RemainTPeriod <= 0)
14:     { ptcb->RemainTPeriod = ptcb->TPeriod;
15:       ptcb->RemainExecTime = ptcb->ExecTime;  }
16:     ptcb = ptcb->OSTCBNext;
17:     OS_EXIT_CRITICAL();  }
18: }
```

<그림 5.24> 태스크 주기 및 마감시한 계산

라인 13~15는 잔여 주기가 0이 되었을 때의 태스크 주기와 실행시간을 재설정하는 루틴이다. 잔여 주기가 0이면 마감시한이 종료하였음을 의미하며 태스크의 잔여 주기와 잔여 실행시간을 태스크의 주기인 TPeriod와 실행시간인 ExecTime로 재설정된다.

<그림 5.25>는 사용자 정의 태스크로서, 라인 12~13은 태스크의

실행이 완료되면 잔여 주기(RemainTPeriod) 시간 동안 스스로를 대기상태(suspend)로 전환한다. 이를 위해 OSTimeDly(timetick) 함수를 사용하였다.

```
1 : void PTask1 (void *pdata)
2 : { char    s[80];
3 :   sprintf(s, "%s  %5d  %5d  %5u  %5u", "Periodic Task 9 ",
4 :         OSTCBPrioTbl[9]->TPeriod,
5 :         OSTCBPrioTbl[9]->RemainTPeriod,
6 :         OSTCBPrioTbl[9]->ExecTime,
7 :         OSTCBPrioTbl[9]->RemainExecTime);
8 :   PC_DispStr(3,8,s, DISP_FGND_BLACK
9 :             + DISP_BGND_LIGHT_GRAY);
10:   for (; ; )
11:   {
12:     if (OSTCBPrioTbl[9]->RemainExecTime <= 0)
13:       OSTimeDly(OSTCBPrioTbl[9]->RemainTPeriod);
14:   } }
```

<그림 5.25> 주기적 태스크 예

<그림 5.26>은 PTS 알고리즘을 구현한 것이다. 라인 4에서 비주기적 태스크의 요청을 판단한 후, 라인 5에서 비주기적 태스크가 추가적으로 실행해야할 요청이 있는지를 판단한다. 추가 요청이 있다면 라인 6에서 잉여시간 크기를 획득한다. 라인 7에서 비주기적 태스크가 실행 중임을 알리기 위해서 변수 APT_Can_Run에 TRUE 값을 설정한 후, 비주기적 태스크를 실행하기 위해서 ATPS를 호출한다.

PTS는 비주기적 태스크 뿐만 아니라 결함이 발생한 태스크의 복

구를 위한 부분에도 관여한다. 결함이 발생한 태스크는 현재 수행 중에 태스크에서 결함이 발생하며 만약 현재 비주기적 태스크가 수행하고 있는 중이라면 이는 결함이 발생하지 않음을 의미한다. 그러므로 라인 11에서 비주기적 태스크가 실행중인지를 판단한 후, 실행하지 않는 상태라면 라인 12에서 결함 요청이 발생하였을 경우, FTM을 호출하여 결함이 발생한 태스크의 복구를 수행한다.

```
1 :  void   PTS (void)
2 :  {
3 :                :
4 :    if(APT_Create == TRUE) {
5 :     if(APT_Request == TRUE) {
6 :       if(Surplus_TimeTable[Time_Index]>0){
7 :         APT_Can_Run = TRUE;
8 :         ATPS();
9 :    }}}
10:              :
11:    if(!APT_Can_Run) {
12:     if(Fault_occur == TRUE) {
13:       FTM();
14:    }}
```

<그림 5.26> PTS(Periodic Task Scheduler) 구현

<그림 5.27>은 5.1절에서 제시된 10개의 주기적 태스크들을 수행시킨 결과이다. 태스크 집합이 수행되고 있는 임의의 시점에 대한 결과이다. 그림에는 각 태스크의 주기, 마감시간까지 남은 주기, 실행시간, 실행완료를 위해 남은 실행시간을 확인 할 수 있다.

<그림 5.27> 주기적 태스크 실행 결과

5.5 FTM(Fault-Tolerant Manager)

본 절에서는 FTM을 구현하고 그 결과를 고찰한다. 결함 허용을 위한 자료 집합은 <그림 5.28>과 같다. 라인 변수 Fault_Generateflag는 결함 발생시 하이퍼 주기 내에서 단일 결함을 발생시키기 위해 사용되는 변수이다. 변수 Fault_occur는 태스크의 결함 발생을 알리는 변수이며 변수 Fault_Task_ID는 결함이 발생한 태스크의 우선순위 값으로써 결함이 발생한 태스크를 식별하는데 사용된다. FT_Complete는 결함 복구가 완료되었을 때 TRUE 값을 가지며 결함이 복구되어 완료되었는지를 판단하는데 사용된다. 마지막으로 배열

변수 BlockedHPTask는 결함이 발생한 태스크가 결함 복구를 수행하는 중에 우선순위가 복구 태스크보다 크면서 마감시간이 큰 태스크의 경우 블록상태로 전환하기 위해서 사용한다.

```
1 :  BOOLEAN Fault_Generateflag = TRUE;
2 :  BOOLEAN Fault_occur = FALSE;
3 :  INT8U Fault_Task_ID;
4 :  BOOLEAN FT_Complete = TRUE;
5 :  INT8U BlockedHPTask[64]={0};
```

<그림 5.28> FTM 자료구조

<그림 5.29>와 같이 시뮬레이션을 위해 우선순위가 6인 태스크에서 임의로 태스크의 결함을 발생시켰다.

