소 특 집

분산 실시간 객체를 지원하는 리눅스 기반 운영체제 TMO-Linux

김 정 국

한국외국어대학교 컴퓨터및정보통신공학부

I. 서 론

실시간 시스템은 응용과 운영체제의 복합체로서 일반적으로 두 가지 의미로 해석할 수 있다. 첫째는, 시스템의 빠른 응답에 초점을 두는 경우이고, 둘째는 시스템 설계 시의 시간조건(timing constraints)에 대한 보장성 및 예측성을 강조하는 경우이다. 전자의 경우는 흔히 RTOS (Real-Time Operating System)라 불리는 내장형 운영체제와 그 응용들이 취하는 접근 기법으로, 소규모의 자원 환경에서의 빠른 인터럽트 처리 및 시스템 응답을 주요 기능으로 한다. 즉 이 때의 실시간 이라는 의미는 멀티 태스킹, 선점 스케줄링, 빠른 인터럽트 및 시스템 반응 지연 등의 의미로 사용된다. 그러나 진정한 실시간 컴퓨팅의 의미는 후자의 경우, 즉, 시간 조건에 대한 보장성 컴퓨팅을 제공하는 것이라 할 수 있다. 하지만 이러한 보장성 컴퓨팅을 제공하는 것은 동일한 운영체제 및 응용이라 할지라도 하드웨어 조건과 응용 프로그램의 여러 수행 환경에 따라 달라지기 때문에 매우 어렵고, 특히 실시간 운영체제의 기능 만으로 그 목적을 달성하는 것은 불가능한 것이 사실이다. 이러한 사실 때문에 보장성 실시간 컴퓨팅은 실시간 시스템 명세 및 설계 기법, 실시간 프로그래밍 패러다임, 실시간 운영체제, 하드웨어 및 수행 환경 등의 모든 수직적 환경에 대해 전반적으로 실시간적 기능 및 분석이 주어져야 하는 분야이다. 무기 제어와 같은 내장형 응용이 그 대표적인 경우로, 실시간 프로세스에 대해 일반 프로세스보다 상대적으로 빠른 서비스를 제공하는 개념의 범용 운영체제와는 달리, 수행 환경의 동적 변화가 거의 없는 환경에서 보장성 실시간 컴퓨팅을 지향하는 경우이다.

본 기고에서는 이와 같이 특수 목적의 내장형 환경에서 보장성 실시간 컴퓨팅을 지향하는 시스템이 설계 시부터 활용할 수 있는 실시간 객체 패러다임의 하나인 실시간 객체 모델 TMO[1-3]와 이의 실행을 지원하는 운영체제인 TMO-Linux[6]에 대해 소개한다. 현재의 TMO-Linux는 상용 RTOS처럼 인터럽트나 시스템 호출에 대한 반응 시간을 최소화 하는 것보다는, 새로운 실시간 프로그래밍 패러다임인 TMO의 지원에 초점을 맞추고, 내장형 시스템이나 일반 서비스 시스템에 사용될 수 있도록 Linux 커널을 기반으로 개발된 것이다. 그러나 앞으로는 RTOS의 경우처럼 여러 가지의 반응 시간 최소화나 선점 가능한 커널 개념도 반영될 것으로 기대된다.

II. 실시간 응용과 실시간 운영체제의 관계

본 기고에서 초점을 맞추고자 하는 실시간 시스템의 특성은 다음과 같이 요약될 수 있다.

