실시간 내장 멀티태스킹 커널의 개발에 재사용 가능한 객체지향 커널 프레임워크의 설계 및 구현

(Design and Implementation of An Object-Oriented Kernel Framework Reusable for the Development of Real-Time Embedded Multitasking Kernels)

이 준 섭 † 전 태 웅 †† 이 승 룡 †††
(Jun Seob Lee) (Taewoong Jeon) (Sungyoung Lee)

요 약 실시간 내장 시스템은 운용 환경과 용도에 따라 멀티태스킹 커널이 지원해야 할 하드웨어 플랫폼과 자원 관리 정책이 다양하게 달라진다. 실시간 내장 시스템에 요구되는 크기와 성능 상의 제약 조건들을 엄격하게 만족해야 하는 멀티태스킹 커널을 새로운 시스템 서비스나 하드웨어에 맞게 개조하는 것은 쉽지 않다. 본 논문은 마이크로 프로세서 기반의 실시간 내장 멀티태스킹 커널의 구현에 재사용 가능한 객체지향 커널 프레임워크의 프로토타입을 객체 합성과 클래스 상속 메커니즘에 의거한 프레임워크 설계 패턴들을 적용하여 개발한 사례를 설명한다. 본 커널 프레임워크는 필요한 커널을 프레임워크의 개조와 확장을 통하여 효율적이고 일관성있게 생성할 수 있도록 하드웨어 환경과 시스템 자원 관리 정책에 의존적인 부분들이 프레임워크의 가변 부위(hot spot)로 분리된 추상 클래스들로 설계되어 있어서 높은 이식성과 개조성이 요구되는 마이크로 프로세서 기반 실시간 내장형 멀티태스킹 커널의 구현에 효과적으로 사용될 수 있다.

Abstract Real-time embedded systems should accommodate many kinds of hardware platforms and resource management policies that vary depending on their operating environments and purposes. It is not an easy job to adapt a multitasking kernel to new system services and hardware platforms, as the kernel must strictly satisfy constraints on its size and performance. This paper describes the design and implementation of an object-oriented multitasking framework that can be reused for implementing microprocessor-based real-time embedded multitasking kernels. In this kernel framework, those parts that can vary depending on hardware platforms and system resource management policies are separated into the hot spots and encapsulated by abstract classes. Our framework thus can be effectively used to implement microprocessor-based real-time embedded kernels that demand high portability and adaptability.

· 본 논문은 한국과학재단의 특정연구개발과제(과제번호: KOSEF 95-0100-07-01-3)와 98핵심전문연구과제(과제번호: KOSEF 981-0923-124-2)의 지원에 의해서 작성되었습니다.
† 비 회 원 : 한국전자통신연구원 표준연구센터 연구원
jslee@pec.etri.re.kr
†† 종신회원 : 고려대학교 자연과학부 교수
jeon@tiger.korea.ac.kr
††† 종신회원 : 경희대학교 전자정보학부 교수
sylee@oslab.kyunghee.ac.kr
논문접수 : 1999년 5월 11일
심사완료 : 2000년 1월 6일

1. 서 론

실시간 내장 응용 프로그램을 효율적으로 개발하기 위해서는 응용 프로그램의 하부에서 병행 태스크들의 생성, 스케줄링, 동기화, 통신 등의 일을 처리해 주는 마이크로 커널이 필요하다[1]. 이러한 실시간 내장 시스템에서 사용되는 커널은 크기와 성능에 엄격한 제약 조건이 따르는 반면, 커널이 지원해야 할 하드웨어 플랫폼이나 시스템 자원 관리 정책은 운용 환경과 용도에 따라 다양하게 달라진다.

기존의 실시간 내장 멀티태스킹 커널들은 재구성 가능한 요소들(configurable elements)에 대한 파라미터들의 설정을 통하여 제한된 범위의 하드웨어 플랫폼과 서비스 정책들만을 지원하도록 설계되어 있다. 예를 들면, RTMK[2]은 우선 순위 기반의 스케줄링만을 지원하는 멀티태스킹 커널이다. 지원되지 않는 새로운 형태의 환경이나 서비스의 제공이 커널에 필요한 경우에는, 기존 커널의 소스코드를 이에 맞도록 직접 임의로 변경해야 하거나 적합한 커널을 신규로 구입 또는 개발하여야 한다.

본 논문에서는 멀티태스킹을 지원하지 않는 C 또는 C++언어로 작성된 실시간 내장 응용 프로그램에게 멀티태스킹 서비스를 제공하기 위한 마이크로 커널의 구현에 재사용 가능한 객체지향 커널 프레임워크를 프로토타입으로 개발한 사례를 설명한다. 본 커널 프레임워크는 상호 협동하는 추상 클래스들과 구체 클래스들로 구성된 객체지향 프레임워크 설계 패턴[3]들로 설계되어 있으며, 하드웨어 플랫폼이나 자원 관리 정책에 의존적인 부분과 독립적인 부분이 각각 커널 프레임워크의 가변 부위(hot spot)와 고정 부위(frozen part)[4,5]로 구분되어 있다.

새로운 멀티태스킹 커널은 본 커널 프레임워크의 가변부위에 해당하는 클래스들을 이들이 제공하는 인터페이스에 맞게 객체 합성과 클래스 상속과 같은 객체지향 재사용 메커니즘[4]에 따라 개조, 확장함으로써 효율적이고 일관성있게 생성할 수 있다. 이렇게 생성된 커널은 실행 시 특정 용도에 맞게 구체화된 커널 객체들이 상속, 다형성 및 동적 결합 메커니즘에 의해 상호 작용함으로써 시스템 서비스를 수행한다.

본 커널 프레임워크가 지원하는 시스템 서비스는 병행 태스크 관리, 멀티태스킹에 필요한 CPU, 메시지 및 세마포어 자원들의 스케줄링과 관리, 그리고 타이머와 인터럽트의 처리 기능 등이다. 메모리 관리, 주변장치 드라이버, 네트워킹 등은 본 커널 프레임워크에 포함되어 있지 않다. 그러나 필요한 경우 프레임워크에 이러한 서비스들을 추가하거나 이들을 지원하는 다른 시스템 프로그램들과 상호 운용이 가능하도록 설계되어 있다.

본 논문의 구성은 다음과 같다. 2장에서는 본 커널 프레임워크에서 개조와 확장이 고려된 가변 부위를 기술한다. 3장에서는 본 커널 프레임워크의 아키텍처가 제공하는 공통적인 제어 흐름들을 기술한다. 4장에서는 본 커널 프레임워크의 아키텍처를 구성하는 핵심 클래스들의 설계 패턴들을 기술한다. 5장과 6장에서는 본 커널 프레임워크를 개조, 확장하여 커널을 생성하는 메커니즘과 그렇게 생성된 커널의 동작을 각각 기술한다. 7장에서는 본 커널 프레임워크을 커널로 확장한 예를 들어 연구 결과를 설명하고 기존의 연구들과 비교한다. 그리고 8장에서 결론을 맺는다.

2. 커널 프레임워크의 가변 부위

커널 프레임워크가 보다 넓은 범위의 실시간 내장 시스템들을 지원하기 위해서는 구현할 멀티태스킹 커널이 서로 다른 하드웨어에 이식되더라도 하드웨어에 종속적인 부분만을 쉽게 분리하여 수정함으로써 사용이 가능해야 하며, 응용 프로그램의 용도에 따라 시스템 자원에 대한 서로 다른 스케줄링 정책과 이에 적합한 알고리즘을 제공할 수 있어야 한다. 예를 들면, 메시지나 세마포어를 기다리고 있는 태스크가 여러 개인 경우에 어떤 태스크에게 메시지나 세마포어를 전달할 것인지를 결정하는 방법을 다양하게 지원해야 한다. 즉, 태스크에게 자원을 할당할 때 FIFO 방식이나 우선순위 방식 등 중, 커널이 사용될 응용 프로그램의 요구에 맞는 스케줄링 정책을 선택하여 구현할 수 있어야 한다.

이러한 요구 조건들을 만족하는 커널 프레임워크는 멀티태스킹 커널의 알고리즘 의존적인 부분과 하드웨어 의존적인 부분을 이들과 독립적인 부분들로부터 분리하여 프레임워크의 가변부위에 속한 클래스들로 추상화함으로써 설계할 수 있다.

