다양한 실시간 스케줄링 알고리즘들을 지원하기 위한 재구성 가능한 스케줄러 모델

(A Reconfigurable Scheduler Model for Supporting Various Real-Time Scheduling Algorithms)

심 재 홍 † 송 재 신 †† 최 경 희 ††† 박 승 규 ††† 정 기 현 ††††

(Jae-Hong Shim) (Jae-Shin Song) (Kyung-Hee Choi) (Seung-Kyu Park) (Gi-Hyun Jung)

요 약 본 연구에서는 다양한 실시간 스케줄링 알고리즘들을 구현할 수 있는 재구성 가능한 스케줄러 모델을 제안한다. 제안 모델은 기본적인 작업(job) 디스패처(dispatcher)와 소프트웨어 타이머를 제공하는 하위 계층의 스케줄링 틀(*framework*)과 이를 기반으로 응용에 적합한 특정 스케줄링 알고리즘을 구현하는 상위 계층의 태스크 스케줄러로 구성된다. 시스템 개발자는 상하 구성 요소간 정보 교환을 위한 커널 내부 인터페이스만 준수한다면, 커널 하부 메커니즘과는 독립적으로 새로운 스케줄링 알고리즘을 구현할 수 있다. 한번 구현된 태스크 스케줄러는 향후 새로운 시스템 구축시 재사용 가능하다. 실시간 리눅스(Real-Time Linux) [5]에 제안된 스케줄링 틀을 구현한 후, 이를 기반으로 대표적인 실시간 스케줄링 알고리즘들을 시험적으로 구현하여 보았다. 이를 통해 다양한 스케줄링 알고리즘들을 하부의 복잡한 커널 메커니즘 수정 없이 독립적으로 개발할 수 있음을 확인하였다. 또한 실험을 통해 두 단계 분리된 구조를 가진 제안 모델의 스케줄링 오버헤드가 하나로 통합된 기존 일체형 스케줄러와 큰 차이가 없음을 확인할 수 있었다.

키워드 : 실시간 스케줄러, 실시간 시스템, 스케줄링 알고리즘, 커널 재구성

Abstract This paper proposes a *reconfigurable scheduler model* that can support various real-time scheduling algorithms. The proposed model consists of two hierarchical upper and lower components, *task scheduler* and *scheduling framework*, respectively. The scheduling framework provides a *job dispatcher* and *software timers*. The task scheduler implements an appropriate scheduling algorithm, which supports a specific real-time application, based on the scheduling framework. If system developers observe internal kernel interfaces to communicate between two hierarchical components, they can implement a new scheduling algorithm independent of complex low kernel mechanism. Once a task scheduler is developed, it can be reused in a new real-time system in future. In Real-Time Linux [5], we implemented the proposed scheduling framework and several representative real-time scheduling algorithms. Throughout these implementations, we confirmed that a new scheduling algorithm could be developed independently without updates of complex low kernel modules. In order to confirm efficiency of the proposed model, we measured the performance of representative task schedulers. The results showed that the scheduling overhead of proposed model, which has two separated components, is similar to that of a classic monolithic kernel scheduler.

Key words : real-time scheduler, real-time system, scheduling algorithm, kernel reconfiguration

† 비 회 원 : 조선대학교 인터넷소프트웨어공학부 교수
jhshim@cesys.ajou.ac.kr
†† 정 회 원 : 아주대학교 정보및컴퓨터공학부
jssong@keris.or.kr
††† 종신회원 : 아주대학교 정보통신전문대학원 교수
khchoi@madang.ajou.ac.kr
sparky@madang.ajou.ac.kr
†††† 비 회 원 : 아주대학교 전기전자공학부 교수
khchung@madang.ajou.ac.kr
논문접수 : 2001년 2월 19일
심사완료 : 2002년 2월 15일

1. 서 론

실시간 시스템은 각종 통신 장비, 사무 자동화 기기, 그리고 정보 가전 기기 등에 이르기까지 다양하고 광범위한 적용 분야를 지니고 있다. 폭 넓은 시장과 잠재적인 시장확대의 가능성, 그리고 제품의 소형화 및 반도체 기술 향상 등으로 인해 실시간 시스템에 대한 관심은 더욱 고조되고 있다[1]. 정보가전 제품의 생명 주기 단축과 이로 인한 제품 개발 기간의 단축은 시스템 개발자들로 하여금 기존 일체형 실시간 운영체제(RTOS)보다는 재구성이 가능한 RTOS를 더 선호하게 만들었다[2]. 이러한 맥락에서 RTOS는 응용 분야별로 널리 알려진 대표적인 알고리즘들을 거의 대부분 지원해야 할 필요가 있다. 이를 바탕으로 시스템 개발자들은 해당 응용에 가장 적합한 알고리즘을 선택하여 사용하거나, 기존 알고리즘을 바탕으로 새로운 알고리즘을 개발할 수가 있기 때문이다 [4].

실시간 시스템을 연구하는 대부분의 연구자들의 애로점은 해당 응용을 쉽게 구현하고 테스트 할 수 있는 범용 실시간 시스템이 없다는 것이다. 이는 실시간 시스템 특성상 대부분의 시스템이 응용의 특성에 맞게 특별히 설계/제작된 보드에 내장되기 때문이다. 비록 일부 상용 RTOS들이 범용 시스템 보드를 동시에 지원하기도 하지만, 소스가 공개되지 않아 일반 연구자들이 쉽게 접근하기 어려운 문제가 있다. 다행이 최근 들어 리눅스를 기반으로 하는 일부 실시간 시스템들이 발표되었다[5-7]. 그러나 이들 시스템은 기존 리눅스 기반에서 실시간 응용의 시간 제약을 만족할 수 있는 기반만을 제공할 뿐, 극히 제한된 실시간 서비스를 제공한다. 따라서 기존에 발표된 알고리즘을 바탕으로 새로운 알고리즘을 개발하고, 이의 성능을 비교 분석하기 위해선 기존 알고리즘도 함께 개발해야 하는 고충이 따른다. 뿐만 아니라, 연구자들은 자신이 연구하는 분야의 소스도 이해해야 하지만, 이것이 미치는 파급 효과로 인해 시스템 전체를 완전히 이해하고 복잡한 하위 단계 커널 모듈도 함께 수정해야 한다. 이는 곧 이해 부족으로 인한 예기치 못한 시스템 에러를 발생시킬 수 있으며, 또 다른 시스템 불안 요소를 생성시킨다.

따라서 본 연구에서는 폭 넓은 실시간 응용의 다양한 요구에 적합한 여러 실시간 스케줄링 알고리즘을 구현할 수 있는 재구성이 가능한 스케줄러 모델을 제안하고자 한다. 이를 바탕으로 시스템 설계자로 하여금 해당 응용에 맞는 최적의 알고리즘을 선택할 수 있게 하고, 응용의 요구 조건을 최적으로 지원하는 시스템으로 재구성할 수 있게 한다. 또한 스케줄링 알고리즘을 커널과 복잡하게 연결된 형태로 구현하는 것이 아니라, 하위 단계의 복잡한 커널 모듈을 수정하지 않고도 해당 알고리즘을 효율적으로 구현할 수 있는 실시간 스케줄링 틀(*framework*)을 제공하고자 한다. 이를 토대로 실시간 시스템 연구자들은 해당 응용을 효과적으로 지원하는 새로운 알고리즘을 개발하고 테스트할 수 있는 기반 환경을 확보할 수 있다.

본 논문의 구성은 다음과 같다. 2절에서 실시간 스케줄링 기법과 각 기법별 대표적인 스케줄링 알고리즘들을 비교 분석하고, 3절에서는 이들을 지원하면서 재구성이 가능한 스케줄러 모델을 제안한다. 4절에서는 시험적으로 구현된 대표적인 스케줄링 알고리즘들의 성능을 측정한 후, 이를 바탕으로 제안 모델의 유용성과 타당성을 검토한다. 5절에서 본 연구와 관련한 기존 연구를 살펴 보고, 마지막 6절에서 향후 연구 계획에 대해 논의한다.

