# DOS 환경 로봇제어기용 실시간 운영체제를 위한 멀티태스킹 커널의 설계 및 구현

# A Design and Implementation of DOS-Based Multitasking Kernel of the Real-Time Operating Systems for Robot Controller

장 호, 이 기 동
(Ho Jang and Ki-Dong Lee)

**Abstract** : In order to implement the real-time operating systems for robot controller, this paper proposes a systematic method for implementing the real-time kernel under the DOS environment. So far, we designed the robot control software and its own operating system simultaneously. Though robot operating systems have simple structure, it allows the developer to have a surplus time and effort to implement complete robot systems. In addition to this, in most cases of this type, operating systems does not support multitasking function, thus, low level hardware interrupts are used for real-time execution. Subsequently, some kinds of real-time tasks are hard to implement under this environment. Nowadays, the operating systems for robot controller requires multitasking functions, intertask communication and task synchronization mechanism, and rigorous real-time responsiveness. Thus, we propose an effective and low costs real-time systems for robot controller satisfying the various real-time characteristics. The proposed real-time systems are verified through real implementation.

**Keywords** : real-time operating systems, real-time kernel, scheduling, intertask communication, synchronization, context switching, robot control software

## I. 서론

일반적으로 실시간 시스템이란 시스템에 주어진 입력으로 발생되는 출력에 대해서 입력부터 출력까지 소요되는 반응시간이 일정한 한계치(deadline)보다 작을 것이 요구되는 시스템을 말한다[1]. 여기에서 주의해야할 사항은 실시간 시스템이라고 해서 항상 가장 빠른 시스템인 것은 아니라는 점이며, 좁은 의미에서 반응시간이 제한시간을 초과하는 경우 정상적으로 동작하지 않는 시스템을 지칭한다. 즉, 정기적으로 발생되는 데이터를 실시간으로 처리하거나 비정기적으로 발생되는 사건(event)에 대한 신속한 대응이 실시간 시스템의 가장 큰 목표이다. 예를 들면, 항공기제어시스템의 경우에는 주기적이고 시의 적절한 응답성을 요구하며, 핵발전소의 경우에는 위기상황에 대한 빠른 응답성이 필수적이고, 항공기예약시스템의 경우에는 그다지 빠른 응답성이 필요한 것은 아니다. 대부분의 제어시스템은 이와 같은 실시간 반응성을 필요로 하는 실시간 시스템으로 구성되어야 한다. 그 중에서도 로봇 제어 시스템은 반응 제한시간이 아주 짧은 실시간 시스템이라고 볼 수 있다.

과거의 로봇 제어기용 소프트웨어 개발에서는 간단한 운영체제를 직접 디자인하여 사용하였으므로 상당히 많은 시간과 노력이 요구되었을 뿐만 아니라, 대개의 경우 구현된 운영체제가 실시간 멀티태스킹을 지원하지 않으므로 응용 작업에서 발생할 수 있는 여러 가지 이벤트들의 실시간 처리와 다양한 작업을 구현하기 위해서는 하드웨어 인터럽트를 빈번하게 이용하여야 했다. 이는 제어를 위한 응용 프로그램 작성을 힘들게 했으며, 실시간의 특성을 갖고 있는 제어기의 기능을 여러 면에서 제약하는 요인이 되었다.

이에 오늘날에는 실시간 특성을 요구하는 제어 소프트웨어의 구축을 용이하게 하기 위해 상용의 실시간 운영체제 개발환경을 많이 이용한다. 즉, 내장형(embedded) 시스템인 로봇제어기를 실시간 운영체제로 만들어 직접적인 관심분야 이외에 대한 노력을 줄이고 로봇시스템 특유의 다양한 기능을 구현하는데 중점을 두고 있다. 한편 기존의 상용 실시간 운영체제들의 개발환경은 주로 UNIX와 같이 필요 이상으로 규모가 큰 시스템에서 동작하는 것[2-4]이 대부분이며 개발 환경 자체가 다양한 실시간 시스템을 구현하기 위하여 만들어진 것이기 때문에 성능 면에서 예상외로 상당히 큰 오버헤드(overhead)를 가지고 있다. 따라서 로봇제어기의 특징만을 살린 최적의 실시간 운영체제로 볼 수 없으며 경우에 따라서는 적정한 로봇제어기를 구현하기 힘들 수도 있다. 또한 오늘날 개인용 컴퓨터가 로봇제어기의 태스크 스케줄링 및 플래닝(planning)을 위해서 사용되는 경향이 많으므로 도스(DOS) 환경 하에서의 로봇용 실시간 운영체제의 용도가 확대되고 있다.