2.1 표준 멀티태스킹 커널

지금까지 멀티태스킹 커널은 일반적으로 특정 하드웨어나 목적을 위하여 만들어져 왔다. 따라서 이러한 커널들은 다른 하드웨어에 이식될 수 없었다. 뿐만 아니라 특정 응용 프로그램을 위하여 만들어 졌기 때문에 다른 응용 프로그램이 이 커널을 사용하기가 어려웠다. 이러한 점을 개선하기 위하여 1982년 유럽의 소프트웨어 회사들이 실시간 커널을 위한 간단하면서도 현실적인 명령어들을 제안하였다. 이것을 Sceptre 표준이라고 한다. 요즘에 개발되는 커널은 이 표준에서 정의하는 서비스를 제공하도록 설계되어있다. 이것은 커널을 새로운 하드웨어에 이식했다면, 커널의 실시간 서비스를 사용하는 다른 모든 응용 프로그램이 이식될 수 있음을 의미한다.[2]

여기서는 Sceptre 표준에서 정의하고 있는 서비스와 Sceptre 표준이 포함하고 있는 요소들에 대해 설명하고자 한다.

2.1.1 표준 멀티태스킹 커널의 서비스

Sceptre 커널에서 정의된 서비스는 [표 1]과 같다.

[표 1]의 커널 서비스들은 사용자의 태스크가 이용할

표 1 Sceptre 커널 서비스[2]

Class	Operation	Parameters	Action
Task Handling	Start	Task	Starts the execution of the task
	Stop	Task	Stops the execution of the task
	Continues	Task	Resumes the execution of the task
	Terminate		Terminates the task that calls this service
	Change priority	Task, priority	Gives task the new priority
	State	Task	Returns task state
	Priority	Task	Returns the task priority
	Current task		Return the id of current executing task
Signaling	Send	Event, task	Sends an event to a task
	Wait	Event list	Waits until one of the events has arrived
	Arrived	Event list	True if all the events have arrived
	Clear	Event list	Resets all events in list to nonarrived state
Communication	Put	Element, queue	Places the element into the queue
	Get	Element, queue	Gets the element from the queue
	Empty	Queue	True if the queue is empty
	Full	Queue	True if the queue is full
Mutual-exclusion	Lock	Region	Requests exclusive property of the Region
	Unlock	Region	Release the region
Task States	Nonexisting		No descriptor associated to task (not created)
	Existing		A descriptor has been defined for the task
	Nonexecutable		Task has descriptor but can' t start execution
	Executable		Task has descriptor and can start execution
	Not in service		Task is executable but execution has not yet been started or is finished
	In service		Task is executable and has started execution but has not finished
	Waiting		Task is in service and waiting for a condition to become True to continue execution
	Active		Task is in service and not waiting for anything except a free processor to execute
	Ready		Task is active and only waiting for processor
	In Progress		Task is active and currently executing on a processor

수 있는 커널 서비스들을 나타내고 있다. 이것은 멀티태스킹 커널에 대한 인터페이스를 나타내며 위에서 나타내지 않은 스케줄링에 관련된 서비스는 커널 수준에서 실행되므로 사용자 태스크는 이를 사용할 수 없다. 즉, 사용자 태스크에서 [표 1]의 서비스를 요청했을 때 멀티태스킹 커널이 필요에 따라 스케줄링을 수행하게 된다.

Sceptre 커널은 앞에서 설명한 멀티태스킹 커널의 기본적인 서비스들 중에서 입출력과 메모리 관리를 제외한 서비스를 제공하고 있다. 태스크 관리는 Task handling, 동기화는 Signaling, 통신은 Communication, 상호 배타성은 Mutual-exclusion에서 처리하고 있다.

Sceptre 커널의 태스크는 여러 상태를 갖는다. 태스크가 아직 생성되지 않은 상태를 나타내는 Nonexisting, 태스크의 제어 블록이 생성된 Existing, 태스크 제어 블록은 있으나 실행될 수 없는 상태인 Nonexecutable, 태스크는 실행 가능하나 아직 실행되지 않은 상태인 Not in service, 태스크의 수행이 아직 끝나지 않은 In service, 태스크가 In service 상태에서 수행을 계속하기 위해 어떤 조건이 만족되기를 기다리고 있는 상태인 Waiting, In service 상태에서 어떤 이벤트를 기다리지도 않으며 비어있는 프로세서를 기다리는 Active, Active 상태에서 프로세서를 기다리고 있는 Ready, 그리고 Active 상태에서 프로세서에서 수행 중인 In progress 상태가 그것이다.

이 모든 태스크 상태가 모든 멀티태스킹 커널에서 사용되는 것은 아니다. 이러한 여러 개의 상태는 필요하게 될지도 모르는 용도를 위해 정의한 것이고 각 멀티태스킹 커널은 필요에 따라 몇 개의 상태만을 정의하고 사용한다. 예를 들어 RTMK[5]에서 태스크는 어떤 신호가 도착하기를 기다리는 Waiting, 필요한 모든 신호가

도착하고 CPU 시간을 할당 받기를 기다리는 Ready, 그리고 수행 중인 In progress를 정의하여 사용하고 있다.

2.1.2 표준 멀티태스킹 커널의 구성 요소

Sceptre 표준이 포함하고 있는 요소는 다음과 같다.[5]

- 태스크/프로세스

프로그램의 수행을 담당하는 에이전트로 이름과 속성을 가지고 있다. 태스크의 컨텍스트는 특정 프로세서가 태스크의 명령어들을 수행하는데 필요한 최소한의 정보를 의미한다.

- 인접 태스크(immediate tasks)

실시간 응용 프로그램과 환경 사이의 인터페이스를 제공하는 태스크로서 주로 인터럽트 메커니즘을 사용하여 조작된다.

- 스케줄러

프로세서의 CPU 파워를 조작하고 이들을 태스크에 할당하는 소프트웨어 또는 하드웨어 모듈로 인접 태스크의 스케줄러는 프로세서의 인터럽트 메커니즘을 사용하고 일반 태스크의 스케줄러는 커널의 일부분으로 되어있다.

- 이벤트

이벤트 객체는 어떤 조건이 만족되어졌다는 신호를 포함하고 있다. Sceptre 표준에서 이벤트는 특정 태스크와 관련이 있으며 이는 효과적인 신호 처리를 가능하게 한다.

- 지역(region)

지역 객체는 CPU의 소유권의 표시이다. 어떤 지역이 아직 수행 중 일 때에는 컨텍스트 스위칭이 발생하지 않는다. 이것은 흔히 임계 영역(critical section)이라고 불리는 것으로 프로그램의 공유 자원에 접근하는 코드를 가진 부분을 말한다.

- 큐(queue)

FIFO 메모리처럼 작동하는 태스크간의 통신 객체이다.

2.2 알고리즘 의존적인 커널의 구성 요소

시스템 자원에 대한 스케줄링 정책이나 스케줄링 알고리즘이 달라지면 이에 따라 태스크 제어 블록이나 준비 큐(ready queue)와 같이 커널의 자원을 관리하거나 이에 영향을 받는 커널의 구성 요소들의 구조가 바뀌어야 한다. 예를 들면 우선순위 기반의 CPU 스케줄링을 사용하는 커널의 태스크는 그 속성으로 우선 순위를 가져야 하며 마감시간 기반의 스케줄링을 사용하는 경우에는 태스크들의 시간제약에 관련된 정보를 속성으로 가져야 한다.

CPU 할당을 대기 중인 태스크들을 관리하는 준비 큐는 일반적으로 태스크 제어 블록들에 대한 더블 링크 리스트(double linked list)의 구조로 구현되지만 성능 향상을 위해 적용하는 스케줄링 알고리즘에 따라 서로 다른 구조를 가질 수 있다. 예를 들어 C/OS[6]는 준비 큐가 비트맵(bitmap) 형태로 되어 있다. 이러한 준비 큐의 구조는 빠른 연산 알고리즘을 통하여 최우선 순위의 태스크를 상수 시간 안에 찾아 낼 수 있도록 하고 있다.

메시지 전달 방식에 따라서도 메시지를 기다리는 태스크들 중에서 메시지를 전달할 태스크를 찾는 서로 다른 알고리즘이 필요하게 되며 이에 따라 이벤트 제어 블록의 구조도 함께 바뀌어야 한다. 세마포어에 대한 스케줄링에서도 메시지의 경우와 유사하게 관련 구성 요소들이 바뀌어야 한다.

본 커널 프레임워크에서는 스케줄링 정책과 알고리즘에 의존적인 부분들이 Strategy 설계 패턴[7]에 따라 개조, 확장이 가능하도록 프레임워크의 가변 부위에 추상화되어 설계되어 있다.

2.3 하드웨어 의존적인 커널의 구성 요소

멀티태스킹 커널이 탑재될 하드웨어에 따라 변경되어야 하는 커널의 구성 요소는 태스크 제어 블록, 컨텍스트 스위칭, 인터럽트 처리 및 시스템 초기화들이 있다.