```
1 :  void PTask2 (void *pdata)
2 :  {
3 :                  :
4 :   for ( ; ; ) {
5 :     if (!(Time_Index % HYPER_PERIOD))
6 :       Fault_Generateflag = TRUE;
7 :       if((Time_Index > 100)
8 :           && (Fault_Generateflag == TRUE)) {
9 :         OSTimeDly(5);
10:         Fault_Generateflag = FALSE
11:         Fault_occur = TRUE;
12:         Fault_Task_ID = 6; }
13:                  :
14:  }}
```

<그림 5.29> 결함 발생 태스크 예

본 논문의 결함 허용은 단일 결함에 대한 복구를 지원한다. 그러

므로 라인 5~6에 결함이 1회 발생하도록 하였으며 라인 7~8에서 임의의 시간(100)에 결함이 발생하도록 하였다. 또한 라인 9에서 임의의 시간 5만큼 시간이 지난 후에 결함이 발생하도록 하였으며 결함이 발생한 태스크와 결함 발생한 것을 알리기 위해서 각각 Fault_Task_ID에는 6을 Fault_occur 변수에 TRUE 값을 설정하였다.

<그림 5.30>은 태스크의 결함이 발생하였다고 판단될 경우 PTS에 의해 호출되는 FTM 알고리즘을 구현한 것이다.

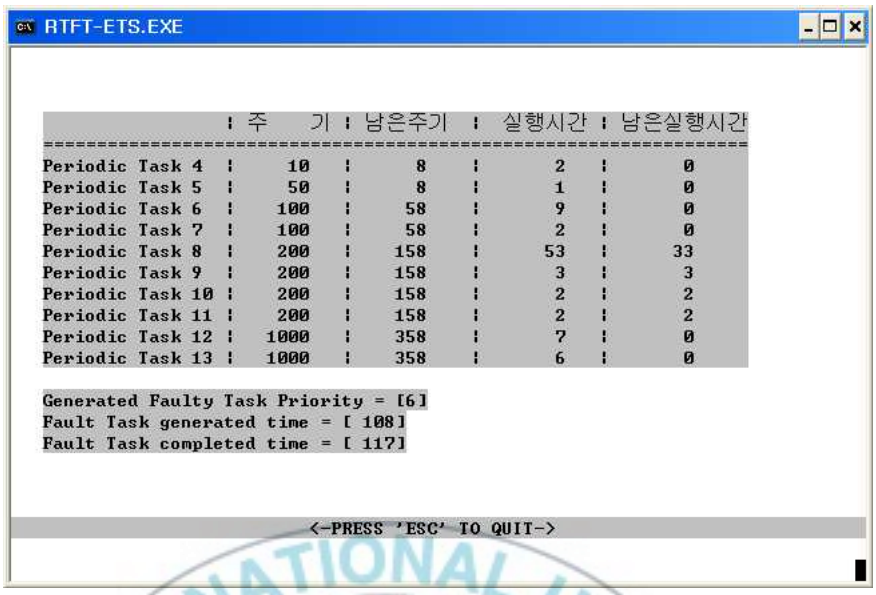
```
1 :    FTM()
2 :    { if (FT_Complete == TRUE) {
3 :        OSTCBPrioTbl[Fault_Task_ID]->RemainExecTime
4 :          = OSTCBPrioTbl[Fault_Task_ID]->ExecTime; }
5 :      FT_Complete = FALSE;
6 :                          :
7 :      if((HighestPriority > Fault_Task_ID)
8 :        && (OSTCBPrioTbl[HighestPriority]->RemainTPeriod >
9 :              OSTCBPrioTbl[Fault_Task_ID]->RemainTPeriod)) {
10:       BlockedHPTask[i] = HighestPriority;
11:       OSTaskSuspend(HighestPriority);
12:       HighestPriority = Fault_Task_ID;
13:       i++; } }
14:                          :
15:    if(OSTCBPrioTbl[Fault_Task_ID]->RemainExecTime <= 0) {
16:     FT_Complete = TRUE; }
17:                          :
18:    if(FT_Complete == TRUE) {
19:     Fault_occur = FALSE;
20:     for(i = 0; i < BlockedCount; i++) {
21:      OSTaskResume(BlockedHPTask[i]); }
22:    }}
```

<그림 5.30> FTM 구현

라인 2에서 결함이 발생한 후, 복구가 완료되지 않았는지를 판단한다. 복구가 완료되지 않았다면, 라인 3~4에서 결함이 발생한 태스크의 실행시간을 초기화하고 현재 복구 모드에서 수행중임을 알리기 위해 라인 5에서 FT_Complete 값을 FALSE로 세팅한다.

라인 7~13은 결함을 복구하는 도중에 우선순위가 높은 태스크를 처리하는 부분이다. 우선순위가 높고 마감시간이 긴 태스크를 블록 상태로 전환함으로써 결함이 발생한 태스크의 실행을 지속시킨다. 이는 결함이 발생한 태스크의 마감시간을 보장하기 위해서다. 라인 15~16은 결함 태스크의 복구 완료시간을 판단하여 복구 완료를 알리는 변수 FT_Complete를 TRUE로 설정한다. 라인 18~21은 결함 복구가 완료되었을 때 처리하는 부분이다. 라인 19에서 결함 복구가 완료되었는지를 판단한 후, 복구 완료로 판단되면 현재 결함이 발생한 태스크가 없다는 것을 스케줄러에게 알리기 위해 변수 Fault_occur를 FALSE로 설정하고 라인 12에서 블록 상태에 있던 태스크들을 준비 상태로 전환시킨 후, RMS에 의한 스케줄링을 수행한다.