기존의 상용 실시간 개발 도구는 로봇시스템을 위한 개발 도구가 아니기 때문에 실제의 실시간성을 제공하는데 있어서 로봇시스템 전용의 개발 도구 보다 많은 오버헤드를 가지며 대부분의 상용시스템은 개발 환경이 워크스테이션급의 환경을 요구한다. 이러한 문제점을 해결하기 위하여 로봇제어기 전용의 간략화된 실시간 커널을 개인용 컴퓨터 환경에서 제공한다면 로봇시스템 개발에 도움이 될 수 있을 것이다.

본 논문의 구성은 2장에서 실시간 시스템의 개념을 살펴보고, 3장에서는 실시간 커널을 설계하며 4장에서는 3장에서 설계된 실시간 커널을 실제 구현하여 로봇제어기용 시스템에 응용하여 본다. 마지막으로 5장에서는 본 시스템의 기대 효과와 앞으로의 개발 방향을 결론과 함께

접수일자 : 1996. 9. 17. 수정완료 : 1997. 2. 28.
장 호 : 영남대학교 전산공학과
이기동 : 영남대학교 전산공학과
※ 본 논문은 1995년도 교육부지원 한국학술진흥재단의 신진교수과제 학술연구조성비에 의하여 연구 되었습니다.

기술한다.

## Ⅱ. 실시간 시스템

실시간 시스템은 관점에 따라 세 가지로 분류할 수 있다[3]. 첫째, 반응 시간 면에서 경성(硬性)실시간(hard real-time)시스템과 연성(軟性)실시간(soft real-time)시스템으로 나뉘며 둘째, 호환성 면에서 폐쇄형(proprietary) 시스템과 개방형(open)시스템으로 나뉜다. 마지막으로 구조에 따라 집중형(centralized)시스템과 분산형(distributed) 시스템으로 나뉜다. 그림 1은 일반적인 실시간 시스템의 계층 구조를 나타내며, 여기에서의 주된 요소는 실시간 스케줄링(scheduling)을 포함한 실시간 커널(kernel)이다.

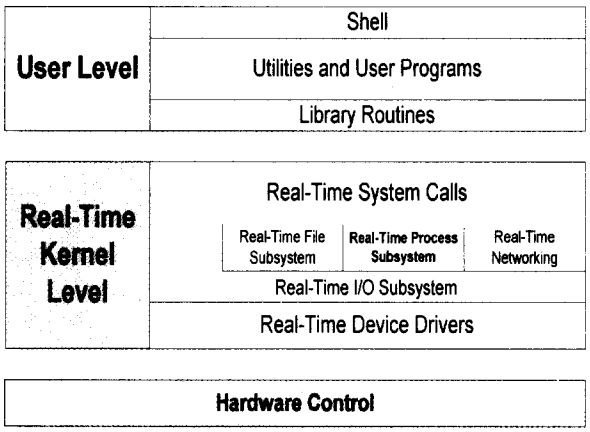


그림 1 실시간 시스템의 계층 구조.

Fig. 1. Hierarchical structure for real time systems.

일반적으로 커널이라 함은 태스크 스케줄링 기능과 태스크 디스패칭(dispatching) 기능 그리고 태스크간의 통신(intertask communication) 기능 및 동기화(synchronization)를 동시에 제공하는 운영체제의 가장 핵심적인 구성 요소를 말한다. 여기에 일반적인 스케줄링 기법 대신 실시간성을 내포한 스케줄링 기법을 대치한 경우를 실시간 커널이라고 한다. 이와 같은 특징을 가진 실시간 운영체제는 단시간 내에 많은 발전을 거듭하여 시장성을 가진 상용 시스템뿐만 아니라 연구 기간의 특수용도 및 범용에 이르는 다양한 시스템들이 개발되었다. 먼저 상용의 실시간 시스템들은 다음의 표 1에 나타나 있다. 이 중에서 VRTX, VxWorks, pSOS+, Nucleus, VMEexec등이 널리 쓰이고 있으며, 이들과 함께 연구용 시스템들도 필요성에 의해 활발히 제작되고 있다. 이들 제품은 여러 가지 형태로 제작되어 있는 경우도 있지만 대부분은 범용이면서 UNIX와 같이 규모가 큰 운영체제를 기반으로 한다. 이로 인해 특수한 실시간 작업을 위해서는 예상외의 오버헤드를 감수하여야 한다. 그러므로 본 연구에서는 로봇 제어에 반드시 필요한 최소한의 기능만을 중심으로 실시간 커널을 제작한다. 실시간 커널의 주요 구성요소들에 대하여 좀더 자세히 살펴보면 다음과 같다.

### 1. 태스크간의 통신과 동기화

태스크간의 통신 방법은 먼저 동일한 전역 변수(global variable)를 통해 데이터를 전달하는 방법을 생각할 수 있지만 이 방법은 데이터의 충돌을 방지하기 위해 임계 영역(critical section)을 구성해야한다. 임계 영역은 동기화 메커니즘을 통해 구현될 수 있는데 이는 디스에이블-인에이블 구조(disable-enable scheme)와 선점형 카운팅 세마포어(counting semaphore)를 사용한 블록킹(blocking)

표 1. 상용 실시간 운영체제.