태스크 제어 블록은 흔히 이미지라고 불리는 레지스터들의 집합과 스택 포인터, 명령 문 포인터(instruction pointer) 등을 가지고 있다. 하드웨어마다 레지스터의 종류와 크기가 다르므로 이러한 이미지는 커널이 사용되는 기반 하드웨어에 따라 서로 다른 구조를 가져야 한다.

어셈블리 언어로 작성되어야 하는 인터럽트 서비스 루틴이나 컨텍스트 스위칭 루틴들은 하드웨어에 따라 각기 다르게 구현되어야 한다. 또한, 하드웨어에 따라 서로 다른 구조를 초기화하거나 이용하는 함수들도 사용되는 하드웨어에 맞게 변경 되어야 한다. 예를 들면, 하나의 태스크를 새로 생성할 때 초기화해야 하는 태스크 제어 블록의 구조가 다르게 설계되어야 하며, 이를 초기화하는 함수의 코드도 다르게 작성 되어야 한다.

본 커널 프레임워크에서는 이러한 하드웨어 의존적인 부분들이 프레임워크의 가변 부위에 속한 클래스들로 추상화되어, 이들을 Bridge 설계 패턴[7]에 따라 개조, 확장함으로써 이들의 수정이 커널 프레임워크의 다른 부분에 영향을 미치지 않으면서 특정 하드웨어에서 작동할 수 있는 커널을 쉽게 생성할 수 있도록 설계되어 있다.

3. 커널 프레임워크의 제어 흐름

본 논문의 멀티태스킹 커널 프레임워크는 커널이 지원해야 할 하드웨어 플랫폼이나 자원관리 정책에 관계없이 모든 커널의 동작에 공통적인 제어 흐름을 프레임워크가 제공하는 아키텍쳐로 설계되어 있다. 본 장에서는 그러한 커널 프레임워크의 제어 흐름들 중, 태스크의 생성, 태스크의 지연, 메시지 송신, 그리고 메시지 수신의 4가지 경우를 예로 들어 설명한다.

태스크의 생성 요청에 대한 제어의 흐름을 살펴 보면 [그림 1]과 같다. 태스크 생성 요청에 대한 서비스는 TaskManager 클래스가 처리한다. TaskManager는 TCB(태스크 제어 블록) 클래스 객체를 생성하고 이를 준비 상태로 초기화한다. 그런 후, TCB 인스턴스로 생성된 태스크를 ReadyQueue 클래스 인터페이스를 통하여 준비 큐에 추가하여 실행 대기시킨다. 마지막으로, TaskManager는 준비 큐에서 실행 대기 중인 태스크들에 대한 CPU 스케줄링을 Scheduler 클래스 인터페이스를 통하여 스케쥴러 객체에게 요청함으로써 태스크 생성 서비스를 마친다.

생성된 TCB 객체들은 TaskManager가 보관, 관리한다. ReadyQueue 클래스는 Scheduler가 CPU를 태스크에게 할당하는데 필요한, 준비 상태의 태스크 정보를 저장, 관리한다. 예를 들면, 우선 순위 기반의 선점형 스케줄링의 경우에는 ReadyQueue에 태스크 ID와 우선 순위가 저장되게 된다.

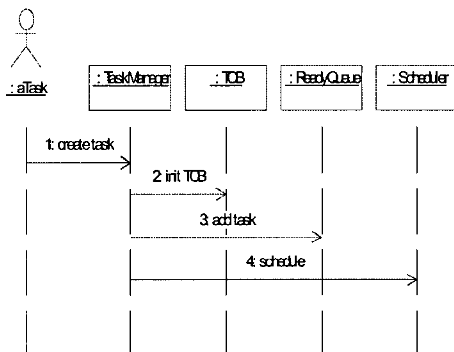

그림 1 태스크 생성

태스크를 일정 시간 동안 중지시키는 제어의 흐름은 [그림 2]와 같다. TaskManager는 우선 해당 태스크의 속성 중 지연 시간을 나타내는 속성을 일정 시간으로 설정하게 된다. 그런 후 TaskManager는 ReadyQueue 클래스 인터페이스를 통하여 준비 큐에게 지연시킬 태스크의 삭제를 요청한다. 마지막으로, 스케줄러에게 CPU스케줄링을 요청하게 된다.

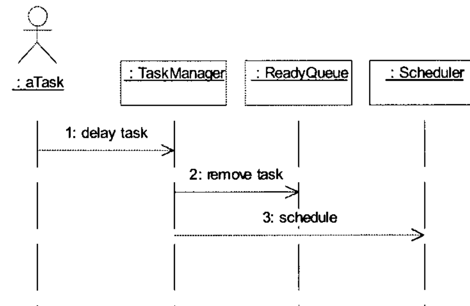

그림 2 태스크 지연

태스크가 메시지 큐에 어떤 메시지를 비동기적으로 전송했을 때의 제어의 흐름은 [그림 3]과 같다. [그림 3]에 나타난 객체들 중, MessageQueue는 태스크에게 전송할 메시지들을 관리하는 큐이고 EventQueue는 해당 MessageQueue 객체로부터 메시지 수신을 기다리고 있는 태스크들의 큐이다. ReadyQueue는 CPU 할당을 기다리는 준비 상태의 태스크들을 관리하는 큐이다.

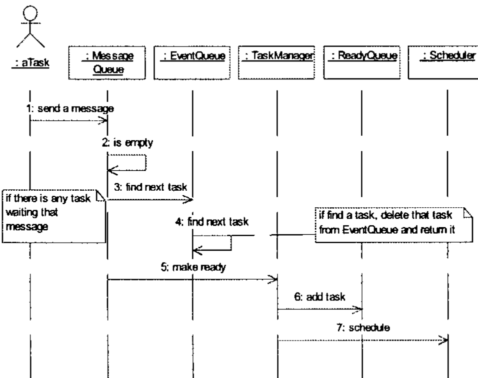

그림 3 메시지 송신

[그림 3]과 같이 태스크가 MessageQueue객체에게 메시지를 전송하면, 메시지 큐는 먼저 메시지 큐가 비어 있는지 검사한다. 메시지 큐가 비어 있지 않으면 아직

이 메시지를 기다리는 태스크가 없다는 것이므로 전송받은 메시지를 메시지 큐에 추가함으로써 모든 과정이 끝나게 되고 메시지 송신을 요청한 태스크가 계속 진행한다. 반대로 메시지 큐가 비어 있으면 이는 해당 메시지 큐에서 메시지를 기다리고 있는 다른 태스크가 존재할 수 있음을 의미한다. 이 경우, 메시지 큐는 자신의 EventQueue에서 이 메시지를 기다리는 태스크를 찾는다.

EventQueue는 미리 정해진 메시지 전달 방식에 따라 메시지를 전달할 태스크를 결정한다. 메시지를 수신할 태스크가 메시지 큐에게 리턴된 경우, 메시지 큐는 Task Manager에게 메시지를 수신할 태스크에 대해 준비 상태로의 변경을 요청한다. TaskManager는 어떤 태스크가 대기 상태에서 준비 상태로 변경되었으므로 이를 ReadyQueue에 추가하고 스케줄러에게 CPU 스케줄링을 요청하게 된다.

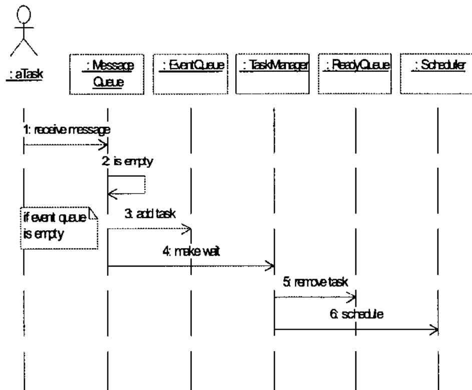

그림 4 메시지 수신

태스크가 메시지를 전달 받고자 할 때의 제어의 흐름은 [그림 4]와 같다. 태스크로부터 메시지 수신을 요청받은 메시지 큐가 비어 있지 않으면 어떤 메시지가 이미 도착해 있는 것이므로 태스크는 메시지 큐로부터 메시지를 전달 받고 수행을 계속하게 된다. 그러나 메시지 큐가 비어 있으면 메시지 큐는 메시지 수신을 요청한 태스크를 자신의 EventQueue에 추가하고 Task Manager에게 해당 태스크에 대해 대기 상태로의 변경을 요청한다. Task Manager는 ReadyQueue에서 해당 태스크를 삭제하고 스케줄러에게 CPU 스케줄링을 요청한다. 위와 같은 메시지 송수신 제어 흐름은 세마포어 자원의 송수신 제어흐름에서도 유사하게 이루어진다.