<그림 5.31>은 결함이 발생한 태스크의 복구 결과를 나타낸다. 결함이 발생한 태스크의 우선순위는 6이며 결함 발생 시간은 108이 된다. 이 시간은 임의로 결함을 발생시킨 시간인 100, 태스크 6보다 우선순위가 높은 태스크 4와 5의 실행 시간의 합인 3, OSTimeDly(5)에 의한 지연 시간 5의 합이다. 또한 결함 태스크의 복구 시간은 결함 발생 시간에 태스크 자신의 실행 시간인 9를 덧셈한 117이 된다.



<그림 5.31> 결함 태스크 복구 결과

5.6 APTS(Aperiodic Task Scheduler)

본 절에서는 비주기적 태스크를 스케줄링하는 비주기적 태스크 스케줄러(APTS: Aperiodic Task Scheduler)의 구현에 대해 기술한다. 비주기적 태스크 관리하기 위해서는 추가적인 자료구조와 실행 함수가 필요하다. 자료구조는 OSApTCBPrioTbl[], OSApTaskCtr, OSApRdyGrp, OSApRdyTbl[]을 추가하였으며, 함수는 OSApTaskCreate(), OS_ApTCBInit(), OSApTaskDel()을 추가하였다.

<그림 5.32>는 u/COS-II에 제시한 태스크 관리 자료구조를 응용하여 추가한 자료구조들이다.

```
1 : OS_EXT   INT8U     OSApRdyGrp;
2 : OS_EXT   INT8U     OSApRdyTbl[OS_RDY_TBL_SIZE];
3 : OS_EXT   INT8U     OSApTaskCtr;
4 : OS_EXT   OS_TCB    *OSApTCBPrioTbl[OS_LOWEST_PRIO + 1];
```

<그림 5.32> 비주기적 태스크 관리를 위한 자료구조

각 자료구조의 용도는 다음과 같다. 태스크 우선순위 테이블을 위한 OSApRdyGrp와 배열 OSApRdyTbl[], 현재 생성된 비주기적 태스크의 수를 저장하는 변수 OSApTaskCtr, 비주기적 태스크의 정보를 저장할 수 있는 배열 OSApTCBPrioTbl[]이다.

```
1 :  INT8U OSApTaskCreate(void (*task)(void *pd), void *pdata,
2 :                                  OS_STK *ptos, INT8U prio)
3 :  {
4 :                      :
5 :      psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, 0);
6 :      err=OS_ApTCBInit(prio,psp,(OS_STK *)0,0,0,(void *)0,0,
7 :                                  (INT32U *)pdata);
8 :      if (err == OS_NO_ERR) {
9 :        OSApTaskCtr++;
10:        OS_EXIT_CRITICAL();
11:        if (OSRunning == TRUE) {
12:          OS_Sched();
13:                      :
14:    return (OS_PRIO_EXIST);
15:  }
```

<그림 5.33> OSApTaskCreate() 함수

<그림 5.33>은 비주기적 태스크를 생성하는 함수를 구현한 것이

다. 라인 5에서 비주기적 태스크가 사용할 스택을 초기화하고 라인 6에서 OS_ApTCBInit() 함수를 이용하여 TCB(Task Control Block)를 초기화한다. 라인 8에서 정상적으로 TCB 초기화가 진행되면, 라인 9에서 비주기적 태스크 수를 1증가시킨다. 라인 11~12에서 실행중인 태스크가 존재할 경우 PTS를 호출하여 실행할 태스크를 스케줄링한다.

<그림 5.34>는 OS_ApTaskCreate() 함수에 의해 호출되는 비주기적 태스크의 TCB를 초기화하는 함수로서 주기적 태스크를 생성할 때 이용된 OS_TCBInit() 함수와 동일한 구조를 가진다.

```
1 : INT8U  OS_ApTCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos,
2 : INT16U id, INT32U stk_size, void *pext, INT16U opt, INT32U *pdata)
3 : {
4 :     OS_TCB      *ptcb;
5 :     ptcb = OSTCBFreeList;
6 :                 :
7 :     ptcb->TPeriod = pdata[0];
8 :     ptcb->RemainTPeriod = pdata[0];
9 :     ptcb->ExecTime = pdata[1];
10:     ptcb->RemainExecTime = pdata[1];
11:                 :
12:     OSApTCBPrioTbl[prio] = ptcb;
13:                 :
14:     OSTCBList                   = ptcb;
15:     OSApRdyGrp                  |= ptcb->OSTCBBitY;
16:     OSApRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
17:                 :
18: }
```

<그림 5.34> 비주기적 태스크 TCB 초기화

그러나 주기적 태스크 생성 단계와의 차이점은 비주기적 태스크는 생성된 TCB를 비주기적 태스크 처리를 위한 우선순위 테이블인 OSApTCBPrioTbl[prio]에 저장하며 태스크를 준비 상태로 만들기 위해서 OSApRdyGrp 변수와 OSApRdyTbl[]을 사용한다는 점이다.

<그림 5.35>는 비주기적 태스크의 실행이 완료되었을 경우, 비주기적 태스크를 삭제하기 위한 함수이다.