- 주기적(periodic) 태스크 및 산발적(sporadic) 태스크에 대한 시간 조건의 적용
- 시간 조건에 대한 보장성(timeliness guaranteed) 컴퓨팅
- 예측성(predictability)

위에서 시간 조건이라 함은 구동 시간, 완료시

간 등으로 볼 수 있고 예측성은 서비스 시간에 대한 사전 분석, 자원의 사전할당 등을 통해 가능한 설계 시의 시간 보장의 개념이다. 이러한 실시간 컴퓨팅의 목표로 가기 위해 중요한 사실은 운영체제와 실시간 응용간의 잘 정의된 상호 협조적 역할이 반드시 있어야 한다는 점이다. 즉 응용 설계 및 작성 수준에서는 실시간 프로세스나 스레드의 실행에 대한 형태 및 시간 조건에 대한 규격적 정의가 있어야 하고, 운영체제는 이와 같은 실시간 실행 정의에 대한 보장성 컴퓨팅을 제공해야 한다. 이와 같은 의미에 대한 예를 들면, 운영체제는 어떤 실시간 서버 프로세스가 수행하는 한번의 서비스에 대해 3초의 서비스 시간을 보장할 때, 클라이언트는 이와 같은 서비스에 대한 요구를 3초 보다 빠른 간격으로 발생시켜서는 보장성 실시간 시스템이 될 수 없다는 사실이다. 특히 경성 실시간 시스템에서는 이와 같이 서비스를 요구하는 클라이언트와 서버의 시간 보장적 개념이 실시간 서버 프로세스와 클라이언트 프로세스 간, 프로세스와 운영체제 간, 외부 장치와 인터럽트 서비스 간에 모두 정의되어야 할 것이다. 이러한 모든 서비스에 대한 운영체제의 보장은 프로세스의 수나 외부 장치의 수에 따라 달라질 것이므로 일반적으로는 불가능하나, 내장형 시스템과 같이 자원이나 환경에 제한되는 시스템에서는 각 경우에 대한 사전 분석과 실험을 통해 예측성이 주어질 수 있을 것이다. 즉 운영체제의 인터럽트 처리나 시스템 호출에 대한 시간 보장은 각 주어진 실행 환경에 따라 달라지므로 이에 대한 분석은 타겟 실시간 시스템 설계자의 몫으로 남는 것이 보통이다. 실시간 응용은 이와 같은 운영체제 서비스에 대한 분석 위에서 작성되어야 하는데, 응용 내부적으로도 여러 형태의 실행 요소간에 시간 보장의 개념이 도입되어 설계되어야 한다. 이러한 응용의 실행 요소란 객체, 서버 및 클라이언트, 호출자와 피호출자, 프로세스와 스레드 등 여러 형태일 수가 있고, 이러한 실행 요소의 형태 및 시간 조건에 대한 규격을 제공하는 것이 실시간 프로그래밍 패러다임이라 할 수 있다.

TMO(Time-triggered Message-triggered Object)는 이러한 실시간 프로그래밍 패러다임의 하나로, 분산 실시간 객체 모델의 일종이다. 본 기고에서는 TMO 모델의 개념과 이의 실행을 제공하는 운영체제인 TMO-Linux에 대해 기술한다.

