# 리눅스 미들웨어(TMOSM/Linux)에서 주기성을 가진 실시간 태스크의 스케쥴링 향상에 관한 연구

박 호 준[†] · 이 창 훈[††]


## 요 약

실시간 응용 제품을 개발하기 위해 운영체제는 실시간 태스크의 시간 보장성(timeliness guarantee)이 지원되어야한다. 그러나 현재 대부분의 운영체제는 실시간 태스크의 시간적 제약조건(timing constraints)을 효율적으로 지원할 수 있는 방법을 제공해 주지 못하고 있다. 실시간 응용의 시간적 제약조건을 지원하기 위해서는 운영체제 커널 변경 방법과 미들웨어 방법이 있다. 본 논문에서는 운영체제 변경없이 잘 알려진 Real-time Object Model인 TMO에 근거한 미들웨어 접근 방식을 적용한다. 현재 TMO(Time-triggered Message-triggered Object) 모델을 기반으로 한 미들웨어로 다양한 운영체제 시스템 상에서 개발되어온 TMOSM(TMO Support Middleware)이 있다. 리눅스 기반의 TMOSM의 스케쥴링 알고리즘은 효율적으로 실시간 스케쥴링을 지원하지만 주기적인 실시간 태스크를 위해 몇 가지 고려해야할 사항들이 있다. 본 논문에서는 주기적인 실시간 태스크를 효율적으로 처리할 수 있는 개선된 실시간 미들웨어 스케쥴링 알고리즘을 제안하고 성능을 비교한다. 제안한 알고리즘은 실시간 미들웨어의 구조를 간단하게 함으로써 시스템 성능 향상과 주기적인 실시간 태스크의 적시성을 더욱더 보장함을 확인하였다.


# A Study on the Scheduling Improvement for Periodic Real-time Tasks on Middleware based on Linux (TMOSM/Linux)

Ho-Joon Park[†] · Chang-Hoon Lee[††]


## ABSTRACT

For real-time applications, the underlying operating system (OS) should support timeliness guarantees of real-time tasks. However, most of current operating systems do not provide timely management facilities in an efficient way. There could be two approaches to support timely management facilities for real-time applications: (1) by modifying OS kernel and (2) by providing a middleware without modifying OS. In our approach, we adopted the middleware approach based on the TMO (Time-triggerred Message-triggered Object) model which is a well-known real-time object model. The middleware, named TMSOM (TMO Support Middleware) has been implemented on various OSes such as Linux and Windows XP/NT/98. In this paper, we mainly consider TMOSM implemented on Linux (TMOSM/Linux). Although the real-time scheduling algorithm used in current TMOSM/Linux can produce an efficient real-time schedule, it can be improved for periodic real-time tasks by considering several factors. In this paper, we discuss those factors and propose an improved real-time scheduling algorithm for periodic real-time tasks. In order to simulate the performance of our algorithm, we measure timeliness guarantee rate for periodic real-time tasks. The result shows that the performance of our algorithm is superior to that of existing algorithm. Additionally, the proposed algorithm can improve system performance by making the structure of real-time middleware simpler.


**키워드 : 실시간 응용(Real-time Application), 미들웨어(Middleware), 실시간 미들웨어 스케쥴링(Real-time Middleware Scheduling), TMO(Time-triggerred Message-triggered Object), TMOSM/Linux(TMO Support Middleware for Linux)**

## 1. 서 론

실시간 응용을 개발하기 위해 운영체제는 실시간 태스크의 시간 보장성(timeliness guarantee)이 이루어져야한다. 그러나 현재 대부분의 운영체제는 실시간 태스크의 시간적 제약조건(timing constraints)을 효율적으로 지원할 수 있는 방법을 제공해 주지 못하고 있다. 실시간 응용의 시간적 제약조건을 지원하기 위해 두 가지 접근 방법이 있을 수 있다. 첫 번째는 운영체제 커널을 변경하는 방법이 있고 두 번째는 운영체제 변경없이 미들웨어로 지원하는 방법이 있다. 첫 번째 방법은 운영체제 커널을 선점형 방식으로 변경하는 것으로 이 방법을 사용할 경우 운영체제 표준 서비스들 중 일부 서비스들은 동작 시 문제를 발생시킬 수 있다. 그러나 미들웨어 방법은 커널 변경 방법보다 실시간 태스크의 시간

※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성·지원사업의 연구결과로 수행되었음.
† 준 회 원 : 건국대학교 대학원 컴퓨터·정보통신공학과
†† 종신회원 : 건국대학교 컴퓨터공학과 교수
논문접수 : 2004년 10월 1일, 심사완료 : 2004년 10월 27일

적인 제약조건을 정확하게 만족시킬 수는 없지만 커널 변경 시 미들웨어는 영향을 받지 않기 때문에 커널과 독립적으로 동작이 가능하다. 따라서 본 논문에서는 실시간 태스크의 시간적 제약조건을 만족시키기 위해 다양한 기능들을 제공하는 미들웨어 방법을 적용한다[1].