2. 실시간 스케줄링 알고리즘

본 절에서는 제안하고자 하는 스케줄러 모델이 지원하는 실시간 스케줄링 기법과 각 기법별 대표적인 스케줄링 알고리즘들을 살펴보고, 이들의 특징에 대해 비교 분석해 보고자 한다.

실시간 스케줄링 기법은 크게 세가지 기법, 즉 우선순위 유도형, 시간 유도형, 공유 유도형 스케줄링 기법으로 분류될 수 있다[6, 8]. 우선순위 유도형(priority-driven) 스케줄링 기법은 각 태스크에게 하나의 우선순위를 부여하고, 특정 순간에 가장 높은 우선순위를 가진 태스크를 우선적으로 선택하여 실행한다. 이 기법은 태스크의 우선순위 부여 방법에 따라 정적 우선순위와 동적 우선순위 스케줄링 알고리즘으로 분류할 수 있다. 가장 잘 알려진 고정 우선순위 스케줄링 알고리즘은 Rate Monotonic(RM)[9]과 Deadline Monotonic(DM)[10]이다. 또한 대표적인 동적 우선순위 알고리즘은 Earliest Deadline First(EDF)[9]과 Minimum Laxity First (MLF)[11]알고리즘이다. 이들 알고리즘들은 태스크의 시간 제약(만기나 주기)을 우선적으로 고려하여 우선순위를 부여하고, 이를 기반으로 스케줄을 결정한다. 또한 소수의 태스크 속성(주기와 실행 시간)만으로 스케줄 가능성을 결정할 수 있고, 구현하기도 간단하다.

지속적이고 잘 알려진 입력 데이타 흐름을 가진 초고속 통신망 스위칭 시스템에서는 매우 높은 예측성을 가진 시간 유도형(time-driven) 스케줄링 기법이 사용된다[12, 13]. 시간 유도형 스케줄링에서는 어떤 태스크를

언제 실행할 것인지를 사전에 오프-라인으로 스케줄링 하여 이를 테이블 형태로 저장하고, 시스템 실행 시간에는 단지 주어진 테이블에 따라 프로세서를 할당한다. 이 범주에 속하는 대표적인 스케줄러가 Cyclic Executive 이다[14].

우선순위 유도형 스케줄링 알고리즘은 매우 효과적이긴 하지만, 실시간 태스크와 비실시간 태스크를 매우 엄격히 구분하며 항상 실시간 태스크들을 우선적으로 스케줄링 한다. 그러나 이러한 접근 방식이 적합하지 않은 경우도 있다. 예를 들어, 실시간 화상 회의 시스템은 아주 엄격한 실시간 성능 보장을 요하지 않는다. 이러한 응용에서는 비례 공유 자원(proportional share resource) 할당 알고리즘이 더욱 적합하다. 이런 스케줄링 기법을 공유 유도형(share-driven) 스케줄링 기법이라 한다. 공유 유도형 스케줄링은 각 태스크에 일정 시간동안 프로세서를 할당한다. 각 태스크에 할당되는 프로세서 시간은 해당 태스크에 주어진 가중치에 따라 비례적으로 달라진다. 이러한 스케줄링 기법의 대표적인 알고리즘이 Weighted Fair-Queuing(WFQ) 알고리즘[15]이며, 주로 초고속 통신망용 스위치를 위한 패킷 데이타 스케줄링에 활용된다. 반면 태스크 스케줄링 알고리즘으로 응용 가능한 Preemptive WFQ[8]는 기반 스케줄링 알고리즘으로 EDF을 사용하며, 모든 작업들은 여러 개의 서로 다른 가중치를 가진 Total Bandwidth Server(TBS)[16]들에 의해 처리된다.

3. 재구성이 가능한 실시간 스케줄러 모델

실시간 시스템에서 모든 실시간 태스크들은 주기, 최악의 경우 실행 시간, 상대 만기, 절대 만기, 도착 시간 등과 같은 시간 요구에 의해 정의된다. 태스크의 시간 요구를 해당 태스크의 태스크 속성이라 하며, 이는 그 태스크에 가해지는 시간 제약이기도 하다. 실시간 스케줄러는 모든 태스크의 속성들이 만족되도록 시스템 자원들을 할당할 책임이 있다. 그러나 응용에 의해 정의된 높은 단계의 태스크 속성은 일반적으로 스케줄러에 의해 직접적으로 이용되지 않는다. 즉, 태스크 속성들은 스케줄링 메커니즘(낮은 단계의 스케줄러)에 의해 처리될 수 있도록 낮은 단계의 속성들로 변환되어야 한다. 예를 들어, RM(또는 EDF) 스케줄링을 사용할 경우, 응용은 단지 높은 단계의 태스크 속성들(주기와 실행 시간, 시작 시간)만 지정한다. 따라서 낮은 단계의 스케줄러와 사용자 응용 사이에 가상적인 중간 단계 스케줄러(스케줄링 알고리즘을 구현한 모듈)가 존재하여, 높은 단계의 태스크 속성들을 낮은 단계의 작업 속성들(작업의 시작 시간, 절대 만기, 우선순위 등)로 변환하여야 한다.

대부분의 실시간 커널에서는 이러한 가상적인 중간 단계 스케줄러와 낮은 단계 스케줄러가 서로 밀접하게 결합되어 하나의 '커널 스케줄러'를 구성한다. 이들은 스케줄러 데이타 구조와 함수들을 공유하면서 복잡하게 얽히고 섞여 있다. 만약 스케줄러 내부에 잠재해 있는 이들 두개의 추상적 모듈들을 분명하게 분리할 수 있다면, 많은 이점을 얻을 수 있다. 그 중에서 가장 명확한 것은 낮은 단계 스케줄러를 수정하지 않고도 새로운 중간 단계 스케줄러를 설계하고 구현할 수 있다는 것이다. 이유는 낮은 단계 스케줄러는 커널의 구조와 기능에 밀접하게 연관되어 있는 반면, 중간 단계 스케줄러는 이를 기반으로 태스크 속성 변환 기능만을 담당하고 있기 때문이다. 따라서 새로운 스케줄링 알고리즘을 개발하고 실험하는 일이 보다 쉬워진다. 뿐만 아니라, 이미 다양한 중간 단계 스케줄러들을 확보하고 있을 경우, 구축하고자 하는 응용에 가장 적합한 것을 선택함으로써 손쉽게 새로운 시스템을 구축할 수 있다. 이는 동일한 커널로 다양한 실시간 응용을 지원하기 위해서는 반드시 필요한 스케줄러의 재구성 기능이다.

본 연구에서는 그림 1과 같은 두개의 구성 요소로 이루어진 재구성이 가능한 스케줄러 모델을 제안한다. 이 스케줄러 모델은 앞 절에서 논의한 스케줄링 알고리즘들을 수용한다. 제안 스케줄러 모델은 스케줄링 알고리즘을 구현한 상위 계층의 태스크 스케줄러와 이것에 의해 생성된 작업 속성들을 이용해 작업들을 구체적으로 디스패칭(dispatching)하는 하위 계층의 스케줄링 틀(*framework*)로 구성되며, 이들 두 구성 요소들은 계층적 상하 관계를 가진다. 또한 두 구성 요소들간 정보 교

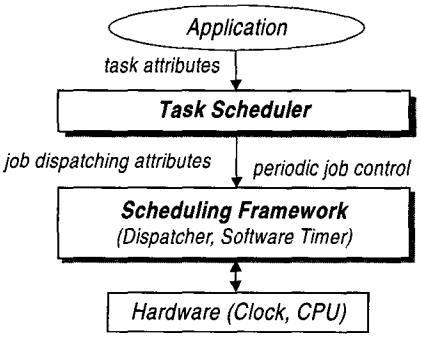

그림 1 두 단계 계층적 구조를 가진 실시간 스케줄러 모델

환을 위한 간단한 스케줄러 내부 인터페이스를 설계하였다. 응용은 응용 프로그래밍 인터페이스(API)를 통해 태스크 속성들을 태스크 스케줄러에게 전달하며, 태스크 스케줄러는 다시 스케줄러 내부 인터페이스를 통해 작업 속성과 타이머 속성을 스케줄링 틀에게 전달한다. 스케줄링 틀은 실행 가능한 작업들을 관리하며, 작업 속성에 따라 최우선순위를 가진 작업을 실행시켜 준다.