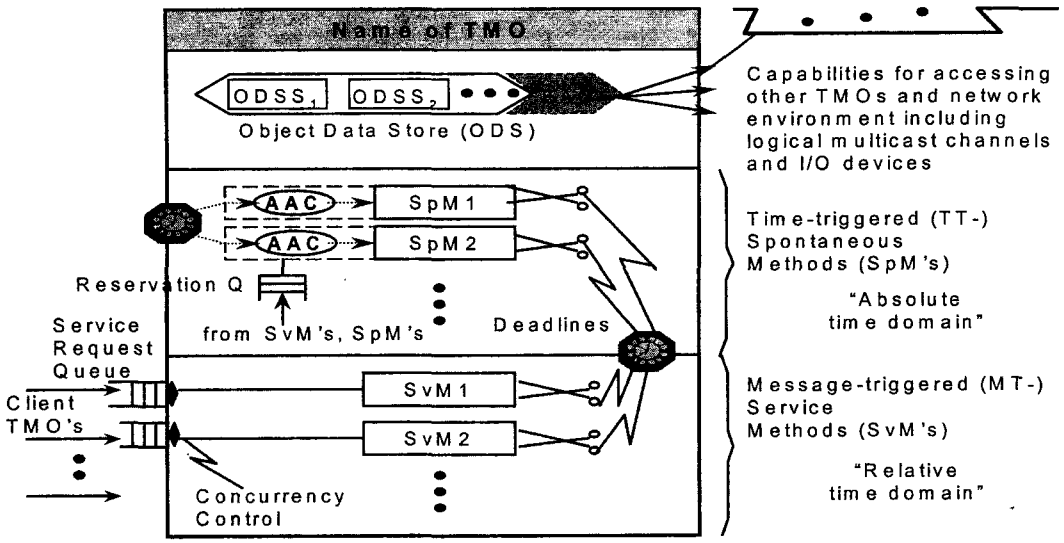
III. 분산 실시간 객체 모델 TMO

분산 실시간 객체 모델 TMO[UCI, Kan Kim]는 실시간 시스템의 시간 보장성 컴퓨팅을 지원하기 위한 설계 패러다임으로 제안된 것이다. TMO 모델이 표방하는 목적은 다음과 같다.

- 경성 및 연성 실시간 시스템에서부터 일반 병행 프로그램 까지 활용할 수 있는 유연한 구조
- 시스템 설계 단계에서의 시간 보장을 고려하는 모델
- 실시간 컴퓨팅, 객체 지향 컴퓨팅, 분산 컴퓨팅, 멀티 스레딩의 결합

TMO의 구조는 <그림 1>에 표시된 바와 같이 일반 객체와는 그 구조에 있어서 몇 가지 다른 점을 가진다. TMO는 크게 ODS(Object Data Store), 시간 구동 메소드(Time-triggered Method; Spontaneous Method(SpM)) 그룹, 그리고 메시지 구동 메소드(Message-triggered Method ; Service Method(SvM)) 그룹의 세 부분으로 구성된다. 다음은 TMO의 특징에 대한 설명이다.

- TMO를 구성하는 한 형태의 멤버인 시간 구동 메소드(이하 SpM)는 객체 내의 동적 메소드로, 객체의 멤버의 특성을 갖는 스레드로 구현되어 외부 호출이 아닌 시간 조건에 의해 자율적으로 구동 된다. SpM에는 시작과 종료 시간, 주기, 각 주기 구동시의 데드라인으로 구성되는 시간 조건(AAC : Autonomous Activation Condition)이 주어지고, 운영체제나 TMO 엔진은 이에 대한 데드라인 기반 스케줄링을 제공하여야 한다.



〈그림 1〉 TMO의 구조[1]

- TMO를 구성하는 또 다른 한 형태의 멤버인 메시지 구동 메소드(이하 SvM)는 다른 TMO 메소드로부터의 IPC 메시지 수신에 의해 구동되어, 메시지에 대한 서비스 완료 시까지의 데드라인이 적용된다. SvM 역시 객체의 멤버의 특성을 갖는 스레드로 구현되며, 운영체제나 TMO 엔진은 이에 대한 데드라인 스케줄링을 제공하여야 한다. SvM의 데드라인은 스케줄링에 반영되는 의미와 함께, SvM의 클라이언트가 데드라인 보다 더 작은 빈도로 메시지를 보내면 시간 내의 서비스가 보장되지 않음을 의미한다.
- SvM을 구동시키는 IPC 메시지는 분산 환경에도 변화 없이 그대로 적용되는, 네트워크에 투명한 분산 IPC이다. 따라서 TMO들의 네트워크로 표현되는 실시간 시스템은 그 구성의 변화 없이 그대로 단일 노드 컴퓨팅이나 분산 컴퓨팅에 적용된다.
- SpM의 모든 실행은 설계 시 이미 알려지는 것이므로 보장성 컴퓨팅을 목적으로 하는 시스템에서는 그 수행이 산발적인 SvM에 비해 실행의 우선 순위를 갖는다. 즉 SvM의 실행이 SpM의 실행을 방해하지 못한다. 이를 BBC (Basic Concurrency Constraint)라 한다.