```
1 :  INT8U   OSApTaskDel (INT8U prio)
2 :  {              :
3 :    ptcb = OSApTCBPrioTbl[prio];
4 :     if (ptcb != (OS_TCB *)0) {
5 :       y = ptcb->OSTCBY;
6 :       OSApRdyTbl[y] &= ~ptcb->OSTCBBitX;
7 :       if (OSApRdyTbl[y] == 0x00) {
8 :          OSApRdyGrp &= ~ptcb->OSTCBBitY; }
9 :       ptcb->OSTCBDly   = 0;
10:       ptcb->OSTCBStat = OS_STAT_RDY;
11:       OSApTaskCtr--;
12:       OSApTCBPrioTbl[prio] = (OS_TCB *)0;
13:                 :
14:       ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
15:       OSTCBList = ptcb->OSTCBNext;
16:                 :
17:       ptcb->OSTCBNext    = OSTCBFreeList;
18:       OSTCBFreeList      = ptcb;
19:       ptcb->OSTCBStkPtr = (OS_STK *)0;
20:       OS_Sched();
21:       return (OS_TASK_DEL_ERR);
22:  }
```

<그림 5.35> 비주기적 태스크 삭제 함수

먼저 라인 5~8에서 비주기적 태스크 준비 리스트로부터 태스크를 삭제한 후, 라인 11에서 비주기적 태스크 수를 1감소시킨다. 그리고 라인 12에서 비주기적 태스크 준비 큐에서 비주기적 태스크의 TCB를 삭제하고, 라인 17에서 실행이 완료된 태스크에 대한 TCB를 OSTCBFreeList로 반환한다. 마지막으로 태스크를 삭제한 후에 라인 20에서 스케줄러를 호출하여 실행할 태스크를 스케줄링한다.

```
1 :  ATPS()
2 :  {
3 :   if(SurplusSize >=
4 :         OSApTCBPrioTbl[HighestPriority]->RemainExecTime) {
5 :        OSApTCBPrioTbl[HighestPriority]->TPeriod =
6 :        OSApTCBPrioTbl[HighestPriority]->RemainExecTime; }
7 :      else
8 :      { OSApTCBPrioTbl[HighestPriority]->TPeriod = SurplusSize; }}}
9 :   if (APT_Can_Run == TRUE) {
10:    y= OSUnMapTbl[OSApRdyGrp];
11:    APHighestPriority = (INT8U)((y<< 3)
12:                              +OSUnMapTbl[OSApRdyTbl[y]]);
13:    HighestPriority = APHighestPriority;
14:    if(OSApTCBPrioTbl[HighestPriority]->TPeriod <= 0) {
15:     if(OSApTCBPrioTbl[HighestPriority]->RemainExecTime <= 0) {
16:           :
17:      OSApTaskDel(AT_Priority);}}}
18:  }
```

<그림 5.36> ATS(Aperiodic Task Scheduler) 구현

<그림 5.36>은 비주기적 태스크 스케줄러인 APTS를 구현한 함수로서, 비주기적 태스크의 실행요청이 발생하고 비주기적 태스크의

실행을 위한 잉여시간이 존재할 경우 PTS에 의해 호출된다. 라인 3~6은 현재 시점에서 비주기적 태스크가 수행해야할 실행시간을 결정한다. 할당 받은 잉여시간의 길이가 비주기적 태스크가 실행해야할 전체 실행 시간과 비교하여 크다면, 비주기적 태스크가 실행할 수 있는 충분한 잉여시간이 존재함을 의미하므로 비주기적 태스크의 전체 실행시간을 변수 TPeriod에 저장한다. 라인 8은 할당받은 잉여시간이 비주기적 태스크가 실행해야할 처리 시간보다 적을 경우 할당받은 잉여시간 만큼만 실행하도록 하기 위해서 할당받은 잉여시간인 SurplusSize을 TPeriod에 저장한다.

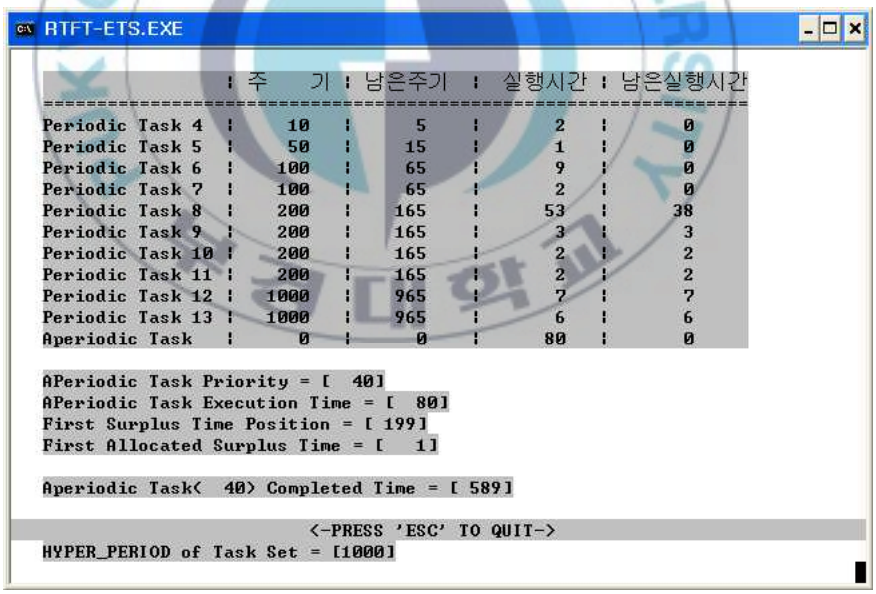
라인 9에서 비주기적 태스크가 실행할 준비가 되었다면 라인 10~12는 비주기적 태스크 준비 큐에서 우선순위가 가장 높은 태스크를 선정한 후, 라인 13에서 가장 높은 우선순위 태스크로 전환하여 비주기적 태스크를 실행하도록 한다. 라인 14~17은 비주기적 태스크의 전체 실행시간이 완료되었을 때, 비주기적 태스크의 완료를 설정하고 완료된 비주기적 태스크를 삭제한다.