4. 커널 프레임워크의 아키텍쳐

본 장에서는 구현된 커널 프레임워크의 아키텍쳐를 기술한다. 본 논문의 커널 프레임워크는 멀티태스킹 커널의 기본적인 서비스들을 크게 태스크 관리, CPU 관리, 메시지 관리, 세마포어 관리로 분담하여 수행하도록 설계되어 있다. 이러한 서비스들을 담당하는 주요한 클래스들은 [그림 5]와 같다.

[그림 5]에서 MessageQueue와 Semaphore는 각각, 메시지 자원과 세마포어 자원에 대한 서비스를 담당하는 클래스들이며, 요청한 자원의 할당을 기다리는 대기 상태의 태스크들을 관리하는 EventQueue 클래스를 구성 요소로 갖는다. Scheduler 클래스는 CPU 자원을 관리하는 클래스로서, ReadyQueue에서 CPU 할당을 기다리는 준비 상태의 태스크들에 대한 CPU 스케줄링을 담당한다.

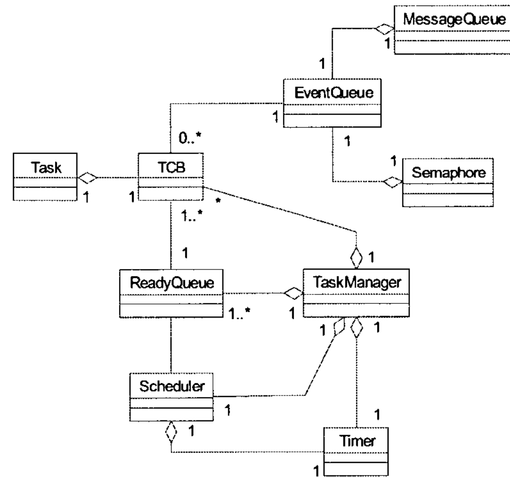

그림 5 커널 프레임워크의 구성 요소들

Task는 독립된 단일 스레드 상에서 실행 중인 프로그램 단위를 나타낸다. 생성된 태스크의 속성과 컨텍스트는 TCB 클래스에서 해당 태스크가 소멸될 때까지 유지, 관리된다. 태스크는 모든 작업이 끝난 후 스스로를 삭제하거나 무한 루프를 수행하는 형태이어야 한다.

Timer 클래스는 타이머의 설정과 설정된 타이머의 시간 경과를 관리하고 타임아웃 시, 이를 처리해야 할 구성 요소들에게 알려주는 역할을 담당한다. 타이머는 커널이 관리하는 CPU, 메시지, 세마포어 자원들을 기다리거나 할당받은 태스크들의 대기 시간이나 점유 시간

을 감시 제어하는데 사용된다. 타이머는 또한 태스크들이 요청한 특정 시각, 지연 시간, 또는 반복 주기를 설정하여 추적하고, 도달 시점에서 이를 알려주는 타이밍 서비스를 제공하는데도 사용된다.

4.1 태스크 관리 설계 패턴

본 커널 프레임워크에서 태스크의 생성에서 소멸에 이르기까지 태스크들에 대한 라이프사이클의 관리는 TaskManager 클래스를 중심으로 이루어진다. TaskManager 클래스는 생성된 태스크들에 대한 정보를 TCB 클래스를 사용하여 저장하고 있으며 태스크의 생성, 소멸 시 이에 대응하는 TCB 객체를 추가 또는 제거한다. 또한 TCB, Scheduler 및 ReadyQueue 클래스 객체들을 제어하여 태스크들의 실행 상태와 속성 값들을 변경, 유지, 관리한다.

TCB 클래스는 커널이 CPU 스케줄링과 컨텍스트 스위칭을 수행할 때 필요한 태스크 제어 정보를 저장하는 객체의 타입을 정의한 클래스이다. 생성된 하나의 태스크마다 이에 대응하는 하나의 TCB 인스턴스가 생성된다. TCB가 저장해야 할 태스크 제어 정보의 구성 요소들은 커널에 사용될 CPU의 종류와 스케줄링 알고리즘에 따라 달라진다. 따라서 TCB 클래스는 특정 하드웨어나 스케줄링 정책에 관계없이 커널에 공통적으로 필요한 인터페이스와 태스크 제어 정보만을 제공하며 커널의 구현 시 사용될 하드웨어와 스케줄링 정책에 따라 이에 적합한 구체적인 서브클래스로 개조, 확장된다.

이와 같이 태스크 관리에는 스케줄링 정책에 따라 가변적인 부분들이 존재한다. 예를 들면, 우선 순위 기반의 스케줄링을 사용하는 경우에는 태스크의 우선 순위를, 시분할 스케줄링을 사용하는 경우에는 CPU 할당 시간을 설정 또는 변경하는 서비스를 지원하는 것이 필요하다. 이를 위해서 TaskManager 클래스는 모든 스케줄링 정책에 공통적인 인터페이스만을 제공하고, 사용되는 스케줄링 알고리즘에 따라 달라지는 서비스 요청들에 대해서는 서브클래스에서 추가적인 인터페이스를 제공하도록 설계되어 있다.

[그림 6]은 태스크 관리에 관여한 클래스들을 나타낸 것이다. [그림 6]에서 TaskManager클래스는 자신의 인터페이스인 addTask(), removeTask(), setState() 등을 통하여 ReadyQueue와 Scheduler 클래스들이 지원하는 서비스들을 외부에 제공하고 이들의 개별적인 인터페이스들은 커널의 다른 구성 요소들로부터 은닉한 Facade 패턴[7] 구조로 설계되어 있다. TaskManager 클래스는 또한 ReadyQueue와 Scheduler 사이에 필요한 상호작용을 중재하고 제어하는 Mediator 행위 패턴[7]을 갖는다. TaskManager와 Scheduler 클래스는 각각 특정 커널에 한개의 인스턴스만이 존재하는 singleton[7] 클래스이다.

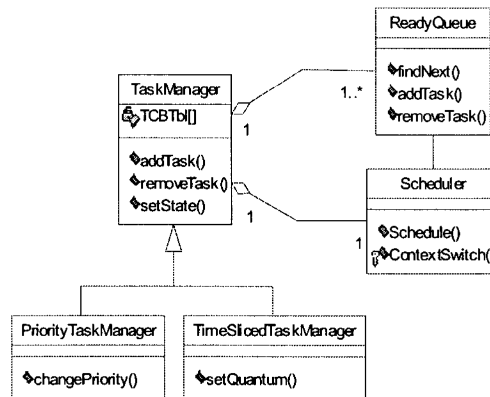

그림 6 태스크 관리 설계 패턴

TaskManager, Scheduler, ReadyQueue, 그리고 TCB 클래스는 모두 적용할 스케줄링 정책과 알고리즘에 따라 이에 적합한 서브클래스들로 확장되어 사용된다. 예를 들면, [그림 6]에서와 같이 TaskManager 클래스가 우선순위 기반과 시분할 기반 스케줄링 정책을 각각 지원하는 서브클래스들로 확장될 수 있다.

4.2 CPU 스케줄링 설계 패턴

시스템 자원에 대한 서비스 요청이나 타이머 등으로부터 인터럽트가 발생하면 커널 수준에서 이를 처리하는 과정에서 자원의 (재)할당에 필요한 스케줄링이 일어나게 된다. 특히, CPU 자원이 새로운 태스크에게 할당되어야 할 경우 CPU 스케줄링과 이에 따른 컨텍스트 스위칭이 일어나게 된다. 본 커널 프레임워크는 이러한 CPU 스케줄링을 Scheduler 클래스가 ReadyQueue 클래스를 사용하여 수행하도록 설계되어 있다. [그림 7]은 CPU 스케줄링에 관여한 클래스들의 설계 패턴을 보여준다.

Scheduler 클래스는 TaskManager로부터의 CPU 스케줄링 요청에 응하여 CPU를 할당 받을 다음 태스크를 결정하고 그 결과, 실행할 다음 태스크가 현재 실행 상태인 태스크와 다르면 이에 따른 CPU 컨텍스트 스위칭을 수행하는 역할을 한다. Scheduler 클래스는 Schedule() 멤버 함수를 인터페이스로 제공하며, 알고리즘에 의존적인 CPU 스케줄링과 하드웨어에 의존적인 컨텍스트 스위칭을 각각 Strategy 설계 패턴[7]과

Bridge 설계 패턴[7]에 따라 ReadyQueue클래스와 자신의 서브클래스에 위임한 클래스이다.