3.1 스케줄링 틀(framework)

제안 모델의 가장 작은 프로세서 스케줄링 단위는 작업(job)이다. 작업들은 단지 스케줄링 목적을 위해서 정의된다. 특정 시간에 임의의 태스크의 작업은 하나 또는 그 이상이 될 수도 있다. 이러한 작업들은 궁극적으로 스케줄링 틀에 의해 스케줄링 된다. 그림 2는 디스패처(dispatcher)와 소프트웨어 타이머로 구성된 스케줄링 틀의 내부 구조를 보여 주고 있다.

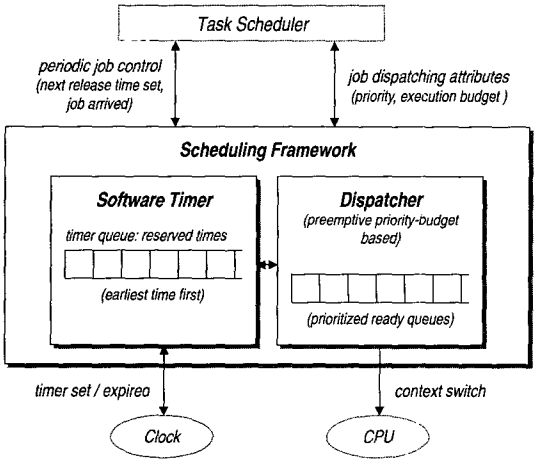

그림 2 스케줄링 틀의 내부 구조

디스패처는 실행 가능한 작업들을 준비 큐에 관리하면서, 이들 중 최우선순위를 가진 작업을 선택하여 프로세서를 할당한다. 소프트웨어 타이머는 스케줄러와 스케줄링 틀에 의해 예약된 시간들을 관리하면서, 예약 시간이 되면 사전 등록된 함수를 실행시켜 준다. 따라서 스케줄링 틀은 시스템 하드웨어와 밀접한 연관을 가진다. 스케줄링 틀은 상위 계층의 태스크 스케줄러로부터 작업 속성, 타이머 속성 등을 넘겨 받는다.

디스패처가 주어진 작업들을 관리하고 스케줄링하기 위해 필요한 작업 속성들은 유효 우선순위(*effective priority*), 절대 만기(*absolute deadline*), 실행 예산(*execution budget*) 등이다. 유효 우선순위는 시스템 내의 다른 작업들에 대한 상대적인 실행 우선권을 나타낸다. 실행 예산은 해당 작업에게 부여된 프로세서 할당 시간이다. 실행 예산을 모두 소진했을 때 해당 작업은 강제로 종료된다. 비록 작업이 끝나지 않았고 아직 실행 예산이 남아 있다 할지라도 만기 시간이 되면 이 역시 강제로 종료된다. 그러므로 이러한 작업 속성들은 실행 작업에 대한 제약 사항으로 작용한다. 그러나 이들은 또한 디스패처가 다음 실행 작업을 선택하는데 있어 중요한 선택 요소로 활용된다. 유효 우선순위는 다음 작업을 선별할 목적으로만 사용되며, 그 작업에 부여된 고유한 우선순위는 아니다.

디스패처는 모든 작업들을 우선순위 순으로 정렬하여 관리한다. 따라서 작업들을 정렬하고 최우선순위 작업을 선별하는 과정에서, 도착한 작업들의 수에 상관없이 항상 고정된 그리고 빠른 처리 시간을 준수하는 것이 무엇보다 중요하다. 디스패처는 최우선순위 작업에게 실행 예산 시간 동안만 프로세서를 할당해 주는 간단한 선점형 스케줄러이다. 이를 선점형 우선순위-실행예산-기반(preemptive priority-budget-based) 디스패처라 한다. 준비 큐에 도착한 작업들 중 항상 가장 높은 우선순위를 가진 작업에게 프로세서 사용권이 주어지고, 현재 수행중인 작업은 언제든지 높은 우선순위 작업에 의해 선점 당할 수 있다.

각 작업이 실행되는 매 시간 단위마다 작업의 실행 예산은 논리적으로 1씩 감소한다. 실행 예산을 모두 소진한 작업들은 강제로 실행이 중단된다. 그렇지 않은 작업들은 정상적으로 실행을 완료할 수 있다. 실행이 중단되었거나 완료한 작업들은 준비 큐에서 제거되고, 이러한 처리 결과는 상위 단계의 태스크 스케줄러에게 통보된다. 따라서 디스패처는 실행 예산 감시를 위해 각 작업마다 하나의 소프트웨어 타이머를 가진다.

소프트웨어 타이머는 태스크 스케줄러와 스케줄링 틀에 의해 예약된 시간들을 관리하면서, 예약된 시간이 되면 이를 통보해 준다. 태스크 스케줄러는 타이머 예약시 타이머 속성을 함께 제출해야 한다. 타이머 속성은 예약 시간, 콜백(*callback*) 함수, 함수 매개 변수 등으로 구성된다. 콜백 함수 및 함수 매개 변수는 예약 시간이 되면 호출되어야 할 함수와 호출될 당시 넘겨주어야 할 함수의 매개 변수이다. 소프트웨어 타이머는 예약된 많은 시간들을 예약 시간 순으로 정렬하여 하나의 리스트로 보관한다. 리스트상의 맨 처음 예약 시간이 되면 소프트웨어 타이머는 리스트상의 예약 시간들 중 현재 시간보다 같거나 이전인 모든 시간에 대해 사전에 등록된 해당 콜백 함수를 매개 변수와 함께 호출한다.

3.2 태스크 스케줄러

태스크 스케줄러는 스케줄링 알고리즘을 구현한 모듈로서 태스크 속성을 작업 속성으로 변환하는 기능을 담당한다. 태스크 스케줄러는 새로운 경성 태스크에 대한 수용 여부를 결정하고, 각 태스크의 새로운 작업 도착을 감시하면서 새로운 작업에 대해 작업 속성을 생성한다. 태스크 스케줄러의 이러한 행위는 스케줄링 알고리즘에 의해 결정된다. 스케줄링 알고리즘은 태스크 스케줄러로 하여금 언제, 어떤 작업에게, 얼마 만큼의 프로세서 시간을, 그리고 어떤 우선권을 부여할 것인지를 결정(즉, 작업 속성 결정)하게 한다.

태스크 스케줄러는 하위 단계의 스케줄링 틀을 기반으로 하나의 특정 스케줄링 알고리즘을 구현한다. 다른 요구 조건을 가진 다양한 실시간 응용을 만족시키기 위해서는 응용별 적합한 여러 종류의 알고리즘들이 필요하며, 이들은 동일한 스케줄링 틀을 기반으로 하나의 독립된 모듈로 존재할 수 있다.

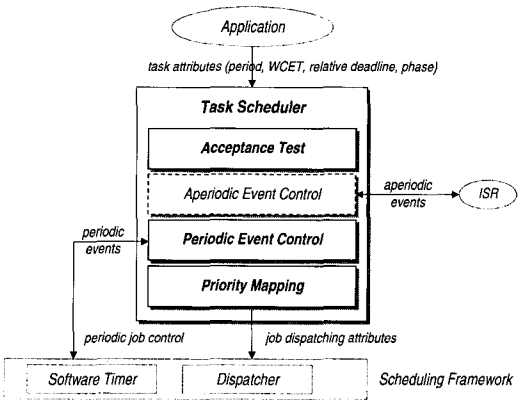

그림 3 태스크 스케줄러의 기능적 모델

태스크 스케줄러가 RM 알고리즘을 지원한다면, 이는 주기적으로 수행해야 하는 태스크에 대해 매 주기의 시작을 타이머에 설정해야 한다. 설정한 타이머가 작동하거나 또는 외부로부터 비주기적인 이벤트가 발생할 경우, 해당 태스크의 작업을 수행하기 위해 필요로 하는 프로세서 시간, 작업의 우선순위, 그리고 작업의 절대 만기 등을 결정한다. 그런 후, 해당 작업과 함께 앞서 결정된 작업 속성들을 정해진 스케줄러 내부 인터페이스를 통해 스케줄링 틀에게 넘겨 준다. 이러한 태스크 스케줄러의 행위는 분리 가능한 네 가지 독립적인 기능을 가진다. 그림 3은 태스크 스케줄러의 기능을 계층적으로 분리하여 도시화한 것이다.