<그림 5.37>은 비주기적 태스크를 생성하기 위한 주기적 태스크(Periodic Task 4)를 나타낸다. 비주기적 태스크의 특성상 임의의 시간에 생성되므로 시뮬레이션을 위해 라인 5에서 임의로 생성시간을 설정하였다. 또한, 라인 7~9에서 OSApTaskCreate() 함수를 이용하여 비주기적 태스크를 생성한다.

```
1 : void   PStartTask (void *pdata)
2 : {        :
3 :  for (; ; )
4 :  { //비주기적 태스크의 생성
5 :   if((Time_Index == 100) && !APT_Create)
6 :   {
7 :    OSApTaskCreate(AperiodicTask,
8 :        (INT32U *)&AperiodicTaskExec[0],
9 :        &AperiodicTaskStk[TASK_STK_SIZE - 1], AT_Priority);
10:    APT_Request = TRUE; }}
11:          :
12: }
```

<그림 5.37> 비주기적 태스크 생성

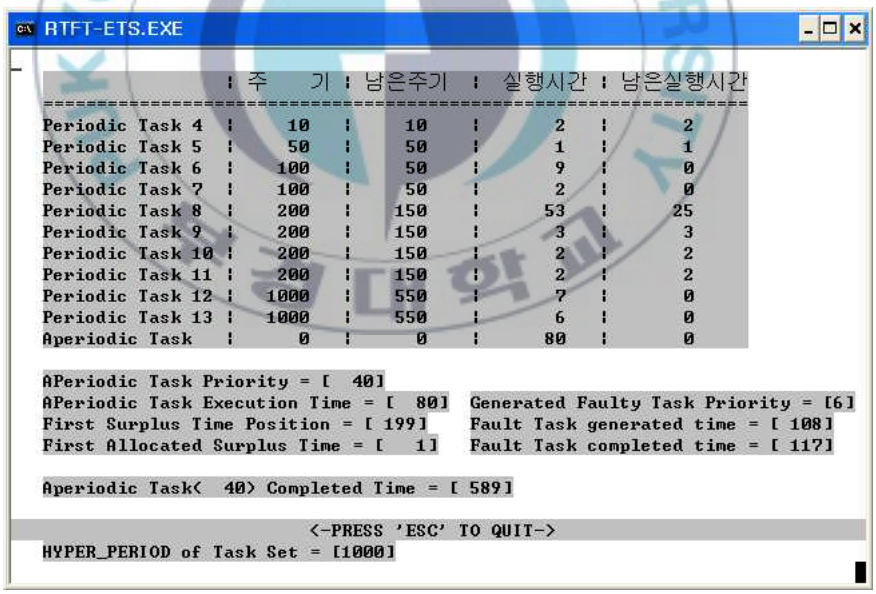
<그림 5.38>은 비주기적 태스크를 실행한 결과 화면이다.



<그림 5.38> 비주기적 태스크 실행 결과

시뮬레이션을 위한 주기적 태스크들은 <표 5.2>에서 제시한 태스크 집합을 사용하였으며, 비주기적 태스크는 임의로 우선순위는 40, 실행시간은 80으로 할당하였다. 비주기적 태스크가 생성된 후, 잉여시간 매니저로부터 할당받은 최초의 처리기 잉여시간의 위치는 199이며 그 크기는 1이다. Hyperperiod는 1000이며, 비주기적 태스크는 시간 t=589일 때 비주기적 태스크의 실행 시간인 80을 모두 할당받게 되며 실행을 완료한다.

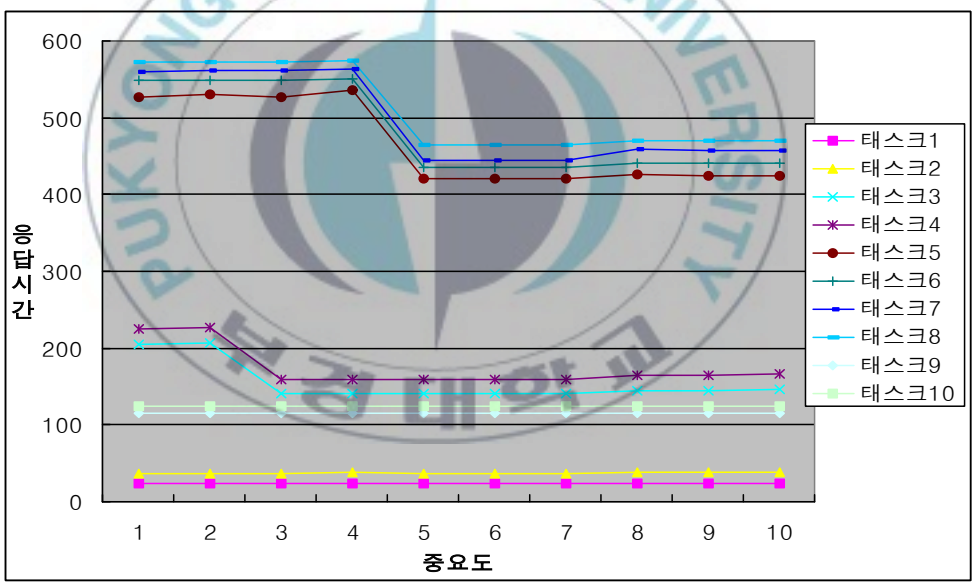
<그림 5.39>는 비주기적 태스크의 실행과 결함 복구를 동시에 처리하는 결과 화면이다. 비주기적 태스크가 생성되어 스케줄러에 실행 요청 후 실행이 완료되고 주기적 태스크에서 결함이 발생하여 결함을 복구하는 결과 화면이다.