실시간 객체 모델로써 잘 알려진 TMO(Time-triggerred Message-triggered Object) 모델을 기반으로 한 미들웨어로 TMOSM(TMO Support Middleware)이 있다. TMOSM 미들웨어는 리눅스와 윈도우 XP/NT/98과 같은 다양한 운영체제 시스템 상에서 개발되어왔으며[2-4] 실시간 태스크의 주기적인 실행, 입력/출력 메시지 핸들링, 데드라인 위반 관리와 같은 기능들을 제공한다. 이러한 기능들을 효율적으로 제공하기 위해 TMOSM은 실시간 미들웨어 스케쥴러와 메시지 핸들러를 지원한다. 실시간 미들웨어 스케쥴러는 하드웨어 타이머 인터럽트 핸들러에 의해 정의된 타임-슬라이스(time-slice) 마다 실행된다. 타임-슬라이스의 값이 작아질수록 또는 주기적인 실시간 태스크의 주기 시간이 작아질수록 실시간 미들웨어 스케쥴러는 더욱더 빈번하게 매 타임-슬라이스마다 실행된다. 이러한 스케쥴링 알고리즘은 CPU 자원 낭비와 시스템 오버헤드의 증가를 유발시킬 수 있다.

본 논문에서는 이러한 문제점을 해결하고 주기적인 실시간 태스크를 효율적으로 처리할 수 있는 개선된 실시간 미들웨어 스케쥴링 알고리즘을 제안한다. 시뮬레이션을 통해 기존의 스케쥴링 알고리즘과 제안한 스케쥴링 알고리즘의 비교 분석을 위해 실시간 태스크 수 및 주기 시간의 변화에 따라 주기적인 실시간 태스크의 시간성 보장률(Timeliness Guarantee Rate)로 성능을 비교 분석하였다.

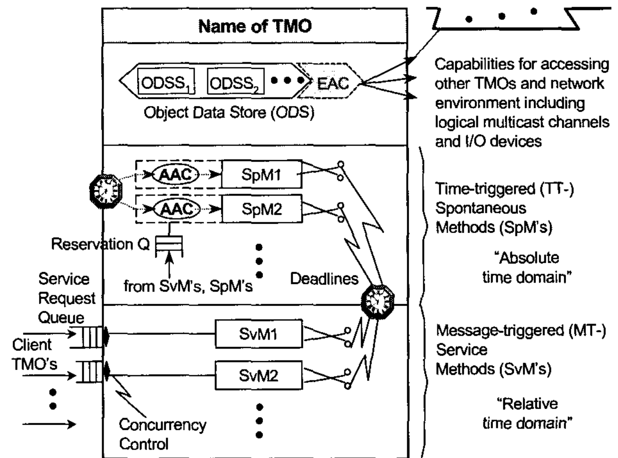
본 논문의 2장에서는 관련 연구로써 TMO 모델과 리눅스를 기반으로한 TMOSM에 대해 기술하고 3장에서는 실시간 미들웨어 설계 시 고려해야할 사항들에 대해 살펴본다. 4장에서는 제안한 실시간 미들웨어 구조와 스케쥴링 알고리즘을 기술하며 5장에서는 성능 분석을 보여준다. 마지막 6장에서는 결론을 맺는다.

## 2. 관련 연구

### 2.1 TMO

TMO는 자연스러우면서도, 문법상의 제약이 적고, 구문상으로 확장이 쉬운 양식화된 객체이다. 또한, 고수준의 실시간 컴퓨팅 모델이기도 하다. 시간에 관련된 기능들은 특별한 제한 없이 형식화된 폼에 의해 설정이 가능하다 [5-6].

(그림 1)에서 보인 바와 같이 기본적인 TMO의 기능들은 다음의 4가지로 구분된다.