새로 생성되는 모든 경성 태스크들은 이들의 만기가 준수될 수 있는지를 검증 받기 위해 수용 테스트(acceptance test)를 거쳐야 한다. 이 테스트는 기존 주기적 태스크들과 사전에 승인된 다른 비주기적 경성 태스크들의 만기를 준수하면서 새로운 태스크를 만기 내에 실행할 수 있는지를 체크 한다. 수용 테스트를 통과하지 못한 태스크들은 실행이 거절되고, 이러한 사실은 사전에 정의된 절차에 의해 사용자에게 통보된다. 이미 시스템 실행 전에 스케줄 가능성이 검증되고, 동적으로 태스크들을 생성/삭제할 필요가 없는 정적 시스템에서는 굳이 시스템 실행 중에 이러한 테스트를 수행할 필요는 없다. 따라서 태스크 스케줄러 내의 이 기능은 시스템의 적용 환경과 기반 스케줄링 알고리즘에 따라 시스템 구성시 선택적으로 추가/삭제가 가능하게 해야 한다.

주기적 태스크에 대해서는 주기적으로 각 태스크를 실행시켜야 한다 (그림 3의 주기적 이벤트 제어 부분). 이를 위해 태스크 스케줄러는 사전에 각 태스크의 주기 시작 시간을 타이머에 설정한다. 설정된 시간 순서에 따라 타이머가 작동하면 해당 태스크의 새로운 작업을 생성한다. 태스크의 최악의 경우 실행 시간을 작업의 실행 예산 속성으로 설정한다. 또한 그 태스크의 다음 작업을 위해 다음 주기의 시작 시간을 소프트웨어 타이머에 미리 설정한다.

비주기적 태스크에 대해서는 해당 태스크로 이벤트가 전달될 때, 그 태스크가 경성 또는 연성 태스크인가에 따라 처리가 달라진다. 경성 태스크인 경우, 주기적 태스크와 같이 해당 태스크의 새로운 작업을 생성하고, 태스크의 실행 시간을 그 작업의 실행 예산 속성으로 설정한다. 그러나 연성 태스크인 경우, 해당 태스크 작업을 서비스해 주는 전용 서버[16-19]에게 의뢰하거나 여유시간 활용(slack stealing) 기법[20, 21]을 적용해야 한다. 이러한 비주기적 태스크 스케줄링은 연구 범위상 본 연구에서 제외되었다. 그러나 성능 측정을 위해 시험적으로 구현된 Preemptive WFQ 알고리즘[15]은 Total Bandwidth Server [16]를 활용한다.

실행 예산이 설정된 작업들은 마지막 단계로 스케줄링 틀의 디스패처에 의해 사용될 우선순위를 할당 받아야 한다. 이 우선순위는 스케줄링 알고리즘별 할당 방법이 모두 다르다. 고정 우선순위 스케줄링 알고리즘(예: RM 또는 DM)인 경우, 해당 태스크 생성시 부여된 우선순위를 그대로 활용하면 된다. (이러한 알고리즘에서는 태스크의 우선순위와 작업의 우선순위는 항상 동일하다.) 그러나 동적 우선순위 알고리즘(예: EDF)은 작업이 생성될 때마다 그 태스크의 앞 전 작업과는 다른

우선순위(절대 만기)가 부여된다. 따라서 이 경우에는 작업의 절대 만기를 우선순위 값으로 설정한다. 공유 유도형 알고리즘(예: Preemptive WFQ)은 EDF를 기반 알고리즘으로 사용하므로 EDF와 동일한 방법으로 각 서버의 우선순위를 할당한다. 시간 유도형(예: Cyclic Scheduler)은 시스템 내에 실행 가능한 작업은 항상 하나 뿐이므로, 우선순위의 의미는 없다.

4. 스케줄러 모델의 구현

본 연구에서는 기반 스케줄링 틀을 바탕으로 다양한 스케줄링 알고리즘을 지원하고, 스케줄링 알고리즘별 독립된 스케줄러 모듈을 제공하기 위해 스케줄러 내부 표준 인터페이스를 정의하였다. 이 인터페이스는 태스크 관리자와 태스크 스케줄러간의 태스크 스케줄러 인터페이스와 태스크 스케줄러와 스케줄링 틀간의 스케줄링 틀 인터페이스로 구성된다.

4.1 스케줄링 틀 인터페이스

스케줄링 틀 인터페이스는 태스크 스케줄러와 스케줄링 틀 사이의 인터페이스로서, 스케줄링 틀이 제공하는 함수 프로토타입들의 집합이다 (그림 4 참조). 이는 하드웨어와 밀접한 연관이 있는 하위 단계의 복잡한 커널 함수와 데이타 구조를 숨기고, 보다 간단하게 상위 단계의 태스크 스케줄러를 구현할 수 있게 해주는 최소한의 함수들로 구성된다. 따라서 시스템 개발자는 주어진 스케줄링 틀 인터페이스만 준수한다면, 시스템 하드웨어와 하위 단계의 복잡한 커널 모듈을 완벽하게 분석하지 않고도 독립적으로 태스크 스케줄러를 개발할 수 있다.

```
/*Dispatcher AP */
rtl_disp_ready(rtl_disp_t jo)rtl_disp_unready(
rtl_disp_t jo)
rtl_disp_dispatch(voidrtl_disp_register_budget_exhaust(
(*callback)())

/*Software Timer AP */
rtl_timer_set(*timer, (*callback)(*), *callback_data)
rtl_timer_start(*timer, when)
rtl timer stop(*timer)
```

그림 4 스케줄링 틀의 주요 인터페이스

스케줄링 틀 인터페이스는 대부분 태스크 스케줄러가 하위 단계인 스케줄링 틀의 서비스를 받기 위한 인터페이스지만 (down call), rtl_disp_register_budget_exhaust()나 rtl_timer_set()으로 등록된 콜백(callback) 함수를 통해 스케줄링 틀에서 태스크 스케줄러로 역으로 호출(up call)되기도 한다. rtl_disp_ready()는 주어진 작업(job)을 스케줄링 틀의 구성 요소 중 하나인 디스패처의 준비 큐에 삽입하고, rtl_disp_unready()는 준비 큐에서 해당 작업을 삭제한다. rtl_disp_dispatch()는 현재 준비 큐 내의 작업들 중 최우선순위 작업을 선택하여 문맥교환을 실시한다. 이러한 함수 호출 시점은 상위 단계의 태스크 스케줄러에 의해 결정된다. 디스패처는 태스크 스케줄러에 의해 결정된 우선순위, 실행 예산(execution budget) 등과 같은 작업 속성을 기반으로 디스패칭 한다. 현재 수행 중인 작업이 주어진 실행 예산을 초과하여 실행할 경우엔 rtl_disp_register_budget_exhaust()를 통해 등록된 콜백 함수(스케줄러가 사전에 등록)를 호출한다. 이 작업의 계속적인 수행 여부는 상위 단계의 스케줄러에 의해 결정된다.

스케줄링 틀의 소프트웨어 타이머는 주기적 태스크의 도착 시간이나 현재 실행 중인 작업의 실행 예산 초과 여부 등을 결정하기 위해 태스크 스케줄러와 디스패처에 의해 사용된다. rtl_timer_set()를 통해 콜백 함수와 이 함수의 매개변수 등을 사전 등록하고, rtl_timer_start()를 이용하여 원하는 시간을 예약한다. 예약된 시간이 되면 타이머는 사전에 등록된 콜백 함수를 매개변수와 함께 호출한다. 현재 작동 중인 타이머는 rtl_timer_stop()를 통해 중지시킬 수 있다. 소프트웨어 타이머는 필요에 따라 언제든지 만들 수 있으며, 개수 제한은 없다.