<그림 5.39> 비주기적 태스크 및 결함 복구 완료

비주기적 태스크의 실행 시간은 80이며 최초로 할당 받은 잉여시간의 크기는 시간 199에서 1이 된다. 그리고 비주기적 태스크의 실행이 완료된 시간은 589가 된다. 결함 허용의 경우 결함이 발생한 태스크의 우선순위는 6이며 결함이 발생한 시간은 시간 5동안 지연을 시킨 후에 발생하도록 하였으므로 108이 된다. 결함이 발생한 시간인 시간 108에서 결함 태스크의 복구 작업이 실행되며 우선순위 6인 태스크의 실행 시간이 9이므로 복구가 완료되는 시간은 117이 된다.

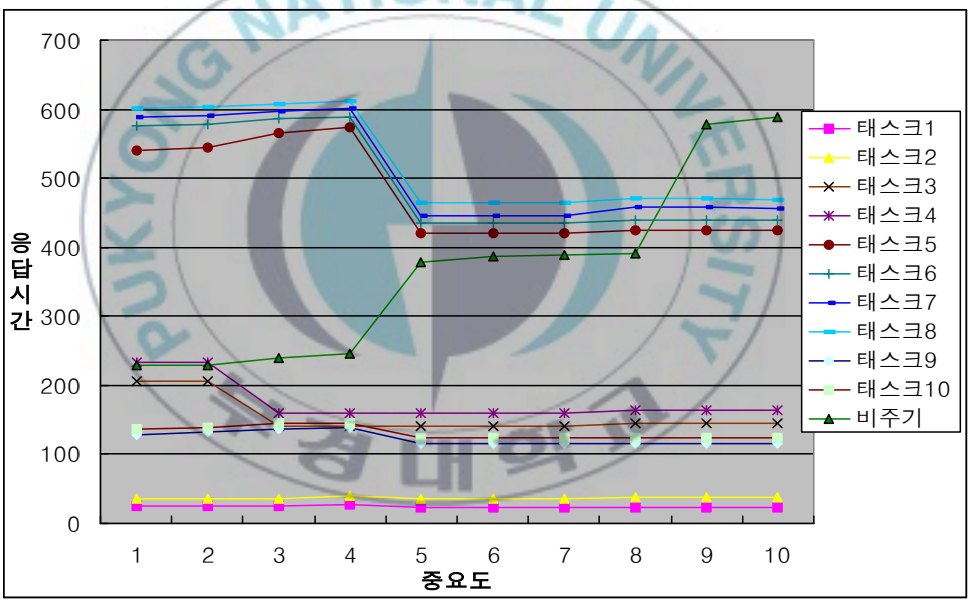
<그림 5.40>은 주기적 태스크 집합의 중요도에 따른 응답 시간을 나타낸다.



<그림 5.40> 주기적 태스크의 중요도에 따른 응답 시간

시뮬레이션을 위해 최대 잉여시간을 비주기적 태스크에 할당할 필요가 있으므로 비주기적 태스크의 실행 시간을 400으로 설정하였으며 중요도를 1~10으로 증가시켜 가면서 응답시간을 측정하였다. 주기적 태스크의 중요도가 증가할수록 태스크의 응답시간이 감소함을 보였다.

<그림 5.41>은 주기적 태스크 집합의 중요도에 따른 비주기적 태스크의 응답시간을 시뮬레이션한 결과이다. 비주기적 태스크의 실행 시간은 80으로 설정하였다.



<그림 5.41> 중요도에 따른 주기적/비주기적 태스크의 응답시간

주기적 태스크 집합의 중요도가 증가할수록 주기적 태스크의 응

답시간이 감소하며 이로 인해 비주기적 태스크의 응답시간이 증가함을 보였다. 이는 태스크를 스케줄링할 경우, 주기적 태스크의 중요도를 고려함으로써 비주기적 태스크의 응답시간을 조절할 수 있음을 나타낸다. 즉, 주기적 태스크의 중요도를 증가시킴으로써 주기적 태스크의 빠른 응답시간을 제공할 수 있으며 주기적 태스크의 중요도를 감소시킴으로써 비주기적 태스크의 응답시간을 감소시킬 수 있다. 이는 주기적 태스크에 중요도를 적용함으로써 주기적 태스크와 비주기적 태스크의 응답시간을 조절할 수 있음을 확인하였다.

6. 결론

최근 임베디드 시스템에서 사용되는 하드웨어의 사양이 향상되고 있으며 이를 이용하는 사용자들의 기능에 대한 요구 또한 다양하게 증가하고 있다. 그러므로 이러한 변화하는 환경에 적용할 수 있는 임베디드 시스템의 운영체제는 태스크를 실행함에 있어서 실시간적인 응답을 제공할 수 있어야 한다. 또한 임베디드 운영체제가 적용되는 시스템 중에는 수행 중에 발생하는 태스크의 결함이 전체 시스템에 치명적인 결과를 초래할 수 있으며 사람의 적극적인 개입이 어려운 경우가 있다. 따라서 이러한 임베디드 실시간 시스템을 위한 운영체제는 작업 결함으로 인한 시스템 고장 혹은 시스템 재시작을 방지하는 기능을 포함하여야 한다. 그러므로 실시간 임베디드 시스템은 운영체제 레벨에서 실시간 태스크를 관리하는 기법과 결함을 허용할 수 있는 기법을 제공할 필요가 있다.