(그림 1) TMO의 기본적인 구조[5]

① Spontaneous Methods(SpM) : 시간에 따라 작동되는 메소드(Time-triggered Method)로 실시간 클럭이 설계된 특정 시간에 도달하였을 때 실행된다. SpM은 AAC(Autonomous Activation Condition)에 의해 설계된 시간 모델을 할당받게 된다. 예로서, 하나의 SpM에 할당된 AAC가 "for t = from 11am to 11 : 40am every 20min start-during (t, t + 5min) finish-by t + 10min"이라고 설정되었을 때, 해당 SpM은 11시부터 매 20분간 5분내에 작동될 것이며, 10분내에 작동을 끝내고 이것을 11시 40분까지 작동하는 것을 의미한다.

② Service Methods(SvM) : 메시지에 따라 작동 되는 메소드(Message-triggered Method)로 실시간 시스템에 네트워크에 의한 메시지가 도달하였을 때 작동한다. 통상 SvM은 시작 시간이 메시지에 따라 결정되므로 AAC는 가지고 있지 않다.

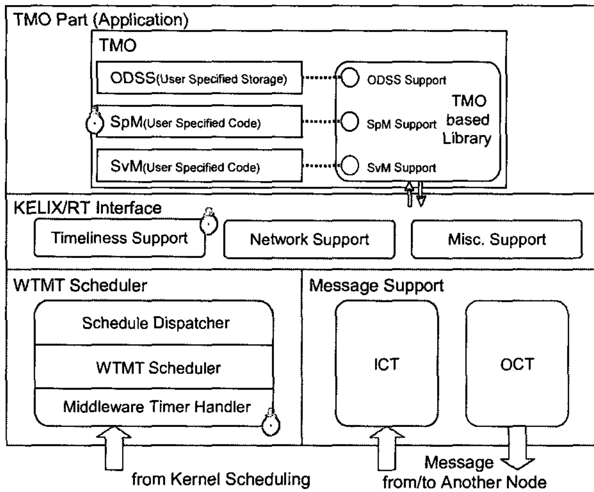
③ Object Data Store(ODS) : TMO내에 위치하여 기본적인 정보를 기억하는 장소이다. 이 정보는 TMO내의 SpM과 SvM이 공유하는 정보로서 동기화되어 관리된다.

④ Environment Access Capability(EAC) : TMO에 대한 기본적인 정보를 저장하는 장소로 네트워크 연결, I/O 장치의 사용, 논리적인 통신 수단 등에 대해 명세한다.

(그림 1)과 같이, 하나의 TMO에는 실행의 주체라 할 수 있는 SpM과 SvM이 공존하고, 같은 정보 영역인 ODS를 공유하기 때문에 정보 접근시 충돌 발생이 예상된다. 이러한 충돌을 피하기 위해서는 BCC(Basic Concurrency Constraint) 규칙이 적용된다. 이 규칙에 따르면 SvM은 ODS에 접근할 때, SpM의 실행을 방해해서는 않된다. SpM은 AAC에 명세된대로 자신이 실행되고, 실행되어야 할 시간을 보장받는 메소드이다. 따라서 같은 ODS를 사용하는 SvM은 이 시간에 접근할 수 없다.

### 2.2 TMO Support Middleware(TMOSM/Linux)

(그림 2)는 TMOSM/Linux의 내부 구조이다. TMOSM/Linux는 크게 응용프로그램 태스크와, 미들웨어 태스크로 구분된다.



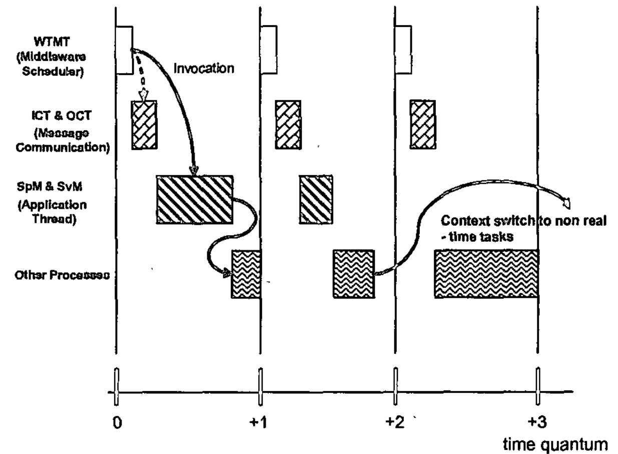

(그림 2) TMOSM/Linux의 기본적인 구조

응용프로그램 태스크는 SpM이나 SvM들로서 TMOSM/Linux에 의해 할당되고 실행된다. 미들웨어 태스크는 주기적인 태스크로서 TMOSM/Linux를 동작시키는 역할을 담당한다. 또한 미들웨어 태스크는 시스템 타이머에 의해 작동되며, 하나의 시스템 내에서 최우선 순위를 보장받아 매 스케쥴링이 발생될 때 마다 먼저 실행된다. 미들웨어 태스크는 다시 WTMT(Watchdog Timer Management Task), ICT (Incoming Communication Task), OCT(Outgoing Communication Task)로 구분되고 그들의 역할은 다음과 같다.