4.2 태스크 스케줄러 인터페이스

태스크 스케줄러 인터페이스는 태스크 관리자와 태스크 스케줄러 사이에 준수해야 할 인터페이스이며, 특정 스케줄링 알고리즘을 구현하는 태스크 스케줄러가 제공해야 하는 함수 프로토타입들의 집합이다. 태스크 스케줄러는 이 인터페이스를 통해 외부와 통신할 수 있다.

```
/* SPI function set provided by a specific scheduler */
typedef struct {
    void (*thread_make_rt_task)(*thread);
    void (*thread_created)(*thread);
    void (*thread_deleted)(*thread);
    void (*thread_wait)(void)
    void (*thread_release)(void);
} rtl_sched_ops_t;
static rtl_sched_budget_exhasut*threa);

/* 스케줄러 모듈 초기화시 호출해야 할 스케줄러 등록 함 */
rtl_register_scheule(class,rtl_sched_ops_*opsisdefault);
rtl_unregister_scheule(class);
rtl_change_scheduler(class);
```

그림 5 태스크 스케줄러 인터페이스와 관련 등록 함수

시스템 개발자는 특정 스케줄링 알고리즘을 구현하고자 할 경우, 그림 5의 스케줄러 클래스 멤버 함수들(rtl_sched_ops_t)을 정의한 후, 이를 스케줄러 초기화 때 관련 등록 함수(rtl_register_scheduler())를 통해 시스템에 등록하여야 한다. rtl_register_scheduler()의 매개 변수 class는 등록하고자 하는 스케줄러의 종류를 의미하며, RTL_SCHED_RM, RTL_SCHED_EDF, RTL_SCHED_CE (cyclic executive), RTL_SCHED_PWFQ (preemptive weighted fair-queue) 등을 지정할 수 있다. 이외에도 시스템 개발자가 새로운 태스크 스케줄러를 개발할 경우 별도의 클래스를 정의하여 추가할 수 있다. 태스크 스케줄러 등록 함수들은 커널 내의 태스크 관리자에서 제공되며, 스케줄러 클래스 멤버 함수들은 주로 커널 내의 태스크 관리자에 의해 호출된다.

태스크 스케줄러는 시스템 재구성시 정적 또는 동적으로 (동적 재구성을 지원하는 시스템의 경우) 시스템에 삽입/삭제 가능하다. 하나의 시스템에 여러 개의 태스크 스케줄러(서로 다른 스케줄링 알고리즘)를 등록할 수 있으나, 항상 하나의 태스크 스케줄러만 사용할 수 있다. 그러나 현재 수행 중인 태스크 집합의 수행이 끝났거나 또는 일부 태스크가 종료하고 다른 태스크 집합으로 대체하는 실행 모드 전환을 수행한 경우에는, rtl_change_scheduler()를 통해 다른 태스크 스케줄러를 사용할 수 있다.

5. 제안 모델의 타당성 검증

본 연구에서는 제안 실시간 스케줄러 모델의 타당성을 검증하기 위해, 이를 실시간-리눅스(RT-Linux) 상에 시험적으로 구현하였다. 기반 실시간-리눅스는 Linux 2.2.14에 구현된 RT-Linux 2.2를 수정/확장하였다. 본 연구는 ETRI의 정보가전용 실시간 OS(Q+ 커널) 개발 사업의 일환으로 수행되었으며, 실제 Q+ 커널에도 시험적으로 구현되었다. 그러나 본 논문에서 실시간-리눅스를 택한 이유는 우선 실시간-리눅스가 기존 리눅스와 같이 소스 개방 정책을 따르고 있고, 다양한 리눅스 사용자 및 실시간 연구자들이 범용 시스템(리눅스)에서 손쉽게 실시간 시스템(실시간 리눅스) 기술을 접할 수 있기 때문이다. 이러한 시스템에 제안 스케줄러 모델을 구현하고 이를 공개함으로써, 보다 많은 연구자들이 제안 모델을 활용하거나 또는 이를 바탕으로 다양한 알고리즘을 개발하고 테스트 할 수 있는 기반 환경으로 활용할 수 있기 때문이다.

본 절에서는 다양한 실시간 스케줄러들을 하부의 복잡한 커널 메커니즘 수정 없이 독립적으로 개발할 수 있는가를 확인하고, 또한 제안 스케줄러 모델의 스케줄링 오버헤드와 타 시스템으로의 이식성을 검토함으로써, 제안 모델의 타당성을 검증하고자 한다.

5.1 스케줄러의 개발 및 재구성 용이성

실시간-리눅스는 rtl_time.o, rtl_sched.o, rtl_fifo.o, rtl_posix.o 등의 동적으로 시스템에 삽입 가능한 네 개의 모듈로 구성되어 있으며, 이 중 본 연구와 밀접한 연관이 있는 rtl_sched.o 모듈은 태스크 관리자, RM 스케줄러, mutex 등을 포함한다. 본 연구에서는 기존 실시간-리눅스의 rtl_sched.o를 제거하고, 제안 스케줄링 틀을 구현한 rtl_thread.o로 대체하였다. rtl_thread.o는 기존 실시간-리눅스와 동일한 태스크 모델과 인터페이스를 지원한다. 이를 위해 rtl_thread.o는 rtl_sched.o 모듈 내의 RM 스케줄러를 제외한 태스크 관리자와 mutex를 포함하며, 제안 스케줄링 틀의 구성 요소인 소프트웨어 타이머, 디스패처 등을 새로이 포함한다. 따라서 rtl_thread.o의 모듈 크기(12KB)는 이와 상응하는 실시간-리눅스 모듈인 rtl_sched.o(10KB)보다 2 KB 가까이 늘어 났으며, 실제 rtl_sched.o의 RM 스케줄러를 제외하고 고려하면 약 4 KB 정도 늘어 났다고 볼 수 있다. 이는 제안 스케줄러 모델의 메모리 사용 오버헤드로 볼 수 있으며, 메모리 사용에 민감한 소형 시스템을 제외한 일반적인 내장형 시스템에서는 충분히 수용될 수 있는 오버헤드로 판단된다.

본 연구의 목적 중 하나는 새로운 태스크 스케줄러를 설계, 구현, 테스트할 수 있는 스케줄링 틀을 제공하고, 이를 기반으로 다양한 실시간 스케줄러를 손쉽게 구현할 수 있도록 하는데 있다. 이러한 전략의 타당성을 확인하기 위해 본 연구에서는 2절에서 논의한 각 스케줄링 기법의 대표적인 알고리즘인 RM(DM 포함), EDF, Cyclic Executive, Preemptive WFQ 등의 태스크 스케줄러들을 구현하여 보았다. 이중 Preemptive WFQ는 EDF 스케줄러상의 Total Bandwidth Server를 이용하여 구현하였다.

이러한 시험적 구현을 통해 하위단계의 복잡한 커널 모듈에 영향을 받지 않고도, 제안 모델이 제공하는 스케줄링 틀과 내부 표준 인터페이스를 준수하면서, 다양한 특성을 가진 서로 다른 스케줄링 알고리즘들을 손쉽게 구현할 수 있다는 것을 확인하였다. 이는 제안 스케줄러 모델이 정책과 메커니즘의 분리(policy/mechanism separation) 기법[22]을 제공하기 때문에 가능하다. 태스크 스케줄러는 태스크의 속성을 작업 속성으로 변환하고, 스케줄링 틀은 이 작업 속성을 참고하여 수행할 작업을 선택한다. 여기서 태스크 스케줄러는 정책을 정

의하고, 스케줄링 틀은 메커니즘을 제공한다. 이런 식으로 스케줄링 정책을 스케줄링 메커니즘으로부터 분리함으로써, 시스템 개발자는 응용의 요구에 가장 적합한 스케줄링 정책만을 정의하고, 이를 하위 단계의 스케줄링 틀을 이용하여 구현하면 된다. 이 과정에서 스케줄링 틀을 수정할 필요는 없다.