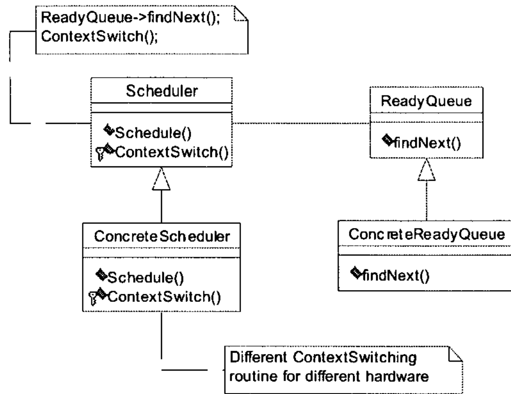

그림 7 CPU 스케줄링 설계 패턴

Scheduler 클래스의 인터페이스인 Schedule() 멤버함수는 ReadyQueue 클래스의 멤버함수인 findNext()와 자신의 멤버함수인 ContextSwitch()를 후크 메소드[4,7]들로 사용하여 구현된 템플릿 메소드[4,7]이다. Schedule()은 CPU를 할당 받아 다음에 수행되어야 할 태스크를 ReadyQueue클래스의 findNext() 멤버함수를 호출하여 찾는다. 컨텍스트 스위칭은 Scheduler클래스의 서브클래스에서 정의되는 가상 멤버 함수인 ContextSwitch()를 호출함으로써 이루어진다.

ReadyQueue 클래스는 CPU 할당을 기다리는 준비 상태 태스크들의 추가, 삭제 및 스케줄링에 필요한 인터페이스를 Scheduler 클래스에게 후크 메소드들로 제공하는 추상 클래스이다. ReadyQueue클래스의 인터페이스는 이를 상속받은 서브클래스에서, 사용될 CPU스케줄링 정책을 효율적으로 지원하는 자료 구조와 알고리즘으로 구현된다. 예를 들어, 우선 순위 방식의 선점형 스케줄링을 사용한다면 ReadyQueue 클래스의 서브클래스에서 ReadyQueue의 자료구조는 히프(heap) 구조를 갖는 우선순위큐로 구현하고, findNext() 멤버 함수는 우선순위큐에서 가장 높은 우선순위를 갖는 태스크를 상수 시간으로 빠르게 찾아내어 반환하도록 구현한다.

Scheduler 클래스의 ContextSwitch() 멤버 함수는 커널이 동작하는 하드웨어의 구조에 따라 달라지는 CPU 컨텍스트의 구성 요소와 스위칭 메커니즘을 자신의 서브클래스에 위임한 가상함수로서, 자신이 속한 클래스의 Schedule() 멤버함수에서 후크 메소드로 사용된다.

[그림 7]에서 Scheduler의 서브클래스들인 ConcreteScheduler와 ConcreteReadyQueue가 각각 특정한 컨텍스트 스위칭과 스케줄링 정책을 구현한 구체 클래스들이라고 가정하였을 때 커널 프레임워크의 스케줄링 수행 메커니즘은 다음과 같다. Scheduler클래스의 클라이언트인 TaskManager 객체가 Scheduler클래스의 인터페이스인 Schedule()을 호출하여 CPU스케줄링을 요청하면 동적 결합에 의해 ConcreteScheduler 클래스의 Schedule() 멤버함수가 호출된다. 호출된 Schedule() 멤버함수는 다시 동적결합에 의해 후크 메소드들인 ConcreteReadyQueue 클래스의 findNext() 멤버함수와 ConcreteScheduler 클래스의 ContextSwitch() 멤버함수를 호출하여 CPU 스케줄링과 이에 따른 컨텍스트 스위칭을 수행한다.

4.3 메시지 관리 설계 패턴

태스크들 사이의 통신을 지원하기 위한 메시지 전송 서비스는 MessageQueue 클래스가 관리한다. MessageQueue 클래스는 sendMessage()와 receiveMessage()를 인터페이스로 제공하며, 전송할 메시지들을 보관하는 MessageContainer 클래스와 메시지를 기다리는 태스크들을 보관하는 EventQueue클래스를 사용하여 메시지 전송 서비스를 수행한다. [그림 8]은 메시지 전송에 관여한 클래스들의 설계 패턴을 보여준다.

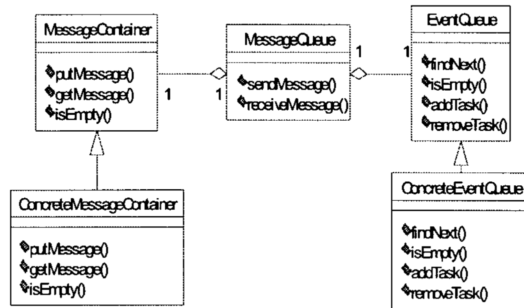

그림 8 메시지 전송 설계 패턴

MessageQueue 클래스는 태스크로부터 메시지의 송신이나 수신 요청 시 이에 응하여 메시지들을 태스크에게 전달하고 전송될 때까지 대기 중인 메시지들과 수신할 메시지를 대기 중인 태스크들을 관리, 스케줄링한다. MessageQueue 클래스는 태스크에게 전달할 메시지들과 메시지를 전달받을 태스크들에 대한 스케줄링을 각각 MessageContainer 클래스와 EventQueue 클래스에

게 위임한 Strategy 패턴[7]으로 설계되어 있다.

MessageContainer 클래스는 전송을 요청받은 메시지의 추가, 삭제 및 스케줄링에 필요한 인터페이스를 MessageQueue 클래스에게 후크 메소드들로 제공하는 추상 클래스이다. MessageContainer 클래스의 인터페이스는 이를 상속받은 서브클래스에서, 사용될 메시지 전달 방식에 적합한 자료구조와 스케줄링 알고리즘으로 구현된다. 예를 들면 FIFO 기반의 스케줄링에서는 가장 오랫동안 보관된 메시지가, 우선순위 기반의 스케줄링에서는 가장 우선 순위가 높은 메시지가 먼저 수신할 태스크에게 효율적으로 전달되도록 메시지 큐의 자료구조와 멤버함수들을 서브클래스에서 정의한다.

EventQueue 클래스는 메시지를 기다리고 있는 태스크들의 추가, 삭제 및 스케줄링에 필요한 인터페이스를 MessageQueue 클래스에게 후크 메소드들로 제공하는 추상 클래스이다. EventQueue 클래스의 인터페이스는 이를 상속받은 서브클래스에서, 사용될 메시지 전달 방식에 적합한 자료 구조와 스케줄링 알고리즘으로 구현된다. 예를 들면 FIFO 기반의 스케줄링 정책에서는 메시지를 가장 오래 기다린 태스크에게, 우선순위 기반의 스케줄링에서는 가장 우선순위가 높은 태스크에게 메시지를 효율적으로 전달할 수 있도록 서브클래스에서 태스크 큐의 자료구조와 멤버함수들을 정의한다.

MessageQueue 클래스는 태스크에게 전달할 다음 메시지를 선택하는 스케줄링 정책과 메시지를 수신할 다음 태스크를 선택하는 스케줄링 정책이 각각 MessageContainer 클래스와 EventQueue 클래스를 통하여 서로 독립적으로 구현될 수 있도록 설계되어 있다. 예를 들면 전달할 다음 메시지의 선택은 FIFO에 의해, 메시지를 수신할 다음 태스크의 선택은 우선순위 기반에 의해 이루어지도록 커널을 구현할 수 있다.

[표 2]는 MessageQueue 클래스의 인터페이스인 sendMessage()와 receiveMessage() 멤버함수의 구현 코드를 요약하여 보여준다. [표 2]에서와 같이, MessageQueue 클래스의 인터페이스는 MessageContainer 클래스와 EventQueue 클래스가 인터페이스로 제공하는 후크 메소드들과 TaskManager 클래스가 제공하는 후크 메소드들을 사용하여 구현된 템플릿 메소드들이다.

[그림 8]에서 ConcreteMessageContainer와 ConcreteEventQueue가 각각 특정한 메시지 스케줄링 정책을 구현한 구체적인 서브클래스들이라고 가정하였을 때 커널 프레임워크의 메시지 전송 메커니즘은 다음과 같다.