Ⅳ. TMO 실행 엔진

분산 환경에서 실시간 객체 TMO의 네트워크 모델을 실행하기 위한 TMO 엔진은 '95년부터 미국 UCI의 DREAM Lab.과 국내 외대의 RTDCS Lab.에서 개발되어 왔는데, 이들 엔진은 크게 미들웨어 형태로 구현된 것과 운영체제 커널의 형태로 구현된 것의 두 가지로 구분된다. 다음은 이들 엔진에 대한 요약이다.

- DREAM kernel : UC Irvine의 DREAM Lab.에서 '95년 개발된 최초의 Stand-alone TMO 엔진[2]으로 MS-DOS를 수정한 커널 형태로 개발되었다. 산업용 제어 시스템의 구현에 적합하다.
- WTMOS(Windows TMO System)[4], TMO-SM(TMO Supprot Middleware) : '98년 외대 RTDCS Lab.과 '99년 UCI DREAM Lab.에서 각각 개발된 윈도우즈 계열 운영체제 상의 TMO 수행을 위한 분산 환경용 미들웨어 엔진이다. TMO의 분산 수행과 함께 기존 윈도우즈 운영체제의 API나 GUI를 그대로 사용할 수 있는 연성 실시간 시스템을 위한 플

랫폼이다.

- LTMOS(Linux TMO System)[5] : '00년 외대 RTDCS Lab.에서 개발된 리눅스 계열 운영체제상의 TMO 수행을 위한 분산 환경용 미들웨어 엔진이다. 미들웨어 엔진이므로 이식성이 좋고, TMO 프로그래밍과 함께 리눅스에서 제공하는 다른 모든 기능도 사용할 수 있다.
- TMO-Linux[6] : '02년 외대 RTDCS Lab.에서 개발된 TMO 실행을 위한 분산 리눅스 운영체제이다. 리눅스의 모든 기능에 분산 TMO의 수행을 위한 시스템 호출들과 스케줄러 부분이 삽입되었다.

다음은 이러한 엔진들이 TMO의 실행 지원을 위해 공통적 가지는 특성에 대한 요약이다.

- C++ 기반의 TMO 프로그래밍
- 1/1000초 단위의 데드라인 기반 스케줄링
- 분산 클럭 동기화
- 멀티 스레드에 의한 SpM 및 SvM의 구현
- 네트워크에 투명한 분산 IPC의 제공
- 분산 IPC에 의한 SvM의 구동 및 시간 조건에 의한 SpM의 구동
- Lock이나 CREW(Concurrent-Read Exclusive-Write) Monitor 등의 동기화 도구의 제공

미들웨어로 구현된 엔진은 이식성이 좋은 반면 스케줄링의 정확도가 떨어지고, 커널로 구현된 엔진은 아직 여러 환경에의 이식성에 취약점을 가지지만 데드라인 기반 스케줄링의 정확도가 높은 장점이 있다. 본 기고에서는 이러한 여러 엔진 중 가장 최근에 개발되어 현재 여러 플랫폼에서 안정화 작업이 진행 중인 TMO 실행 지원 운영체제인 TMO-Linux의 구현 기법과 기능에 대해 소개하기로 한다.

V. TMO-Linux의 설계와 구현

TMO-Linux는 전술한 바와 같이 분산 환경에서 협동 실행하는 TMO들을 지원하는 리눅스 기반 운영체제이다. TMO-Linux에 의한 분산 실시간 객체 TMO의 실행 지원은 커널의 "분산 실시간 프로세스 관리 부분"과 "TMO 매크로 및 객체 라이브러리"를 통해 이루어진다.

TMO-Linux의 분산 실시간 프로세스 관리 부분은 TMO의 실행 제공을 위한 엔진의 기저 기능을 제공하지만, TMO 모델을 사용하지 않는 일반 실시간 시스템의 작성에도 활용될 수 있다. 리눅스의 커널 내부에 개발 삽입된 "분산 실시간 프로세스 관리 기능"들의 구성은 다음과 같다.