제안 모델의 또 다른 이점은 시스템 재구성(조립성)을 보다 유연하게 해준다는 것이다. 본 연구에서는 각 스케줄링 기법별로 널리 사용되고 그 성능이 입증된 다수개의 스케줄링 알고리즘들을 제안된 스케줄링 틀을 바탕으로 실험적으로 구현하였으며, 이들을 하위 단계의 기본 틀인 스케줄링 틀과 함께 제공한다. 시스템 구축자는 매번 기존 알고리즘을 새로이 구현할 필요 없이, 이미 제공된 다양한 알고리즘들 중 구축하고자 하는 시스템에 가장 적합한 것을 선택하여 시스템을 재구성할 수 있다. 이는 제안 모델이 구성 요소들을 기능별로 독립된 모듈로 분리하여 지원하기 때문에 가능하며, 이로 인해 기존 개발된 소프트웨어 모듈들의 재사용과 시스템 재구성을 보다 용이하게 해주는 효과를 얻을 수 있다.

5.2 스케줄링 오버헤드(overhead)

본 절에서는 제안 모델상의 스케줄러와 기존 일체형 스케줄러의 스케줄링 오버헤드를 비교함으로써, 계층적으로 분리된 제안 모델의 상대적인 스케줄링 오버헤드가 얼마나 더 큰지를 확인하고자 한다.

스케줄링 오버헤드로 주기적 태스크의 활성화/비활성화 지연을 측정하였다. 주기적 태스크의 활성화 지연이란 우선순위가 가장 높은 태스크가 도착하여 이 태스크가 최초로 실행되기까지 소요된 총 시간을 말한다. 주기적 태스크의 비활성화 지연이란 우선순위가 가장 높은 태스크가 현 주기 작업을 모두 마치고, 그 다음 우선순위의 태스크로 문맥 교환을 실시하는데 소요된 총 시간을 의미한다. 성능 측정에 사용된 시스템은 Pentium II MMX 233MHz, 32KB L1 Cache(Enabled), 512KB L2 Cache(Enabled), 128MB RAM, 33MHz PCI bus 등의 하드웨어사양을 가진다. 본 연구에서는 펜티엄 프로세서에서 제공하는 TSC(time stamp counter)를 이용하여 성능 측정을 하였다.

그림 6과 그림 7은 각각 스케줄러별 주기적 태스크의 활성화/비활성화 지연을 보여 준다. 각 그림의 x축은 준비 큐에 이미 도착해 있는 태스크의 개수이고, y축은 태스크 개수별 활성화/비활성화 지연 시간이다. 그림에서 RM과 EDF는 본 연구에서 제안한 스케줄링 틀상에 구현된 주기적 태스크 스케줄러이며, RTL-RM는 기존 실시간-리눅스가 제공하는 Rate Monotonic 스케줄러이다.

그림에서 RM은 태스크들이 항상 고정된 우선순위를 가지고 최우선순위 태스크를 고정된 상수 시간 내에 찾아내는 알고리즘[26]을 사용함으로써, 준비된 태스크의 개수와는 무관하게 항상 고정된 활성화/비활성화 지연을 보인다. 반면, EDF의 경우 동적 우선순위를 가지므로 최우선순위 태스크를 찾기 위해 준비된 태스크를 순차적으로 찾아야 한다. 따라서 준비된 태스크 수에 비례하여 활성화 지연이 증가한다. 그러나 비활성화 지연은 태스크 개수와 상관없이 항상 고정된 시간을 보인다. 이는 태스크를 준비 큐에 삽입할 당시 동적 우선순위에 따라 태스크를 정렬하여 리스트로 관리하므로, 최우선순위 태스크가 종료하고 난 후 별도의 리스트 검색 없이 바로 그 다음 준비된 태스크를 선택할 수 있기 때문이다. 실시간-리눅스의 일체형 스케줄러(RTL-RM)는 비록 고정 우선순위이지만 항상 모든 태스크 리스트를 순차적으로 따라 가면서 최우선순위 태스크를 찾기 때문에 태스크 수에 비례하여 활성화/비활성화 지연이 증가한다.

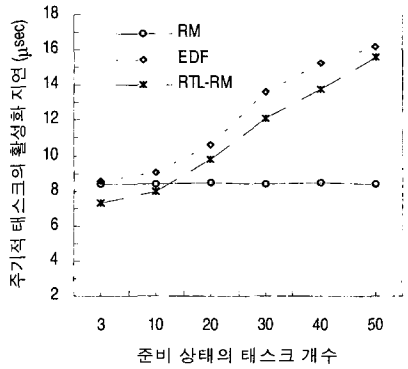

그림 6 주기적 태스크의 활성화 지연

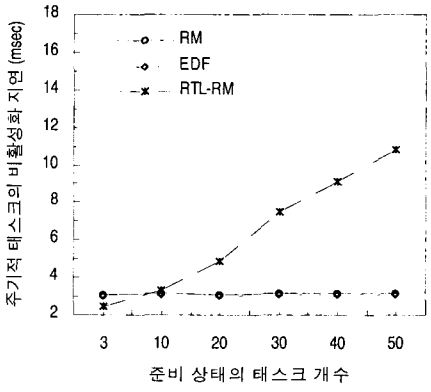

그림 7 주기적 태스크의 비활성화 지연

실험에 사용된 각 스케줄러는 서로 다른 스케줄링 목적과 구현 방법을 사용하므로 이들 간의 활성화/비활성화 지연시간을 직접적으로 비교하는 것은 의미가 없다. 그러나 본 연구에서는 간접적으로 이들을 비교함으로써, 두 단계 계층적 구조를 가진 제안 스케줄러 모델상에서 구현된 스케줄러(RM, EDF)라 할지라도 기존 일체형 스케줄러(RTL-RM)보다 스케줄링 오버헤드가 크게 차이가 나지 않는다는 것을 확인하고자 한다.

RTL-RM은 순차적으로 각 태스크의 주기(우선순위)를 비교함으로써 최우선순위 태스크를 선별하는 단순한 알고리즘을 사용한다. 이는 곧 태스크 개수가 많을수록 지연시간이 증가하지만, 반대로 태스크 개수가 적을수록 오히려 최적에 가까운 스케줄링 지연시간을 가진다는 것을 의미한다. 이런 측면에서 그림 6과 그림 7의 준비된 태스크 개수가 3일 때 RM과 RTL-RM의 스케줄링 지연을 비교해 보면, 약 1.5 μsec 정도의 활성화 지연 차이와 약 0.8 μsec 정도의 비활성화 지연 차이를 보인다. 이는 곧 제안 스케줄러 모델의 상대적인 스케줄링 오버헤드로 간주될 수 있다. 또한 EDF의 경우 순차적으로 각 태스크의 절대만기(우선순위)를 비교함으로써 최우선순위 태스크를 선별한다는 면에서, RTL-RM과 유사한 구현 방법(순차적 태스크 리스트 탐색)을 사용한다. 따라서 그림 6에서 두 스케줄러 모두 태스크 개수가 증가하면서 활성화 지연도 증가하는 것을 볼 수 있다. 이 그림에서 전반적으로 EDF가 RTL-RM보다 약 1.5 μsec 정도 더 큰 활성화 지연을 보이는데, 이 또한 제안 스케줄러 모델의 상대적인 스케줄링 오버헤드로 간주된다.

이상의 주기적 태스크의 활성화/비활성화 지연 실험 결과와 간접적인 비교를 통해, 두 단계 계층적 구조를 가진 제안 스케줄러 모델이 기존 일체형 스케줄러보다 약 2 μsec 범위 내의 상대적인 스케줄링 오버헤드를 가진다는 것을 확인할 수 있다. 이는 곧 제안 스케줄러 모델이 스케줄링 기반 틀을 제공하고, 이를 기반으로 태스크 스케줄러를 분리하여 독립된 모듈로 존재하게 할지라도, 스케줄링 오버헤드는 기존 일체형 스케줄러와 크게 차이가 나지 않음을 확인할 수 있다. 반면, 앞 절에서 논의한 바와 같이 스케줄러 재구성과 새로운 스케줄러 개발을 효과적으로 지원할 수 있는 이점을 제공한다.