① WTMT : 이 태스크는 운영체제에 의해 타이머로써 동작되는 주기적인 태스크다. 주 임무는 응용프로그램 태스크(SpM이나 SvM)의 스케쥴링을 담당한다. 또한, WTMT는 응용프로그램 태스크가 사용하는 타이머를 갱신하는 역할을 수행하고 데드라인을 처리하는 역할을 수행한다. 따라서 WTMT는 TMOSM/Linux의 핵심 태스크다.

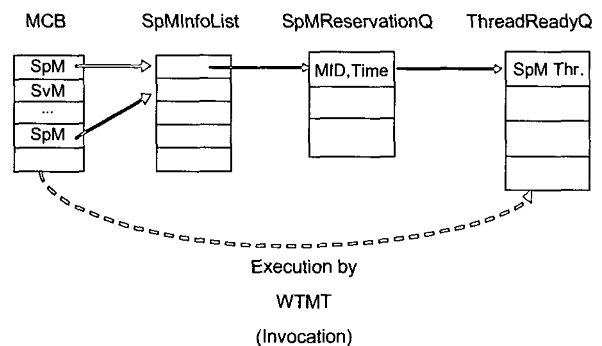
② ICT & OCT : 이 태스크는 TMOSM/Linux에 메시지를 전달하는 임무를 가진다. ICT는외부의 메시지를 입력받는 장소이며, OCT는 내부의 메시지를 외부로 보내는 역할을 수행한다. 통상 메시지는 네트워크를 통한 UDP 소켓을 사용하여 구현된다. 특히, ICT는 시스템 외부나 자체에서 메시지가 도달하면 해당 메시지에 관련된 SvM을 찾아내어 WTMT로부터 스케쥴링을 요청하는 기능을 담당한다.

(그림 3)은 미들웨어 태스크와 응용프로그램 태스크들간의 타이밍 챠트(Timing-chart)를 표현한 것이다. 각각의 미들웨어 태스크의 우선순위는 먼저 WTMT가 가장 높고 그 다음이 ICT와 OCT로 정해진다. 미들웨어 태스크가 더 이상 해야 할 임무가 없을 경우, 응용프로그램 태스크(SpM, SvM)들이 실행된다. 더 이상 실행될 응용프로그램 태스크가 없다면 마지막으로 시스템내의 실시간 시스템이 아닌 다른 태스크가 실행된다.




(그림 3) 미들웨어 태스크의 실행 우선 순위와 타이밍 챠트

시스템 내의 SpM, SvM들은 기본적으로 유사한 메소드들이지만, 특히 TMOSM/Linux에서 SpM은 WTMT에 의해 주기적으로 실행된다는 것이 다르다. (그림 4)는 이러한 SpM을 작동하기 위해 WTMT가 SpM을 관리하는 내부구조를 보여준다.



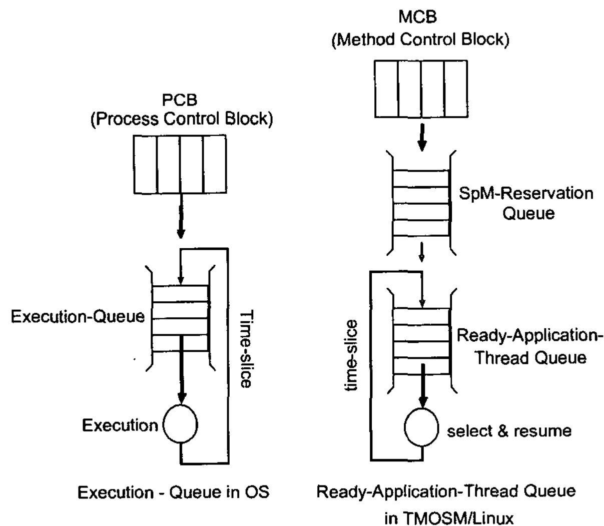

(그림 4) WTMT내에서의 SpM 처리 과정

WTMT는 주기적으로 SpM의 MCB(Method Control Block, AAC 및 SpM의 각종 정보들이 저장되는 장소)를 관찰하여 실행 시간이 임박한 SpM을 인출하여 SpM-Reservation Queue에 삽입한다. 이렇게 삽입된 SpM은 후에 다시 한번 WTMT에 의해 확인되어 Ready-Application-Thread Queue에 배치되며, 이와 수반되어 WTMT는 다음 번 주기(time-slice)에 SpM을 실행하게 된다.