어떤 태스크가 특정 메시지 큐 객체의sendMessage() 멤버 함수를 호출하면, 호출된 MessageQueue 클래스 객체는 EventQueue 클래스의 인터페이스를 통하여 isEmpty() 함수를 호출한다. 그러면 동적 결합에 의해 ConcreteEventQueue 클래스의 isEmpty() 함수가 호출되어 현재 메시지를 기다리고 있는 태스크가 있는지를 확인하게 된다. 메시지를 기다리 태스크가 있으면 EventQueue 클래스의 인터페이스를 통하여 ConcreteEventQueue 객체에게 findNext() 함수를 호출하여 사용되는 메시지 전달 방식에 따라 메시지를 전달할 다음 태스크를 찾는다. 그런 후 TaskManager를 통하여 찾은 태스크의 상태를 준비 상태로 바꾸면 모든 작업이 끝나게 된다. 메시지를 기다리고 있는 태스크가 없는 경우에는 MessageContainer 클래스의 인터페이스를 통하여 ConcreteMessageContainer 객체에게 putMessage() 함수를 호출하여 메시지를 저장하고 작업을 끝내게 된다.

태스크가 특정 메시지 큐 객체의receiveMessage() 멤버 함수를 호출하면 호출된 MessageQueue 클래스 객체는 MessageContainer 클래스의 인터페이스를 통하여 isEmpty() 함수를 호출하여 도착한 메시지가 있는지 확인한다. 메시지가 있으면 MessageContainer 클래스의

표 2 sendMessage()와 receiveMessage()

<table>
<tr><th>sendMessage(m)</th><th>receiveMessage()</th></tr>
<tr><td><pre>if (eventQueue->isEmpty())
  messageContainer->putMessage(m);
else {
  taskId = eventQueue->findNext();
  taskManager->setState(taskId, Ready);
}</pre></td><td><pre>if (messageContainer->isEmpty())
  return messageContainer->getMessage();
else
  eventQueue->addTask();</pre></td></tr>
</table>

인터페이스를 통하여 ConcreteMessageContainer 객체에게 getMessage() 함수를 호출하여 메시지를 태스크에게 전달하고 모든 작업을 끝낸다. 도착한 메시지가 없을 경우에는 EventQueue 클래스의 인터페이스를 통하여 ConcreteEventQueue 객체에게 addTask() 멤버 함수를 호출하여 메시지 수신을 요청한 태스크를 큐에 대기시키고 TaskManager를 통하여 태스크의 상태를 대기 상태로 바꿈으로써 작업을 끝내게 된다. 메시지 송수신 서비스의 결과로서 태스크의 상태 변화가 발생하면 컨텍스트 스위칭에 의해 다음에 수행할 태스크가 바뀔 수 있다.

4.4 세마포어 관리 설계 패턴

태스크들 사이의 동기화 서비스는 Semaphore 클래스를 통하여 이루어진다. 세마포어 관리는 메시지 관리와 유사한 설계 패턴을 갖는다. 다만, 메시지 관리에서는 메시지 자원을 보관하는 MessageContainer 클래스가 사용된 반면, 세마포어 클래스에서는 이에 대응하는 세마포어의 가용한 동기화 자원의 수를 나타내는 count가 세마포어 클래스의 속성으로 사용된다. 세마포어 관리에 관여한 클래스들의 설계 패턴은 [그림 9]와 같다.

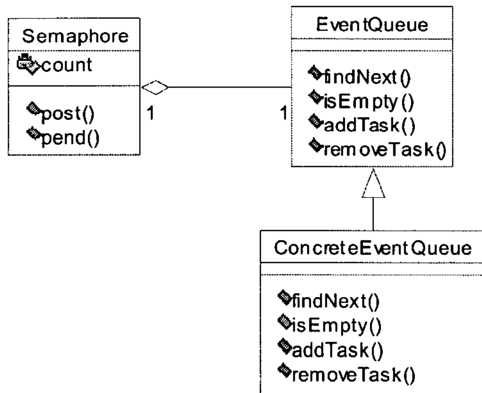

그림 9 세마포어 설계 패턴

[그림 9]에서 EventQueue는 세마포어에서 동기화 자원의 할당을 기다리는 태스크들의 정보를 관리하는 클래스로서, Semaphore 클래스에게 스케줄링에 필요한 인터페이스를 후크 메소드들로 제공한다. 즉, Semaphore 클래스의 동기화 스케줄링 알고리즘은 EventQueue의 서브클래스에서 구체적으로 정의된다. 또한, 메시지 큐에서 메시지를 전달 받을 태스크의 선정과 세마포어에서 동기화 자원을 전달 받을 태스크의 선정은 각각 메시지 관리 클래스의 EventQueue와 세마포어 관리 클래스의 EventQueue의 독립적인 확장을 통하여 서로 다른 스케줄링 정책을 지원하도록 구현할 수 있다.

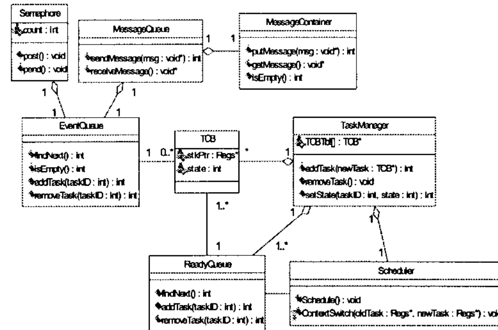

그림 10 커널 프레임워크의 설계 패턴

5. 커널 프레임워크의 확장을 통한 멀티태스킹 커널의 생성

4장에서 다룬 본 논문의 커널 프레임워크 설계 패턴들을 종합하여 이에 관여한 핵심 클래스들의 인터페이스와 이들의 상호관계로 요약하면 [그림 10]과 같다. [그림 10]의 클래스들은 커널이 관리하는 시스템 자원에 대한 스케줄링 정책과 커널이 작동될 하드웨어 환경에 따라 가변적인 부분들이 객체 합성과 클래스 상속을 통하여 개조와 확장이 용이하도록 설계되어 있다.

[그림 10]의 커널 프레임워크 클래스들 중, Semaphore와 MessageQueue를 제외한 나머지 클래스들은 프레임워크의 고정 부위와 확장 부위 사이의 인터페이스를 제공하는 가변 부위에 해당한다. 예를 들면, ReadyQueue, MessageContainer, TaskManager, 그리고 EventQueue 클래스는 특정 커널에서 사용하고자 하는 스케줄링 정책을 구현한 서브 클래스들로 확장된다. 또한, TCB와 Scheduler 클래스는 커널이 사용되어질 하드웨어에 따라 변경되어야 하는 부분을 포함하고 있다. 이들은 사용할 하드웨어에 맞는 레지스터 구조나 컨텍스트 스위칭 코드로 재정의한 서브 클래스들로 확장된다.

본 커널 프레임워크에 고정되어 설계된 부분과 본 커널 프레임워크를 사용하여 용도에 맞는 멀티태스킹 커널을 생성하기 위해서 확장, 추가되는 부분들은 [그림 11]과 같다.

[그림 11]에서 확장 부위에 속한 클래스들의 상위 클래스들이 프레임워크의 가변 부위를 형성한다. 프레임워

크로부터 확장되는 부분은 Strategy 패턴[7]과 Bridge 패턴[7]의 설계 구조로 이루어져 있다. 프레임워크가 이와 같이 확장되어 생성된 커널에서 제어의 흐름은 프레임워크의 템플릿 메소드들이 가변 부위의 클래스 인터페이스가 제공하는 후크 메소드들을 사용하여 확장된 서브 클래스의 멤버 함수들을 동적 결합에 의해 호출함으로써 이루어진다.

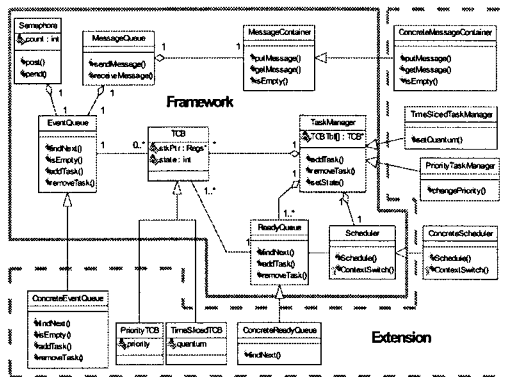

그림 11 커널 프레임워크의 고정, 가변 및 확장 부위

6. 확장된 멀티태스킹 커널의 동작

본 장에서는 본 논문의 커널 프레임워크가 확장된 커널에서의 제어흐름을 우선 순위 기반의 스케줄링 정책을 지원하는 멀티태스킹 커널로 확장된 경우의 객체들의 상호작용들을 예로 들어 설명한다. 먼저 태스크 생성에 관한 제어의 흐름은 [그림 12]와 같다.