5.3 제안 모델의 이식성

스케줄링 틀 내의 디스패처는 대부분의 전통적인 RTOS들이 지원하는 선점형 우선순위-기반 스케줄러와 유사하다[23-25]. 이 스케줄러는 사용자에 의해 주어진 우선순위를 기반으로 최우선순위 작업에게 프로세서를 할당해 준다. 또한 대부분의 이러한 스케줄러는 동일한 우선순위를 가진 작업이 여러 개일 경우, 선택적으로 라운드-로빈 방식을 사용할 수 있게 해준다. 즉, 사전에 사용자 또는 시스템에 의해 정의된 시분할(time slice) 만큼의 프로세서 시간을 동일 우선순위 작업들에게 분배한다. 본 연구의 디스패처 역시 상위 단계의 태스크 스케줄러에 의해 주어진 실행 예산 만큼의 프로세서 시간을 각 작업에게 할당하는 기능을 가지고 있다. 따라서 기존 라운드-로빈을 지원하는 선점형 우선순위-기반 스케줄러와 본 연구의 디스패처와는 고정 우선순위에 기반한 동일한 스케줄링 원리를 가진다. 따라서 기존 시스템들은 본 연구의 디스패처와 간단한 함수들의 집합인 소프트웨어 타이머를 많은 수정 없이 손쉽게 수용할 수 있다. 이는 곧 기존 시스템에 본 연구의 스케줄링 틀을 쉽게 이식할 수 있다는 것을 의미하며, 이 경우 상위 스케줄러를 많은 수정 없이 수용할 수 있다. 즉, 제안 스케줄링 틀만 이식되면, 이를 기반으로 개발된 기존의 다양한 실시간 스케줄링 알고리즘들을 많은 수정 없이 이식할 수 있다는 것을 의미한다.

6. 관련 연구

지금껏 수 많은 실시간 운영체제가 개발되었다. 그러나 대부분의 실시간 운영체제는 하나의 기반 스케줄링 알고리즘만을 제공한다. RT-Mach는 기반 스케줄링 알고리즘으로 고정 우선순위 및 라운드-로빈(fixed priority & round-robin)과 RM을 지원한다[27]. 반면, HARTIK [28]은 EDF를 기반 스케줄링 알고리즘으로 사용하며, 비주기적 작업을 위한 TBS[16] 서버를 지원한다. North Carolina 대학(university of North Carolina)에서 추진 중인 Free BSD 프로젝트[29, 30]는 비례적 공유 스케줄링을 지원하고, Rialto[31]는 시간 기반형 스케줄링 기법을 제공한다. 또한 기존의 VxWorks[23], pSOS[24], VRTX[25] 등과 같은 상용 RTOS들은 선점형 우선순위 기반(preemptive priority-based) 스케줄러나 라운드 로빈(round-robin) 스케줄러와 같이 극히 제한된 실시간 스케줄러만 제공한다.

Illinois 대학(university of Illinois)에서 추진 중인 개방 환경(Open Environment) 프로젝트는 복잡한 응용들을 상호 독립적으로 설계/검증하고 스케줄링할 수 있는 두 단계 계층적 스케줄러를 설계하고[17], 이를 구현하였다[32]. 이 연구에서 제시된 이론적 기반을 통해 서로 다른 스케줄러를 사용하는 서로 다른 응용들에게 독립적으로 스케줄 가능성을 보장해 줄 수 있다. 모든 응용들(각 응용은 다시 여러 개의 태스크들로 구성됨)은 두

단계 계층적 스케줄러의 하위 단계인 EDF 스케줄러에 의해 스케줄링 된다. 각 응용은 사전에 정의된 서버에 의해 서비스되고, 각 서버는 응용별 고유한 스케줄러를 이용해 해당 응용의 작업들을 스케줄링 한다. 개방 환경 전략은 동시에 서로 다른 스케줄러를 사용해야 하는 복잡한 응용들을 지원해야 하는 대형 시스템에 적합하며 (실제 이 전략은 여러 실시간 응용과 비실시간 응용들을 동시에 지원해야 하는 Window NT등에 구현되었음 [32]), 일반적으로 하나의 응용만을 지원하고 그 응용에 적합한 하나의 실시간 스케줄러를 요하는 대부분의 내장형 실시간 시스템에는 적합하지 않다. 이에 비해 본 연구에서는 동일한 실시간 커널을 활용하되 서로 다른 스케줄러를 요하는 다양한 내장형 실시간 응용들(각 응용별로 시스템을 정적으로 재구성해야 함)을 위한 기반 스케줄링 틀과 스케줄러 재구성 방안 등을 제안하였다.

본 연구와 유사한 연구는 RED(Real-Time and Embedded)-Linux프로젝트이다 [6]. RED-Linux 는 기존 실시간-리눅스를 기반으로 시스템 유연성을 향상시키기 위해 범용 스케줄링 틀을 제공한다. 이 틀은 우선순위 기반형, 시간 기반형, 공유 기반형 등 세가지 스케줄링 기법별로 별도의 디스패처를 기본 커널 모듈로 제공하고, 각 스케줄링 알고리즘별 전략은 별도의 서버 태스크(할당자라고 함)에서 결정한다. 이 할당자는 가장 우선순위가 높은 태스크 형태로 실행되면서 각 작업의 시작시간, 우선순위 결정요소(주기, 절대만기, 또는 시작시간 등), 실행 예산 등을 결정하여 디스패처에 통보한다. 디스패처는 각 작업의 시작시간을 자신이 제어하면서 시작시간이 되면 할당자가 지정한 우선순위 결정요소에 따라 최우선순위 태스크를 선택한다. 할당자를 별도의 서버 태스크로 분리함으로써 사용자가 직접 스케줄링 알고리즘을 선택하거나 변경할 수 있는 장점은 있으나, 할당자와 디스패처간 빈번한 정보 교환이 요구될 경우 과도한 문맥교환 오버헤드가 발생할 수 있다. 이에 반해 본 연구에서는 각 작업의 시작시간과 우선순위 결정을 하위 단계 디스패처에서 제어하도록 하지 않고, 상위 단계의 태스크 스케줄러에서 하도록 했다. 반면 각 작업의 시작시간 제어를 위한 메커니즘으로 하위 단계에 별도의 소프트웨어 타이머를 제공하고 있다. 따라서 본 연구의 디스패처는 RED-Linux에서 처럼 자신이 직접 작업들의 시작시간을 제어하지 않으며, 우선순위도 상위 태스크 스케줄러가 결정한 값을 그대로 활용할 뿐이다. 또한 상위 태스크 스케줄러는 RED-Linux의 할당자와 달리 별도의 태스크로 존재하는 것이 아니라, 커널을 구성하는 독립된 스케줄러 모듈로 존재한다.

7. 결론

본 연구에서는 실시간 응용들의 다양한 요구 사항을 충족시킬 수 있는 여러 실시간 스케줄링 알고리즘을 지원할 수 있는 재구성 가능한 스케줄러 모델을 제안하였다. 제안 스케줄러 모델은 기본적인 작업 디스패처와 소프트웨어 타이머를 제공하는 하위 계층의 스케줄링 틀과 이를 기반으로 응용에 적합한 다양한 스케줄링 알고리즘들을 구현할 수 있는 상위 계층의 태스크 스케줄러로 구성된다. 실시간-리눅스상에 제안된 스케줄러 모델을 구현하고 성능 측정을 해 본 결과, 두 단계 계층적 구조를 가진 제안 모델이 하나로 통합된 기존 시스템과 성능상의 차이가 거의 없다는 것을 확인할 수 있었다. 또한 스케줄링 틀상에 대표적인 스케줄링 알고리즘들을 시험적으로 구현하여 봄으로써, 새로운 알고리즘을 하부의 복잡한 커널 메커니즘 수정 없이 독립적으로 개발할 수 있음을 확인하였다.

전통적으로 내장형 제어 시스템과 같은 경성 실시간 시스템의 경우 주기적 경성 태스크들로만 구성되지만, 오늘날 실시간 시스템의 활용 범위가 확대되면서 주기적 경성/연성 태스크와 비주기적 경성/연성 태스크 등이 복합적으로 구성된 시스템에 대한 요구가 늘어 나고 있다. 따라서 본 연구에서는 또한 이러한 복합 태스크 집합을 스케줄링하기 위한 제안 모델의 확장 작업을 추진하고 있다.