## 3. TMOSM/Linux에서의 실시간 미들웨어 스케쥴링 알고리즘

본 절에서는 주기적인 실시간 태스크인 SpM를 구동시키는 TMOSM의 WTMT 스케쥴링 알고리즘 설계 시 고려사항에 대해 살펴본다.

① (그림 5)는 범용 운영체제와 리눅스 기반의 TMOSM에서 태스크 구동을 위한 내부 구조를 나타낸 것이다[8]. 리눅스 또는 윈도우 운영체제의 스케쥴러는 태스크 구동을 위해 Execution-Queue를 관리한다. 운영체제의 스케쥴러는 매 타임-슬라이스 동안 Execution-Queue에 있는 "runnable"한 상태의 태스크를 구동시킨다. 단일 태스크를 스케쥴하는 DOS 상에서 실시간 미들웨어를 개발하기 위해서는 실시간 미들웨어 상에서 큐 지원이 필요하다[7]. 그러나 리눅스 또는 윈도우 XP/NT/98과 같은 현재 대부분의 범용 운영체제는 멀티 태스크 스케쥴 지원이 가능하기 때문에 실시간 미들웨어 상에서 큐 지원이 필요 없다. 따라서 리눅스 기반의 TMOSM이 SpM-Reservation-Queue와 Ready-Application-Thread-Queue를 핸들링하는 것은 CPU 자원 낭비를 초래할 뿐만 아니라 시스템 오버헤드를 증가시킨다.




(그림 5) 태스크 구동을 위한 OS와 리눅스 기반의 TMOSM 내부 구조

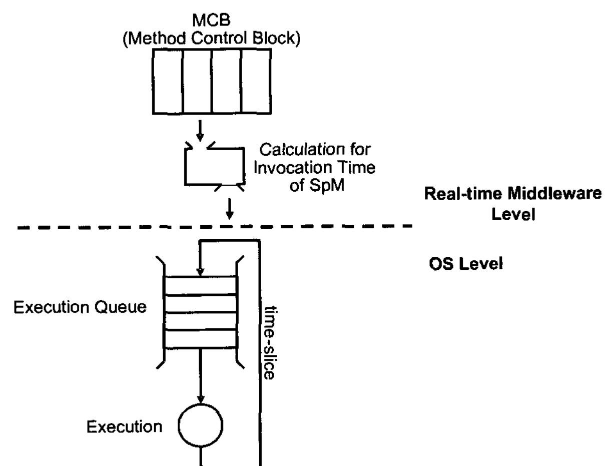
② TMOSM의 WTMT 스케쥴러는 매 타임-슬라이스마다 SpM 구동 여부를 결정하기 때문에 실시간 응용의 시간성 보장은 타임-슬라이스 값에 의해 좌우된다. 예를 들어, SpM의 주기가 1초이고 타임-슬라이스가 1msec인 경우, WTMT 스케쥴러는 SpM 구동 여부를 결정하기 위해 1초 동안 1,000번 실행된다. 이럴 경우, WTMT 스케쥴러는 SpM을 한번 구동시키기 위해 999번의 불필요한 실행이 발생된다. 그러므로 위에서 언급하였듯이, 이러한 스케쥴링 알고리즘은 CPU 자원 낭비를 초래하며 시스템 오버헤드를 증가시킨다.

③ TMOSM의 SpM큐 구조가 배열(Array) 구조로 되어 있기 때문에 SpM 수가 적은 경우에는 큐 탐색 시간이 문제가 되지 않지만 시스템 오버헤드 증가 또는 SpM 수가 증가할 경우 탐색 시간도 증가하게 된다. 따라서 실시간 미들웨어 상에서의 큐 지원은 성능 저하를 초래할 수 있다[9].