[그림 12]에서 aTask객체가 TaskManager 클래스 인터페이스를 통하여 addTask() 멤버함수를 호출하여 태스크의 생성을 요청하면 동적 결합에 의해 PriorityTaskManager 클래스 객체의 addTask()가 호출된다. 호출된 PriorityTaskManager 객체는 TCB, ReadyQueue, Scheduler 클래스 인터페이스를 통하여 동적 결합된 PriorityTCB, ConcreteReadyQueue, ConcreteScheduler 클래스 객체들을 [그림 12]에서와 같이 순차적으로 호출한다. 예를 들면, PriorityTaskManager 객체가 ReadyQueue 클래스 인터페이스를 통하여 addTask(int)를 호출하면 동적 결합에 의하여 ConcreteReadyQueue 클래스의 addTask(int) 멤버 함수가 호출된다. ConcreteReadyQueue 클래스는 확장된 커널 프레임워크에서 사용하고자 하는 스케줄링 알고리즘에 필요한 준비 리스트의 구조를 갖도록 정의되어야 한다. 즉, TaskManager, ReadyQueue 그리고 Scheduler 클래스들은 Strategy 패턴에 따라 특정한 스케줄링 알고리즘을 구현한 서브 클래스들로 확장, 합성된다. ConcreteReadyQueue 클래스의 addTask(int)는 ConcreteReadyQueue 클래스의 구조에 맞게 준비 리스트에 태스크 정보를 삽입하는 역할을 한다.

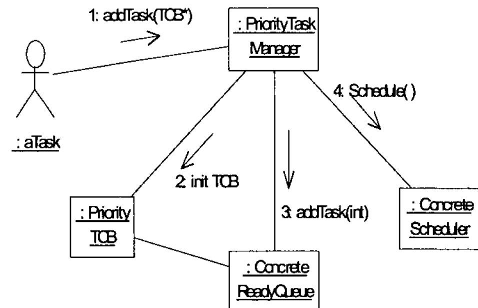

그림 12 확장된 멀티태스킹 커널에서의 태스크 생성

[그림 13]은 메시지를 기다리는 태스크가 존재하는 메시지 큐에 메시지를 전송할 경우의 제어 흐름의 일부을 나타낸 것이다.

여기서도 [그림 12]의 경우와 마찬가지로 Strategy 패턴이 적용되었다. 태스크들 사이의 메시지 전달을 담당하는 메시지 큐의 인터페이스의 sendMessage(void *) 호출을 통하여 ConcreteMessageQueue 클래스 객체가 메시지송신 요청을 받으면 ConcreteMessageContainer 클래스 객체에 현재 저장되어 있는 메시지가 있는지 확인하게 된다. MessageContainer 클래스의 서브 클래스인 ConcreteMessageContainer 클래스는 사용자가 필요로 하는 메시지 전달 방식을 지원할 수 있도록 메시지 큐의 구조를 정의하고 있다. isEmpty()의 결과, 메시지 큐가 비어 있으면 이는 현재 메시지를 기다리고 있는 태스크가 존재할 수 있으므로 ConcreteEventQueue 객체에 등록되어 있는 태스크들 중에서 메시지를 전달 할 태스크를 찾게 된다. 이는 findNext() 메시지에 의해 이루어 지며 이는 동적 결합에 의하여 ConcreteEventQueue 클래스의 findNext() 멤버 함수를 호출한다. ConcreteEventQueue 클래스는 메시지 전달 방식에 맞도록 설계되어 있는 EventQueue의 서브 클래스이다.

이처럼 멀티태스킹 커널의 기본적인 제어의 흐름은 커널 프레임워크에 모두 고정되어 있고 변화가 필요한 부분에 있어서는 객체 합성과 클래스 상속에 의해 필요

한 부분만을 개조, 합성하여 사용할 수 있도록 함으로써 커널의 다른 부분에 영향을 주지 않고 커널을 확장할 수 있다.

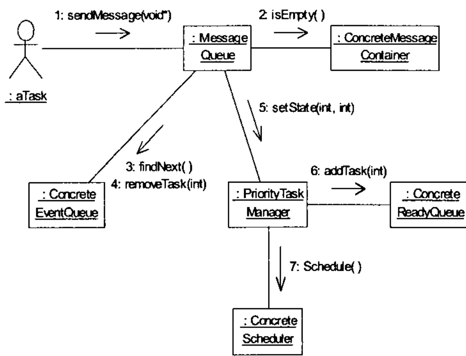

그림 13 확장된 멀티태스킹 커널에서의 메시지 전송

7. 연구 결과

본 절에서는 본 논문에서 연구된 멀티태스킹 커널 프레임워크가 커널의 효율적인 구현에 재사용될 수 있음을 확인하기 위하여 5절과 6절에 기술한 프레임워크 확장 방법에 따라 본 논문의 커널 프레임워크를 Intel 80x86 플랫폼을 지원하는 멀티태스킹 커널로 확장한 사례를 설명한다. 커널 프레임워크를 확장하여 구현된 커널은 uCOS[6]와 동등한 기능과 성능을 갖도록 하였다. μC/OS[6]의 경우에는 우선 순위 기반의 멀티태스킹 커널로 약 1000 NCSL로 이루어져 있다. 본 논문에서 설계한 프레임워크는 600NCSL로 이루어졌으며, 이를 확장하여 Intel 80x86 계열의 마이크로 프로세서에서 동작하며 메시지 큐만을 사용하고, 우선 순위의 변경이나 시스템 시간을 읽어 오는 등의 서비스는 제공하지 않지만 μC/OS와 같이 우선 순위 기반의 스케줄링을 지원하고 우선 순위 기반으로 메시지 및 세마포어를 전달하도록 확장했을 때, 1100 NCSL의 멀티태스킹 커널을 생성할 수 있다. 1100NCSL의 코드에서 600NCSL은 이미 개발된 프레임워크를 재사용했으므로 500NCSL만이 새로 개발된 것이다. 즉, 54.5%의 코드를 재사용한 것이다. 이러한 확장을 위해서 커널 설계자가 해야 하는 일은, 사용하고자 하는 알고리즘에 따라 자료 구조를 정의하고 그에 필요한 연산을 정의해야 하는 ReadyQueue, EventQueue, 그리고 MessageContainer 클래스와 Task 클래스의 일부분을 재정의하고, 사용될 하드웨어에 따라 변경되어지는 Task 클래스의 생성자와 스케줄링에 필요한 태스크 정보를 반환하는 함수, Scheduler 클래스, 그리고 인터럽트 관련 서비스를 수정 또는 추가하는 것이다.

표 3 μC/OS와 본 논문의 프레임워크의 비교

	μC/OS	본 논문의 프레임워크
CPU 스케줄링	고정(우선 순위 기반)	변경 가능
메시지 전달 방식	고정(우선 순위 기반)	변경 가능
세마포어 전달 방식	고정(우선 순위 기반)	변경 가능
전체 라인수 (8086, 우선순위 기반)	1000NCSL	1100NCSL (600NCSL 재사용)

기존의 실시간 커널들[8, 9]은 시스템 파라미터들이 허용하는 범위 내에서 재구성 가능하도록 설계되어 있다. 커널의 그 밖의 부분을 용도에 맞게 변경하기 위해서는 커널의 소스코드를 임의로 수정하여야 한다. 이에 비하여 본 논문의 커널 프레임워크는 커널 구현자가 객체 합성에 의한 블랙박스 재사용 방식과 클래스 상속에 의한 화이트박스 재사용 방식[10]을 병용하여 효율적이고 일관성있게 커널을 개조, 확장할 수 있도록 설계되었다. 본 커널 프레임워크를 확장하여 구현된 커널은 응용 프로그래머에 의해 객체 합성과 시스템 파라미터 설정을 통한 블랙박스 방식으로 실시간 내장 응용 시스템의 개발에 사용될 수 있다.

한편, 재사용성이나 재구성성이 고려된 멀티태스킹 클래스 라이브러리[11, 12]나 객체지향 프레임워크로 설계된 운영체제[13]들도 소개되어 있다. [11]이나 [12]의 멀티태스킹 클래스 라이브러리의 경우에는 일반적으로 한가지 태스크 스케줄링 방식만을 지원하도록 되어 있으며 특정 용도로의 수정이 불가능하도록 되어 있다. 또한 사용될 하드웨어에 따른 변경 또한 불가능하다. 객체지향 운영체제의 경우에는 용도에 따른 변경이나 다른 하드웨어로의 이식은 자유로우나 내장형으로 사용하기에는 너무 규모가 크다는 단점이 있다. 반면 본 논문에서 구현한 프레임워크 형태의 커널은 사용될 하드웨어나 알고리즘에 따라 일부를 수정함으로써 쉽게 새로운 커널을 생성할 수 있도록 하였다

본 커널 프레임워크는 객체지향 프레임워크 설계 패턴들[4, 7]을 기반으로 하여 설계되었다. 특히, 커널의 가변 부위들을 고정 부위로부터 분리하여 확장 가능한 추상 클래스들로 설계하는데 유용한 설계 패턴들이 주

로 적용되었다. 하드웨어 환경에 의존적인 부분은 Bridge 설계 패턴에 따라 설계되었다. 스케줄링 정책에 따라 가변적인 부분은 Strategy 패턴에 따라 설계되었다. 긴밀한 상호 작용을 해야 하는 커널 클래스들의 결합 구조는 Facade 패턴과 Mediator 패턴에 따라 설계하여 클래스들의 상호 의존도와 결합도를 완화하였다.