- 주기적 실시간 프로세스
- 분산 IPC와 실시간 서버 프로세스 인터페이스
- 데드라인 기반 실시간 스케줄러

TMO 매크로 및 객체 라이브러리 부분은 실시간 객체와 그 내부의 동적 스레드인 SpM과 SvM을 TMO-Linux 커널이 제공하는 주기적 실시간 프로세스나 실시간 서버 프로세스로 변환하는 기능을 제공한다.

1. 주기적 실시간 프로세스 인터페이스

주기적 실시간 프로세스는 이미 생성된 리눅스 운영체제의 실시간 프로세스가 커널에 주기, 실행 데드라인 등의 시간 조건을 등록함으로써 생성되고 관리된다. 이러한 등록 및 주기적 실시간 프로세스의 실행을 위한 새로운 커널 API들은 다음과 같다.

- Rt_PP_Register(period, deadline, start_time, stop_time) : 이 시스템 호출은 이미 생성된 실시간 프로세스를 주기적 실시간 프로세스로 등록하는 데 사용된다. 매개변수 중 deadline은 매 주기의 실행에 대해 실행 완료 데드라인을 의미하고, start_time 및 stop_time은 주기적 실시간 프로세스로의 전체적 활동 시간을 의미한다. 주기적 실시간 프로세스로 등록이 되면 커널은 이 프로세스를 관리하기 위한 정보 블록(MCB : Method Control Block)을 생성하여 원래의 리눅스 프로세스를 위한 task 구조체에 연결시키게 된다.

MCB의 주요 내용은 동적인 시간 조건에 관한 것들이다.

- Rt_PP_Wait_Invocation() : 주기적 실시간 프로세스로 등록을 마친 프로세스가 주기의 도래에 의한 구동을 기다리는 커널 API이다. 따라서 SpM에 주어지는 데드라인은 Rt_PP_Wait_Invocation()에서 깨어난 후, 다음 Rt_PP_Wait_Invocation() 실행까지의 최대 허용 시간이 된다. 반환 값 TRUE는 정상적 주기의 도래에 의한 구동을 의미한다.
- Rt_PP_Exit : 주기적 실시간 프로세스가 일반 실시간 프로세스로 환원된다. 이에 따라 커널 내부에 할당되었던 MCB 등의 정보 블록도 해제된다.

이와 같은 시스템 호출 들을 이용한 주기적 실시간 프로세스의 실행 형태는 다음과 같다.

```
Rt_PP_Register(period, deadline,,) ;
while (Rt_PP_Wait_Invocation( ) = = TRUE)
{
    do a periodic job
        within the given deadline ;
}
Rt_PP_Exit() ;
```

2. 실시간 서버 프로세스와 분산 IPC 인터페이스

TMO-Linux는 분산 IPC와 연계된 실시간 서버 프로세스 인터페이스를 제공한다. 여기서 실시간 서버 프로세스란 분산 IPC 메시지의 수신에 의해 구동되어, 서비스 완료 시까지 데드라인 기반 스케줄링의 적용을 받는 새로운 형태의 프로세스를 말한다. 실시간 서버 프로세스는 일반적 실시간 프로세스가 서비스 데드라인을 등록함으로써 생성된다. 실시간 서버 프로세스는 자신이 사용할 분산 IPC 채널을 할당하고 이 채널을 사용하여 message-triggered 방식의 실행을 하게 된다.

TMO_Linux의 분산 IPC는 LAN 서브넷 환경에서 브로드캐스팅 방식으로 지원되는 것으로 네트워크에 대해 투명성을 가지므로, 한 노드내의 IPC와 노드 간 IPC에 구분이 없이 사용할 수 있다. 이는 TMO의 네트워크로 구성된 실시간 시스템의 분산 환경의 노드 구조를 바꾸어도 프로그램의 수정이 필요 없음을 의미한다.

실시간 서버 프로세스 및 분산 IPC와 관련된 커널 API는 다음과 같다.