## 4. 제안한 실시간 미들웨어 스케쥴링 알고리즘

### 4.1 제안한 구조

리눅스 기반의 TMOSM에서 WTMT 스케쥴링 알고리즘를 보완하기 위해 개선된 실시간 미들웨어 스케쥴링 알고리즘을 제안한다. (그림 6)은 WTMT 스케쥴러에 의해 SpM이 구동되는 내부 구조를 보여준다. WTMT 스케쥴러는 본 논문에서 제안한 스케쥴링 알고리즘에 의해 MCB에 등록되어 있는 SpM의 구동여부를 결정한다. 구동될 SpM은 Execution-Queue에 배치되고 운영체제 스케쥴러는 매 타임-슬라이스마다 스케쥴링 정책에 따라 Execution-Queue에 있는 태스크를 스케쥴링한다. 제안한 구조는 실시간 미들웨어에서 SpM-Reservation-Queue와 Ready-Application-Thread-Queue를 사용하지 않기 때문에 미들웨어 구조를 간단하게 한다.




(그림 6) SpM의 실행을 위해 제안한 내부 구조

### 4.2 제안한 스케쥴링 알고리즘

매 타임-슬라이스마다 MCB에 등록되어 있는 SpM의 구동 여부를 결정하기 위해 본 논문에서는 최대공약수 방법을

적용한다. 아래 수식은 SpM 주기와 최대공약수를 이용하여 SpM의 구동 주기를 구하는 방법을 보여준다. 예를 들면, 타임-슬라이스 값이 1msec이고 각 SpM의 주기가 3초, 5초인 경우, 3초의 주기를 갖는 SpM은 3,000타임-슬라이스, 5초 주기를 갖는 SpM은 5,000 타임-슬라이스로 나타낼 수 있다. 본 논문에서는 3,000과 5,000를 변환 주기로 명명한다. 3,000과 5,000의 최대공약수는 1,000이 되며 이 값은 1,000 타임-슬라이스로 볼 수 있다. $T_{GCD_period}$값은 WTMT 스케쥴러가 SpM의 구동여부를 결정하기 위해 MCB를 탐색하는 주기 시간이 된다. 즉, WTMT 스케쥴러는 매번 1타임-슬라이스마다 MCB에 등록되어 있는 SpM의 구동 여부를 결정하는 것이 아니라 1,000타임-슬라이스 마다 SpM의 구동 여부를 결정하게 된다. 따라서 빈번하게 발생되는 MCB 큐의 검색 횟수를 줄여준다. 새로운 SpM이 생성되거나 SpM의 주기가 갱신되면 $T_{GCD_period}$값이 새로 계산된다. 이러한 스킴은 본 논문에서 제안한 실시간 미들웨어 스케쥴링 알고리즘이 적용된다

$$t^{1}_{변환주기} = S_pM_1의\ 주기 / 타임-슬라이스$$
$$t^{2}_{변환주기} = S_pM_2의\ 주기 / 타임-슬라이스$$
$$\cdots$$
$$t^{N}_{변환주기} = S_pM_N의\ 주기 / 타임-슬라이스$$

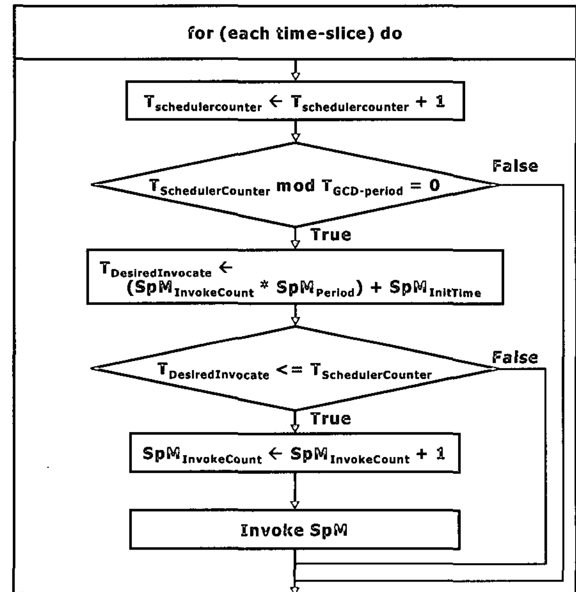
◦ 초기 실행인 경우,

$$T_{GCD_주기} = G.C.D\,(t^{1}_{변환주기}, t^{2}_{변환주기}, \cdots, t^{N}_{변환주기})$$

◦ 새로운 SpM 생성 및 SpM 주기 갱신인 경우,

$$t^{new}_{변환주기} = S_pM_{new}의\ 주기 / 타임-슬라이스$$
$$T_{GCD_주기} = G.C.D.\,(T_{GCD_주기}, t^{new}_{변환주기})$$