본 논문에서는 임베디드 시스템에서 실시간성과 결함 허용을 제공하는 태스크 스케줄러를 설계하고 구현하였다. 실시간성을 제공하기 위해서 스케줄러에 주기적 태스크들을 실행하기 위한 RMS 알고리즘을 적용하였으며 비주기적 태스크들을 실행하기 위한 잉여시간을 제공하는 방법을 제시하였다. 또한 일시적인 결함이 발생한 태스크의 재실행을 통해 결함을 허용하는 결함 복구 기법도 제시하였다.

제안한 스케줄러는 태스크 실행 가능성 검사 관리자(ECM : Executability Check Manager), 백업시간 관리자(BTM : Backup

Time Manager), 잉여시간 관리자(STM : Surplus Time Manager), 주기적 태스크 스케줄러(PTS : Periodic Task Scheduler), 결함 허용 관리자(FTM : Fault Tolerant Manager), 비주기적 태스크 스케줄러 (APTS: APeriodic Task Manager)로 구성되어 있다.

실행 가능성 검사 관리자는 주기적 태스크 집합을 스케줄링하기 전에 주기적 태스크들의 실행 가능성을 검사함으로써 잘못 설계된 태스크 집합의 실행 및 재설계로 인한 오버헤드를 줄일 수 있다. 백업 시간 관리자는 실행 중인 태스크에서 결함이 발생할 경우 태스크를 재실행하여 결함을 복구하기 위한 백업시간을 계산하고 관리한다. 잉여시간 관리자는 비주기적 태스크가 사용할 처리기 여유시간을 관리하는 구성요소로서 주기적 태스크들의 실행을 위한 처리기의 시간을 제외한 유휴시간을 비주기적 태스크가 사용할 수 있도록 처리기 유휴시간을 계산하고 관리한다.

주기적 태스크 스케줄러는 주기적 태스크들의 스케줄링을 담당한다. 그리고 비주기적 태스크의 요청이 발생할 경우 비주기적 태스크 스케줄러를 호출하여 비주기적 태스크를 실행하도록 하며 태스크의 결함이 발생할 경우 결함 허용 관리자의 결함 복구 작업을 지원한다.

결함 허용 관리자는 태스크에서 일시적인 결함이 발생할 경우 백업시간 관리자에게 백업시간을 요청하고 할당받은 백업 시간을 이용하여 결함이 발생한 태스크를 복구한다. 본 논문에서 제공되는 결함 허용 기법은 주기적 태스크들뿐만 아니라 결함이 발생한 태스크의 마감시간도 보장한다.

비주기적 태스크 스케줄러는 비주기적 태스크의 요청이 발생할 경우 잉여시간 관리자로부터 비주기적 태스크가 사용할 수 있는 처리기의 잉여시간을 할당받은 후 비주기적 태스크를 실행한다.

구현된 스케줄러를 시뮬레이션한 결과 주기적 태스크들의 실행 마감시간과 비주기적 태스크의 실행 및 완료를 보장하였다. 또한 비주기적 태스크의 빠른 응답시간을 제공할 수 있으며 주기적 태스크에 중요도를 적용하여 빠른 응답시간이 필요한 주기적 태스크의 응답시간을 줄일 수 있다. 즉, 주기적 태스크에 중요도를 적용함으로써 주기적 태스크와 비주기적 태스크의 응답시간을 조절할 수 있음을 확인하였다. 그러므로 실시간 임베디드 시스템에서 태스크집합의 스케줄링 정책을 수립할 때 예측하여야 할 태스크들의 시간응답성을 주기적 태스크의 중요도를 이용하여 결정할 수 있다. 또한, 제안한 결함 복구 기법은 태스크의 일시적인 결함 발생에도 정상적인 태스크들의 마감시간을 보장하며 또한 결함이 발생한 태스크의 마감시간도 만족하는 것을 확인할 수 있었다.


PUKYONG NATIONAL UNIVERSITY
부경대학교

<참 고 문 헌>

[1] J. T. Baldwin, *Predicting and Estimating Real-Time Performance*. Embedded Systems Programming, 8(2), Feb. 1995.

[2] L. Doyle and J. Elzey, "Successful Use of Rate Monotonic Theory on a Formidable Real Time System", *In 11th IEEE Workshop on Real-Time Operating Systems and Software*, pp.74-78. IEEE, May 1994.

[3] H. Kopetz, "Automotive Electronics-Present State and Future Prospects", *In FTCS* 25, 1995.

[4] K. W. Tindell, Fixed Priority Scheduling of Hard Real-Time Systems, PhD thesis, Univ of York, UK, 1994.

[5] H. Zou, F. Jahanian, "A Real-Time Primary- Backup Replication Service", *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, No. 6, 1999.

[6] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior", *RTSS* pp.166-171, 1989.

[7] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment",

JACM, 20(1), pp.46-61, 1973.