- Rt_SP_Register(deadline) : 이 커널 API는 일반적 실시간 프로세스를 분산 IPC의 메시지 수신에 의해 구동되는 실시간 서버 프로세스로 변환하는 시스템 호출이다. 매개변수 deadline은 매 메시지 수신 후, 이 메시지에 대한 완료 시까지의 데드라인을 의미한다.
- Alloc_Channel (Channel_Id, Max_Message_Size) : 등록을 마친 실시간 서버 프로세스나 주기적 실시간 프로세스가 앞으로 사용할 분산 IPC 채널을 시스템에 할당 등록하는 API이다. 이와 같은 채널의 할당은 이 채널을 사용할 모든 분산 노드의 모든 프로세스가 해야 하며, 이 채널은 이용하여 송신되는 메시지는 같은 Channel_Id를 사용하는 모든 분산 노드에 배달된다.
- Rt_SP_Wait_Invocation (Channel_Id, Msg_P) : 등록을 마친 실시간 서버 프로세스가 분산 IPC 메시지의 수신을 기다리는 API이다. 메시지의 수신에 의해 구동이 시작되면 서비스 완료, 즉, 다음 Rt_SP_Wait_Invocation의 호출 까지 데드라인 기반 스케줄링 적용된다. 이러한 시스템 호출들을 사용한 실시간 서버 프로세스의 실행 형태는 다음과 같다.

```
Rt_SP_Register (deadline) ;
Alloc_Channel ( Channel_ID,
                Max_Message_Size) ;
while ( Rt_SP_Wait_Invocation (Channel_id,
      Message-pointer) = = TRUE) {
    do a service job within its given
  deadline ;
}
Dealloc_Channel(Channel_Id) ;
Rt_SP_Exit() ;
```

- Send_Message(Channel_Id, Message_Pointer) : 이 시스템 호출은 같은 노드나 원격 노드들의 해당 채널로 메시지를 배달하는 API 이다. 각 노드의 채널로 메시지가 배달되면 각 노드의 채널에서 대기하던 프로세스들이 구동된다.
- Receive_Message(Channel_Id, Message_Pointer) : 이 시스템 호출은 메시지 수신을 기다린다는 점은 Rt_SP_Wait_Invocation() 과 같다. 그러나 Rt_SP_Wait_Invocation() 이 실시간 서버 프로세스의 구동과 데드라인 스케줄링의 시작에 사용되는 반면, Receive_Message는 단순 메시지 수신 대기에 사용되는 분산 IPC 인터페이스이다.

3. TMO-Linux의 스케줄러

TMO_Linux의 스케줄러는 기존 리눅스 커널의 스케줄러 부분에 주기적 실시간 프로세스와 실시간 서버 프로세스를 위한 데드라인 기반 스케줄링이 추가된 것으로, 클럭 인터럽트의 bottom-half handler 내에 위치하여 1/1000초 간격으로 실행된다. TMO_Linux의 스케줄러가 참조하는 주요 정보는 각 실시간 프로세스들의 MCB내 시간 조건에 관한 정보들이다. MCB 내의 각종 시간 조건에 대한 정보는 매 스케줄러 구동 시마다 동적으로 변화된다. 1밀리 초마다 구동되는 TMO_Linux 스케줄러의 주요 작업을 열거하면 다음과 같다.

- 주기적 실시간 프로세스의 주기적 구동(On_time Activation)
- 실행 중인 주기적 실시간 프로세스 및 실시간 서버 프로세스의 잔여 데드라인에 의한 우선 순위 조정(Deadline Driven Scheduling)
- 클럭 인터럽트 마지막에서의 우선 순위에 의한 문맥 교환(Context Switching)

실행 중인 두 가지 형태의 프로세스에 대한 우선 순위 조정 부분은 rate-monotonic이나 EDF (Earliest Deadline First)등이 사용될 수 있으나 TMO-Linux는 기본적으로 실행 완료까지의 잔여 시간(laxity)를 기반으로 한 LLF(Least Laxity First) 스케줄링을 제공한다. 즉 TMO_Linux는 각 프로세스들의 매번 수행 시마다 경험적 worst-case-execution-time을 기록하여 이를 스케줄링에 다음과 같이 반영한다.

new Priotiy=f(deadline-left-to-finish−(historic-worst-case-execution-time-execution-time))

위에서 execution-time은 금번의 실행에서의 누적 실행 시간을 의미한다.