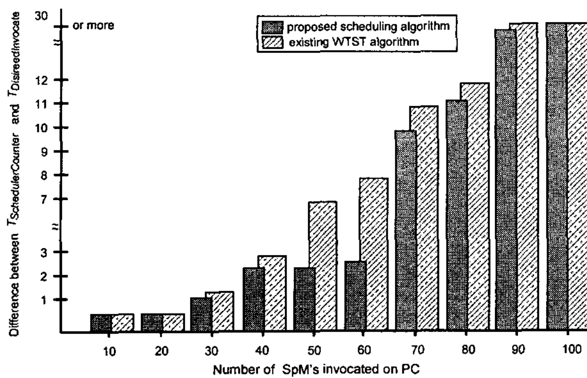
(그림 7)은 제안한 실시간 미들웨어 스케쥴링 알고리즘을 보여준다. $T_{SchedulerCounter}$은 타임-슬라이스가 발생할 때마다 WTMT 스케쥴러에 의해 값이 증가된다(1~2). WTMT 스케쥴러는 $T_{SchedulerCounter}$값과 $T_{GCD_period}$값을 이용하여 MCB 큐를 검색할지를 결정한다(3). ($T_{SchedulerCounter}$ mod $T_{GCD_period}$)값이 0이면 SpM이 구동될 시간인 $T_{DesiredInvocation}$을 구한다(4). 만약 $T_{SchedulerCounter}$값이 $T_{DesiredInvocation}$값보다 크면 SpM이 구동된다(5~6). SpM의 다음 주기가 발생할 때 SpM이 구동될 수 있도록 $SpM_{InvokeCount}$값을 증가시킨다(7). 제안한 구조에서는 SpM-Reservation-Queue와 Ready-Application-Thread-Queue가 없기 때문에 SpM이 구동되면 WTMT 스케쥴러는 SpM를 Execution-Queue에 배치한다. 제안한 스케쥴링 알고리즘은 WTMT 스케쥴러가 MCB 큐의 검색 횟수를 줄여줌으로써 CPU 자원 낭비 및 시스템 오버헤드 문제를 해결해준다.

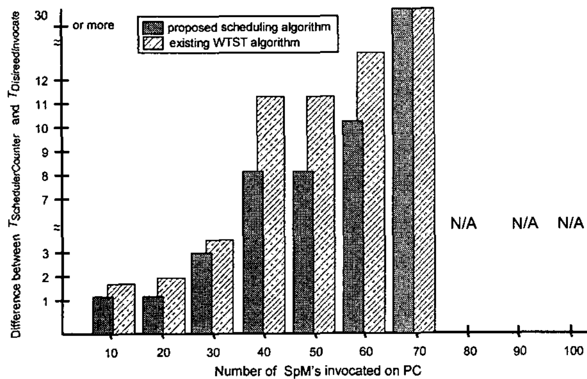



(그림 7) 제안한 실시간 미들웨어 스케쥴링 알고리즘

## 5. 실험 및 성능 평가

제안한 스케쥴링 알고리즘의 성능 분석을 위해 사용된 시뮬레이션 환경은 다음과 같다. 시뮬레이션은 하드웨어상의 성능 차이가 뚜렷한 x86 리눅스 시스템의 펜티엄4(1.5GHz) PC와 임베디드 시스템인 StrongArm CPU(206MHz)를 사용한 리눅스가 탑재된 PDA에서 하였다. SpM의 수는 10에서 100까지 10씩 증가하며 SpM의 주기는 1초에서 5초 사이의 랜덤 값을 사용하였다. 각각의 SpM은 500msec의 무한루프로 구성된다. (그림 8)은 PC와 PDA에서 시뮬레이션 결과를 보여준다. 제안한 스케쥴링 알고리즘과 기존의 알고리즘에서 WTMT 스케쥴러가 SpM를 구동할 때마다 $T_{SchedulerCounter}$와 $T_{DesiredInvocate}$의 차이를 계산하였다. (그림 8)에서처럼 제안한 스케쥴링 알고리즘이 리눅스 기반의 TMOSM 스케쥴링 알고리즘보다 성능이 나음을 보여준다. 시스템 오버헤드가 없는 경우 차이는 0 또는 1이 되며 이것은 SpM의 적시성 보장이 정확하게 지원됨을 의미한다. 그러나 SpM의 수가 증가할 경우, 차이는 급격히 증가하게 되며 시스템의 오버헤드로 인해 WTMT 스케쥴러는 SpM의 적시성를 보장해주지 못한다. 특히 SpM의 수가 90또는 100인 경우, 값의 차이가 매우 크거나 또는 계산할 수 없었다. 그 이유는 PC 또는 PDA의 성능이 SpM의 수가 많은 경우 처리를 하지 못하기 때문이다. 따라서 이러한 값의 차이가 증가할수록 SpM의 적시성 보장은 힘들게 된다.