본 커널 프레임워크는 커널이 사용하는 메모리 영역이 정적으로 할당되도록 설계되었다. 즉, 본 프레임워크로부터 만들어진 커널에서의 클래스 생성자(constructor)들은 객체의 생성에 컴파일러에 의해 할당된 메모리 영역을 사용하며 생성된 객체에 필요한 초기화 작업들을 추가적으로 수행하게 된다. 그러나 생성된 객체들에 대한 참조에는 포인터가 사용되었다. 따라서, 실행 시 상호 작용하는 커널 객체들의 바인딩은 동적으로 이루어진다. 이와 같이 정적 메모리 할당과 동적 바인딩 방식을 함께 사용함으로써 본 프레임워크로부터 생성되는 커널의 효율성과 유연성에 균형을 유지하도록 하였다. 커널이 사용할 메모리 영역의 크기는 커널에 포함될 클래스들의 선택과 선택된 각 커널 클래스의 인스턴스 최대 개수를 지정하는 시스템 파라미터들을 통하여 컴파일러에 의해 결정된다.

8. 결론 및 향후 연구 과제

본 논문에서는 실시간 내장 응용 프로그램의 개발에 필요한 멀티태스킹 커널로 쉽게 개조, 확장할 수 있는 커널 프레임워크의 프로토타입을 개발한 사례를 소개하였다. 본 논문에서는 내장 멀티태스킹 커널을 쉽게 변경, 확장 할 수 있도록 함으로써 커널의 재사용성을 높이고자 하였다. 즉, 커널의 복잡한 제어 흐름을 고정적인 부분에 한하여 프레임워크로 구성함으로써 일부 소스 코드뿐만 아니라 제어의 흐름 등과 같은 커널 도메인의 분석 결과를 재사용할 수 있도록 하였다.

본 커널 프레임워크는 멀티태스킹 커널들에 고정적인 부분들과 새로운 커널을 생성할 때 변경되어야 하는 부분들이 추상 클래스들로 분리된 객체지향 프레임워크 설계 패턴으로 설계되었다. 따라서 상속과 동적 바인딩의 개념을 이용하여 하드웨어 의존적인 부분과 알고리즘 의존적인 부분을 커널의 용도에 맞도록 쉽게 수정할 수 있다. 커널의 설계자는 복잡한 커널의 제어 흐름과 구조를 고려 할 필요 없이 CPU 스케줄링이나 메시지 전달 방식과 같은 특정 부분만을 고려함으로써 쉽게 원하는 기능을 수행하는 커널을 생성해 낼 수 있게 된다.

본 논문에서 개발한 커널 프레임워크는 스케줄링 정책과 하드웨어 환경에 의존적인 부분들이 객체 합성과 클래스 상속 메커니즘에 따라 효율적이고 일관성있게 개조, 확장될 수 있다. 따라서 본 커널 프레임워크는 기존의 상용 멀티태스킹 커널들보다 작은 규모를 가져야 하면서 높은 이식성과 개조성이 요구되는 응용 분야의 마이크로 프로세서 기반의 실시간 내장형 멀티태스킹 커널의 효율적인 구현에 효과적으로 사용될 수 있을 것으로 기대된다.

본 논문의 커널 프레임워크에서는 TCB를 사용하는 클래스들이 TCB의 friend 클래스로 선언되어 있어서 TCB에 대한 직접적이고 빠른 접근을 허용한다. TCB는 하드웨어 플랫폼과 스케줄링 정책에 따라 가변적인 클래스이다. 따라서 TCB는 커널의 생성 시 특정 하드웨어와 스케줄링 정책에 적합한 서브클래스로 확장, 변경된다. 이 때, 이에 영향을 받는 TCB의 friend 클래스들도 확장된 TCB와 상호 일관성있는 서브클래스들로 함께 확장되어야 한다. 현재는 TCB와 관련된 일부 클래스의 확장이 이러한 사항을 충분히 반영하지 못하고 있다. 따라서 본 커널 프레임워크를 Abstract Factory[7]와 같은 유연한 객체 생성 설계 패턴들을 적용하여 커널 객체들이 특정 하드웨어 플랫폼과 스케줄링 정책에 따라 상호 일관성있게 생성될 수 있도록 설계할 필요가 있다.

본 커널 프레임워크가 지원하는 시스템 서비스는 경량 멀티스레딩(lightweight multi-threading)과 이에 필요한 CPU, 타이머, 메시지, 세마포어 자원의 관리, 그리고 인터럽트의 처리이다. 메모리 관리, 디바이스 드라이버, 네트워킹 등은 본 프레임워크에 포함되어 있지 않다. 향후, 본 커널 프레임워크는 메모리 관리와 네트워킹 서비스를 선택적으로 추가할 수 있도록 확장할 예정이다. 특히, Corba와 인터넷 웹 분산 환경에서 작동 가능한 실시간 내장 멀티태스킹 커널의 구현에도 재사용 가능하도록 확장할 예정이다.

참 고 문 헌

[1] R. Moore, "How to Use Real-Time Multitasking Kernel In Embedded Systems," Micro Digital Associates, Inc., 1995

[2] J. F. Bortolotti, et al., "RTMK : A Real-Time MicroKernel," Dr. Dobb's Journal, May 1994

[3] R. E. Johnson, "Frameworks = (Components + Patterns)," Communications of the ACM, Oct. 1997, pp. 39-42.

[4] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995

[5] H. A. Schmid, "Systematic Framework Design by Generalization," Communications of the ACM, Oct.

1997, pp. 48-51
[6] J. J. Labrosse, μC/OS The Real-Time Kernel, R&D Publications, 1992
[7] E. Gamma, et al., Design Patterns : Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
[8] "RTOS Buyer Guide," http://www.realtime-info.be/encyc/market/rtos/rtos.htm
[9] "rtos free or must have resources," http://www.eg3.com/real/freertos.htm
[10] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," Communications of the ACM, Oct. 1997, pp. 32-38.
[11] K. Gibson, "C++ Multitasking Class Library," Dr. Dobb' s Journal, May 1994, pp. 28-34
[12] J. A. Joins, "Multitasking Classes," http://www.eos.ncsu.edu/eos/info/ie/ie307_info/www/guide.dir/node42.html
[13] R.H. Campbell, N. Islam, P. Madany, "Choices, Frameworks and Refinement," Computing Systems, 5(3), 1992
[14] T. Quatrani, Visual Modeling with Rational ROSE and UML, Addison Wesley Longman, Inc., 1998
[15] J. Rumbaugh, et al., The Unified Modeling Language Reference Manual, Addison-Wesley, 1999

이 준 섭
1997년 고려대학교 전산학과 학사. 1999년 고려대학교 대학원 전산학과 석사. 1999년 3월 ~ 현재 한국전자통신연구원 연구원. 관심분야는 객체지향 소프트웨어 공학, 소프트웨어 프레임워크, 디자인 패턴, 웹 어플리케이션 개발 방법론, XML/ EDI 등.

전 태 웅
1981년 서울대학교 계산통계학과 학사. 1983년 서울대학교 계산통계학과 석사. 1992년 Illinois Institute of Technology 전산과학 박사. 1983년 ~ 1987년 금성통신 연구소 주임연구원. 1992년 ~ 1995년 LG산전 연구소 책임연구원. 1995년 ~ 현재 고려대학교 자연과학부(전산학 전공) 부교수. 관심분야는 소프트웨어 테스팅, 소프트웨어 아키텍쳐, 객체지향 프레임워크, 실시간 소프트웨어 공학

이 승 룡
1978년 고려대학교 재료공학과 학사. 1987년 12월 Illinois Institute of Technology 전산학 석사. 1991년 12월 Illinois Institute of Technology 전산학 박사. 1992년 ~ 1993년 Governors Srtate University 조교수. 1993년 ~ 현재 경희대학교 전자정보학부 (전자계산공학 전공) 부교수. 관심분야는 실시간 컴퓨팅, 멀티미디어 시스템