4. 분산 실시간 객체 TMO의 지원

앞에서도 설명되었듯이 TMO의 지원을 위해서는 다음과 같은 기능들이 제공되어야 한다.

- 객체 내의 두 가지 형태의 멤버-스레드(SpM과 SvM)
- 데드라인 기반 스케줄링과 분산 IPC

객체 내의 메소드인 SpM과 SvM은 각각 스레드로서 구현되어 객체 내의 자료를 공유하도록 하여야 한다. 단 SpM들은 실행 형태상 주기적 실시간 프로세스로, SvM들은 실시간 서버 프로세스로 각각 변환되어야 한다. 그러나 이 프로세스들은 객체 내의 자료를 공유하여야 하므로 clone 시스템 호출을 통해 자료 영역을 공유하는 프로세스로 생성되어야 한다. TMO를 지원하는 매크로 및 객체 라이브러리의 기본적 역할은 SpM과 SvM을 각각 clone된 주기적 실시간 프로세스와 실시간 서버 프로세스로 변환 생성하는 일이다. 참고로 미들웨어 TMO 엔진인 LTMOS에서는 SpM과 SvM들이 Pthread로 구현되었으나, TMO_Linux에서 이들을 자료영역을 공유하는 clone된 프로세스로 사상되어 제어의 계층을 축소하였다. 〈그림 3〉은 이와 같은 기능을 지원하는 매크로 및 객체 라이브러리를 이용한 TMO 인스턴스의 예이다.

```
class my_TMO : public TMO
{
private :
        void      SpM1(void);
        void      SvM1(void);
        SpM (SpM1);
        SvM (SvM1);
private:
    Object Data Store; // shared by SpMs and SvMs
public :
        void      InitInstance(void);
          :
};
void my_TMO::InitInstance(void)
{  // SvM, SpM initialization
        SpM_Init (SpM1);
        SvM_Init (SvM1);   :
}
```

```
SpM_Body (my_TMO, SpM1)
void my_TMO::SpM1(void)
{
   Rt_PP_Register (100, 150, 200, FOREVER);
  While (Rt_PP_WaitInvocation() == TRUE) {
                do_realtime_job();
  }
  Rt_PP_Exit();
}
Svm_Body(my_TMO, SvM1)
void my_TMO::SvM1(void)
{
    ...
}
```

〈그림 2〉 TMO class 정의의 예

위의 예에서 사용된 매크로나 사전 정의된 객체의 기능은 다음과 같다.

- TMO class : 일반 C++ 클래스가 기본 TMO 클래스를 상속하여 사용자가 정의하는 TMO 클래스로 선언된다. 기본 TMO 클래스는 TMO의 생성과 관리에 필요한 정보들을 갖는다.
- SpM(SpM name)과 SvM(SvM name) 매크로 : SpM name과 SvM name을 가진 멤버 함수가 일반 멤버 함수가 아닌 스레드로 구동되는 SpM이나 SvM임을 선언하는 매크로이다
- SpM_Init(SpM name)과 SvM_Init(SvM name) 매크로 : TMO의 멤버함수 InitInstance()는 SpM_Init(SpM name)과 SvM_Init(SvM name)을 포함하도록 반드시 작성되어야 한다. SpM_Init(SpM name)과 SvM_Init(SvM name)은 clone 시스템 호출을 통하여 TMO_Linux의 자료영역을 공유하는 native thread들을 생성하는 역할을 한다.
- SpM_Body(TMO name, SpM name)과 SvM_Body(TMO name, SvM name) 매크로 : 뒤에 이어지는 멤버함수가 SpM이나 SvM의 몸체임을 선언한다.