(그림 8) 시뮬레이션 결과

## 6. 결 론

본 논문에서는 실시간 미들웨어 스케쥴링 알고리즘 설계 시 고려해야할 문제점들에 대해 살펴보았으며 향상된 실시간 미들웨어 스케쥴링 알고리즘을 제안하였다. 제안한 실시간 미들웨어 구조는 큐를 사용하지 않는 단순한 구조로써 실시간 태스크의 주기와 최대공약수를 이용하여 주기적인 실시간 태스크의 구동 여부를 효과적으로 결정할 수 있는 스킴을 제안하였다. 또한 제안한 스케쥴링 알고리즘과 기존의 스케쥴링 알고리즘을 비교 분석하였다. 주기적인 실시간 태스크 수가 작은 경우에는 두 알고리즘의 성능 차이가 크지 않았다. 그러나 주기적인 실시간 태스크 수가 증가할수록 제안한 스케쥴링 알고리즘의 성능이 나음을 확인하였다.

## 참 고 문 헌

[1] Park, H. J. and Lee, C. H, "Deadline Handling in a Real-time Middleware on LINUX," IDPT, pp.648-651, 2003.

[2] Kim, K. H., Ishida, M. and Liu, J., "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation," Proc. ISORC'99 (IEEE CS 2nd Int'l Symp. On Object-oriented Real-time distributed Computing), pp.54-63, 1999.

[3] Kim, J. G., Kim, M. H., Min, B. J. and Im, D. B., "A soft Real-Time TMO Platform-WTMOS-and Implementation Techniques," Proc. ISORC'98, Kyoto, Japan, 1998.

[4] Kim, M. H. and Kim, J. G., "Linux based TMO execution platform for embedded applications," presented at UKC 2004, (proceedings will be published in Oct., 2004).

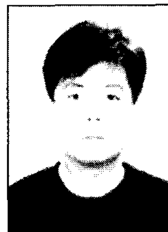
[5] Kim, K. H., "APIs for Real-Time Distributed Object Programming," IEEE Computer, Vol.33, No.6, pp.72-80, 2000.

[6] Kim. K. H., "Real-Time Object-Oriented Distributed Software Engineering and the TMO Scheme," Int'l Jour. Of Software Engineering & Knowledge Engineering, No.2, pp.251-276, 1999.

[7] Kim, K. H. (Kane) and Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potential," Proc. COMPSAC'94 (IEEE Computer Society's 1994 Int'l Computer Software & Applications Conf.), Taipei, pp.392-402, 1994.

[8] Robbins, K. A. and Steven, "Practical UNIX Programming," Prentice Hall, 1996.

[9] Kim, K. H., Subbaraman, C. and Kim, Y., "The DREAM Library Support for PCD and RTO.k programming in C++," Proc. WORDS'96 (IEEE Computer Society 2nd Workshop on Object-oriented Real-Time Dependable Systems), Laguna Beach, pp.59-68, 1996.



**박 호 준**

e-mail : hjpark@konkuk.ac.kr
1996년 경원대학교 물리학과(학사)
1998년 건국대학교 전자계산학과(석사)
1998년~현재 건국대학교 컴퓨터·정보통신공학과 박사과정
관심분야 : Embedded System, Real-time Middleware

**이 창 훈**

e-mail : chlee@konkuk.ac.kr
1980년 연세대학교 수학과
1977년 한국과학기술원 전산학과 석사
1993년 한국과학기술원 전산학과 박사
1996년~2000년 건국대학교 서울캠퍼스 정보통신원 원장
2000년~2002년 건국대학교 정보통신대학원 원장
2001년~2002년 건국대학교 정보통신대학 학장
1980년~현재 건국대학교 컴퓨터공학과 교수
관심분야 : 지능시스템, 운영체제, 보안, 전자상거래 등