[8] WindRiver, "VxWorks 5.4", http://www.windrivers.com

[9] Accelerated Technology, "Nucleus real-time, multitasking kernel", http://www.acceleratedtechnology.co.kr/nucleus.html

[10] Micrium, "uC/OS-II, The Real-Time Kernel", http://www. uco ii.com.

[11] QNX software systems Ltd., QNX System Architecture User's Guide, http://www.qnx.com, 1999

[12] FSML, "RTLinux", http://www.fsmlabs.com/.

[13] Douglas Boling, "Updated with New Kernel Features, Windows CE 3.0 Really Packs a Punch", http://www.msdn.microsoft.com, July, 1999

[14] Velos Home-page, http://www.velos.co.kr/

[15] Integrated Systems, "pSOS System Concepts", http://www.isi.com, 1996.

[16] Be Inc., "The Media OS", technical white paper, http://www.be.com/products/freebeos

[17] RedHat, "eCOS", http://sources.redhat.com/ecos/

[18] J. J. Labrosse, *Micro C/OS-II Second Edition*, CMP Books, 2002.

[19] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System", *The BCS Computer Journal*, vol. 29,

no. 5, pp.390-395, 1986.

[20] N. C. Audsley, A. Burns, M. Rihardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach", *In Proc. of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pp.133-137, 1991.

[21] K. W. Tindell, A. Burns, and A. J. Wellings, "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks", *Real-Time Systems*, vol. 6, no. 2, pp.133-151, 1994.

[22] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems", *Proc. 13th Real-Time Systems Symp.*, pp.110-123, 1992.

[23] T. S. Tia, "Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems", *Technical Report* No.UIUCDCS-R-95-1906, University of Illinois

[24] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, *"A categorization of real-time multiprocessor scheduling problems and algorithms"*, In Joseph Y. Leung, editor, Handbook on Scheduling Algorithms, Methods, and Models, pp.30.1-30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004

[25] Y. Oh and S. Son, "Enhancing fault-tolerance in rate-

monotonic scheduling", *Real-Time Systems*, pp.315-330, 1994.

[26] S. Ghosh, R. Melhem, D. Mossé, and J. Sensarma, "Fault-tolerant rate-monotonic scheduling", *Real-Time Systems*, vol. 15, no. 2, pp.149-181, 1998.

[27] M. Pandya and M. Malek, "Minimum achievalble utilzation for fault-tolerant processing of periodic tasks", *IEEE Trans. on Comput.*, vol. 47, pp.1102-1113, 1998.

[28] S. Gosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems", *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 3, pp.272-284, 1997.

[29] Y. Chen, G. Xiong, "Imprecise computation fault -tolerant rate-monotonic scheduling", *ICA3PP'02, Algorithms and Architectures for Parallel Processing*, 2002.

[30] Y. S. Hong and H. W. Goo, "A fault-tolerant technique for scheduling periodic tasks in real-time systems", *Proc. of the Second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2004.

[31] S. Ghosh, D. Mosse, and R. Melhem, “Implementation and Analysis of a Fault-Tolerant Scheduling Algorithm”, *IEEE Transactions on Parallel and Distributed Systems*, Mar 1997.

[32] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems", *Proc. 13th Real-Time Systems Symp.*, pp.110-123, 1992.

[33] A. Burns, A. J. Wellings, C. M. Bailey, and E. Fyfe, "The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation", *Technical Report YCS-1993-190*, Department of Computer Science, University of York, 1993.

[34] 노학종, 김강진, 김문회, "임베디드 OS 기술 동향", *한국정보처리학회지*, vol. 11, no. 6, pp.42-55, 2004.

[35] 김선자, 김홍남, 김채규, "인터넷 정보가전용 RTOS 기술현황", *정보과학회지*, 제19권 제4호, pp.57-64, 2001.

[36] 노학종, 김강진, 김문회, "임베디드 OS 기술 동향", *한국정보처리학회지*, vol. 11, no. 6, pp.42-55, 2004.

[37] 박현선, 김인국, "MicroC/OS-II를 위한 실시간 스케줄링 구현", 단국대학교 전자계산학과 컴퓨터 과학전공 석사 졸업논문, 2002.

[38] 홍성수, "실시간 운영체제(Velos) 개발", *정보처리학회지*, 제9권 제1호, pp.95-102, 2002.

[39] 정경훈, 김병훈, 이동건, 김창수, 탁성우, "확장성 및 실시간성을 고려한 실시간 센서 노드 플랫폼의 설계 및 구현", *한국통신학회 논문지* 제32권 제8호, pp.509-520, 2007.

[40] 김병훈, 정경훈, 탁성우, “주기 및 비주기 태스크의 효율적인 관리를 위한 실시간 센서 노드 플랫폼의 설계”, *정보처리학회지* 제14-C권 제4호 통권 제114호, pp.371-382, 2007

[41] 김희헌, 박학봉, 박문주, 박민규, 조유근, 조성재, “잉여 여유시간을 이용한 연성 비주기 태스크들의 효율적인 스케줄링”, *정보과학회논문지*, 시스템 및 이론 제36권 제1호, 2009.2

[42] 김형일, 이승룡, 이종원, 김정순, “혼합 우선순위 시스템에서 경성 비주기적 태스크 스케쥴링 알고리즘”, *정보과학회논문지*(A) 제22권 제10호, pp.1445-1455, 1995

[43] 김진석, 김용석, “경성 비주기적 태스크들을 위한 온라인 스케줄링 알고리즘”, *정보통신논문지*, 제 5집, 2001