이와 같이 생성된 SpM이나 SvM은 실제 TMO의 ODS를 공유하는 TMO-Linux의 실시간 프로세스로 생성되므로 생성 후에 각각, 주기적 실시간 프로세스 인터페이스와 실시간 서버 프로세스 인터페이스를 사용하게 된다.

VI. 결 론

빠르거나 온라인이라는 의미가 아닌, 보장성 컴퓨팅을 제공하는 실시간 시스템을 구축하는 것은 매우 어려운 일이다. 내장형 시스템의 경우에는 자원에 대한 동적인 요구가 한정되므로 보장성 실시간 컴퓨팅을 구현하는데 도움이 되는 요소를 갖고는 있지만, 그렇다 하더라도, 상용 내장형 실시간 운영체제를 도입하는 것만으로 해결되는 것은 아니다. 즉, 실시간 시스템을 구축하려면 가장 상위 수준의 시스템 설계 기법에서부터 가장 하위 수준의 실시간 운영체제의 보장 및 빠른 응답에 이르기 까지 전 과정에서의 실시간 시스템을 위한 기술 또는 도구가 있어야 한다.

이러한 측면에서 TMO-Linux는 응용 시스템 설계자를 위한 새로운 실시간 프로그래밍 모델인 TMO를 제공하도록 개발된 리눅스 기반 운영체제이다.(물론 내장형 리눅스 계열로도 사용 가능하다.)

TMO-Linux는 이러한 실시간 프로그래밍 모델에만 초점을 맞춘 것이므로, 본격적인 경성 실시간 시스템용 운영체제가 되기 위해서는 기존

RTOS의 접근 방법에 의한 최적화도 이루어져야 할 것이다. 그러나 TMO-Linux는 일단 실시간 응용을 구축하는 좀 더 체계적인 모델을 지원하는 최초의 리눅스 기반 운영체제라는 점과, TMO 미들웨어들보다 스케줄링 정확도를 향상 시켰다는 점에 의미를 가진다 할 수 있다.

참 고 문 헌

[1] Kim, K. H. and Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", Proc. 18th IEEE Computer Software & Applications Conference, pp.392-402, November 1994.

[2] Kim, K., "Object Structures for Real-Time Systems and Simulators", IEEE Computer, August 1997, pp.62-70.

[3] Kim, K. H., "Group Communication in Real-Time Computing Systems : Issues and Directions", Proc. FTDCS '99 (7th IEEE Workshop on Future Trends of Distributed Computing Systems), Cape Town, South Africa, Dec. 1999, pp.252-258.

[4] Kim, J. G., Kim, M. H., Min, B. J., and Im, D. B., "A Soft Real-Time TMO Platform-WTMOS-and Implementation Techniques", Proc. 1st IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 256-264, April 1997.

[5] Kim, J. G. and Cho, S. Y., "LTMOS : An Execution engine for TMO-Based Real-Time Distributed Objects", Proc. PDPTA'00 Vol. V, pp2713-2718, Las Vegas, June 2000.

[6] Kim, H. J., Park, S. H., Kim, M. H. and Kim, J. G., "TMO-Linux : A Linux-Based Real-Time Operating System Supporting Execution of TMOs, " Proc. 5th IEEE International Symposium on ObjectOriented Real-Time Distributed Computing, pp.288-296, Washington. D. C. April, 2000.

저 자 소 개

金 正 國

1977년 2월 서울대학교 계산통계학과(학사), 1979년 2월 한국과학기술원 전산학과(석사), 1986년 3월 한국과학기술원 전산학과(공학박사), 1983년 3월~1985년 2월 : 한국외대 컴퓨터공학과 전임강사, 1986년 3월~1990년 2월 : 한국외대 컴퓨터공학과 조교수, 1990년 3월~1995년 2월 : 한국외대 컴퓨터공학과 부교수, 1995년 3월~현재 : 한국외대 컴퓨터공학과 정교수, 〈주관심 분야 : 운영체제, 실시간 시스템, 분산 처리〉