실시간 시스템 설계시 고려해야 할 또 다른 중요한 요소는 시스템 자원의 효율적인 사용이다. 교착상태를 방지하고 우선순위 역전(priority inversion)으로 인한 블록킹(blocking) 시간을 한정시키기 위해, 우선순위 승계 전략(priority inheritance protocol), 우선순위 높임 전략(priority ceiling protocol), 스택 자원 전략(stack resource policy) 등과 같은 여러 자원 관리 정책들이 제안되었다. 이러한 자원 관리 정책들은 태스크 스케줄링 알고리즘과 밀접한 연관성을 가진다. 따라서 경성 태스크를 위한 자원 관리 정책은 기반 스케줄링 알고리즘과 별개로 지원될 수 없으며, 스케줄링 알고리즘 설계시 함께 고려되어야 한다. 따라서 본 연구에서는 제안된 스케줄러 모델에서 재구성을 지원하면서 자원과 태스크를 통합적으로 스케줄링 할 수 있는 모델에 대해 연구하고 있다.

참 고 문 헌

[1] 남민우, "실시간 OS 시장 동향," 한국 정보처리학회 논문지, 제5권, 제4호, Jul. 1998.

[2] 한국전자통신연구원, "조립형 실시간 OS 개발 사업," 기술 보고서, Jan. 1999.
[3] 이윤석, "RTOS 성능 평가 방법 및 현황," 한국정보처리학회 정보가전용 실시간 OS 컨퍼런스 자료집, pp 73-83, Nov. 2000.
[4] 최경희, "스케줄러의 재구성을 지원하기 위한 실시간 커널의 구조," 한국정보처리학회 정보가전용 실시간 OS 컨퍼런스 자료집, pp 45-56, Nov. 2000.
[5] Michael Barabanov, "A Linux-based Real-Time Operating System," Master thesis, New Mexico Institute of Mining and Technology, June 1997.
[6] Yu-Chung Wang and Kwei-Jay Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," *Proc. of 20th IEEE Real-Time Systems Symposium*, Dec. 1998.
[7] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus, "Temporal Resolution and Real-Time Extensions to Linux," Technical Report ITTC-FY98-TR-11510-03, Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998.
[8] Jane W. S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.
[9] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 40-61, 1973.
[10] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, No. 2, pp. 237-250, 1982.
[11] A. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environments," *PhD. Dissertation*, MIT, 1983.
[12] Ching-Chih Han, Kwei-Jay Lin, and Chao-Ju Hou, "Distance-Constrained Scheduling and its Applications to Real-Time Systems," *IEEE Transaction on Computers*, Vol. 45, No. 7, pp. 814-826, Dec. 1996.
[13] H. Kopetz, "Time-Triggered Model of Computation," *Proc. of Real-Time Systems Symposium*, pp. 168-177, Dec. 1998.
[14] T. P. Bake and A. Shaw, "The Cyclic Executive Model and Ada," *Proc. IEEE Real-Time Systems Symposium*, pp. 120-129, Dec. 1988.
[15] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queuing Algorithm," *Journal of Internetworking Research and Experience*, pp. 3-26, Oct. 1990.
[16] M. Spuri and G. C. Buttazzo, "Efficient Aperiodic Service under the Earliest Deadline Scheduling," *Proc. of Real-Time Systems Symposium*, pp. 2-11, Dec. 1994.
[17] Z. Deng and Jane W. S. Liu, "Scheduling Real-Time Applications in an Open Environment," *Proc. of the 18th IEEE Real-Time Systems Symposium*, pp. 308-319, Dec. 1997.
[18] J. P. Lehoczky, L. Sha, and J. K. Stronider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. of Real-Time Systems Symposium*, pp. 261-270, Dec.1987.
[19] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *The Journal of Real-Time Systems*, No. 1, pp. 27-60, 1989.
[20] J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Priority Preemptive Systems," *Proc. of Real-Time Systems Symposium*, pp. 110-123, 1992.
[21] S. Ramos-Thuel and J. P. Lehoczky, "On-line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems," *Proc. of Real-Time Systems Symposium*, pp. 160-171, 1993.
[22] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Sulf, "Policy/Mechanism Separation in HYDRA," *Proc. of Symposium on Operating Systems Principles*, Nov. 1975.
[23] Wind River Systems, *VxWorks Programmer"s Guide 5.3.1*, Edition 1, 1996.
[24] Integrated Systems, *pSOSystem Programmer"s Reference*, 1997.
[25] Microtec Division, *VRTX User"s Guide*, Mentor Graphics Corporation, 1998.
[26] Jean J. Labrosse, *MicroC/OS-II The Real-Time Kernel*, R&D Books, pp 75-104, 1999.
[27] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," *Proc. of USENIX 1st Mach Symposium*, Oct. 1990.
[28] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli, "HARTIK 3.0: a Portable System for Developing Real-Time Applications," *Proc. of 4th International Workshop on Real-Time Computing Systems and Applications*, Oct. 1997.
[29] K. Jeffey, F. D. Smith, A. Moorthy, and J. Anderson, "Proportional Share Scheduling of Operating System Services for Real-Time Applications," *Proc. of Real-Time Systems Symposium*, pp. 480-491, Dec. 1998.
[30] I. Stoica, H. Abdel-Wahab, K. Jeffey, S. Baruah, J. Gehrke, and G. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time,

Time-shared Systems," *Proc. of Real-Time Systems Symposium*, pp. 288-299, Dec. 1996.
[31] M. B. Jones, D. Rosu, and M-C. Rosu, "CPU Reservation and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. of the 16th ACM Symposium on Operating System Principles*, pp. 198-211, Oct. 1997.
[32] Z. Deng, J. W. S. Liu, and J. Sun, "An Open Environment for Real-Time Applications," *Real-Time Systems Journal*, Vol. 16, No. 2/3, pp. 155-186, May 1999.

심 재 홍

1987년 서울대학교 전산과학과 졸업(학사). 1989년 아주대학교 컴퓨터공학과 졸업(석사). 2001년 아주대학교 컴퓨터공학과 졸업(박사). 1989년 ~ 1994년 서울시스템(주) 공학연구소. 2001년 ~ 2001년 9월 아주대학교 정보통신전문대학원 BK21 전임연구원. 2001년 10월 ~ 현재 조선대학교 인터넷소프트웨어공학부 전임강사. 관심분야는 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템

송 재 신

1983년 충남대학교 전자공학과 졸업(학사). 1990년 원광대학교 전자공학과 졸업(석사). 2000년 아주대학교 컴퓨터공학과 박사과정(수료). 1984년 ~ 1991년 ETRI 연구원, 중학교, 고등학교 교사. 1991년 ~ 1997년 한국교육개발원 부연구위원. 1997년 ~ 현재 한국교육학술정보원 연구위원. 관심분야는 Web-Based Instruction, 시스템 프로그램, 교육정보화 등

최 경 희

1976년 서울대학교 수학교육과 졸업(학사). 1979년 프랑스 그랑데꼴 Enseeiht대학 졸업(석사). 1982년 프랑스 Paul Sabatier대학 정보공학부 졸업(박사). 1982년 ~ 현재 아주대학교 정보통신전문대학원 교수. 관심분야 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템 등

박 승 규

1974년 서울대학교 응용수학과 졸업(학사). 1976년 한국과학기술원 전산학과 졸업(석사). 1982년 프랑스 Institute National Polytechnique de Grenoble 전산학과 졸업(박사). 1976년 ~ 1992년 KIST, KIET, IBM 왓슨 연구소, ETRI 연구위원. 1992년 ~ 현재 아주대학교 정보통신전문대학원 교수. 관심분야 컴퓨터 구조, 멀티미디어, 실시간시스템, 이동컴퓨터 등

정 기 현

1984년 서강대학교 전자공학과 졸업(학사). 1988년 미국 Ilinois주립대 EECS 졸업(석사). 1990년 미국 Purdue대학 전기전자공학부 졸업(박사). 1991년 ~ 1992년 현대반도체 연구소. 1993년 ~ 현재 아주대학교 전자공학부 교수. 관심분야는 컴퓨터구조, VLSI 설계, 멀티미디어 및 실시간 시스템 등