Table 1. Commercial products of real-time operating systems.

| Company | Product |
|---|---|
| Accelerated Technology, Inc. | Nucleus RTX, Nucleus PLUS |
| A.T.Barrett and Associates | RTXC, RTXC/MP |
| Byte-BOS Integrated Systems | Byte-BOS |
| Eyring | PDOS, VMEPROM |
| Integrated Systems | pSOS+ |
| JMI Software Consultants | C EXECUTIVE |
| Kadak | AMX Multitasking Exec. |
| Lynx | LynxOS |
| Micro Digital | SMX |
| Microware Systems | OS-9, OS-9000 |
| Microtech | VRTX |
| RTMX-UniFLEX | UniFLEX |
| Spectron Microsystems | SPOX |
| U.S.Software | Multi Task! C Exec. |
| Wind River Systems | VxWorks |

세마포어 기법을 사용한다. 이 방법 외에도 원형 문자 버퍼(circular character buffer)의 조작을 통해서 메시지를 전달하는 방법이 있으며, 이는 메일 박스(mailbox)를 사용한다[5,6]. 본 연구에서는 위의 방법들을 모두 지원한다.

먼저 전역변수 방법은 간단하고 속도가 빠르다는 장점을 가지고 있는 반면에 높은 우선 순위의 태스크가 적절하지 못한 시간에 낮은 우선 순위의 태스크를 선점할 경우 전역 데이터를 훼손시킬 염려가 있다. 이를 위하여 데이터 버퍼(data buffer)를 이용한다. 즉, 데이터를 생산하는 태스크와 소비하는 태스크 사이에 버퍼를 마련하여 시간적인 완충 역할을 하게 한다. 이때의 버퍼는 스택이나, 구조화되지 않은 변수들의 집단과 같은 다양한 데이터 구조를 가질 수 있다.

메일 박스는 메시지 교환용으로 사용되며 관리자(supervisor) 태스크를 포함하는 태스크 제어 블록 모델에 기초한 시스템에 적합하다. 기본적인 오퍼레이션에는 게시(post)와 보류(pend), 그리고 수신(accept)이 있다. 메일 박스는 각 태스크들과 그에 필요한 자원들, 그리고 그 자원들의 상태를 리스트화한 테이블을 유지한다. 시스템 콜이나 하드웨어 인터럽트에 의해 관리자 태스크는 메일 박스 내에서 보류 상태의 태스크가 있는지 검사한다. 이때 자원의 상태가 사용 가능하면 그 태스크는 재 시작한다. 또한 교착상태(deadlock)를 방지하기 위해 보류 오퍼레이션에 타임아웃(timeout) 기능을 포함시킨다.

큐(queue)는 메일 박스의 배열로 구현될 수 있으며, 공유 장치의 장치서버(device server)를 구성하는데 편리하다. 이때는 장치에 대한 서비스 요구를 받아들이기 위해 링버퍼(ring buffer)를 사용하고 링버퍼에 대한 접근을 제어하기 위해 링버퍼의 머리와 꼬리에 각각 큐를 사용한다. 이는 스풀(spool) 프로그램이나 데몬(daemon)과 같은 장치 제어 소프트웨어 구성에 유용하다.

임계 영역을 보호하기 위해 세마포어라는 특별한 변수를 사용한다. 임계 영역을 보호하기 위한 락(lock) 역할의 메모리 위치를 세마포어로 정의하고 이를 $S$라고 한다. 여기에 웨이트(wait)와 시그널(signal)이라는 두 가지 오퍼레이션을 행한다. 웨이트($\mathbf{P}(S)$) 오퍼레이션은 세마포어($S$)가 거짓이 될 때까지 프로그램 호출을 보류시키며, 시그널($\mathbf{V}(S)$) 오퍼레이션은 세마포어($S$)를 거짓으로 설정한다. 하나 이상의 프로세스에서 임계 영역을 보호하는 것을 이진(binary) 세마포어라고 하며 공유 자원을 보호

하거나 사용되지 않은 자원의 개수를 유지하기 위해 카운팅 세마포어를 사용하기도 한다.

임의의 사건 발생은 뒤따르는 프로세스가 반응적으로 동작하도록 항상 적절한 플래그를 설정한다. 사건 플래그(event flag)의 설정은 제어의 흐름을 운영체제로 넘기고 적절한 핸들러(handler)를 기동시킨다. 이때 사건 발생을 기다리고 있는 태스크를 블록(block) 상태라고 한다. 대부분 사건 플래그는 예외 처리 핸들러(exception handler)나 인터럽트 핸들러(interrupt handler) 등에 사용된다.

## 2. 실시간 스케줄링 기법

대부분의 실시간 시스템들에서는 실시간 처리를 요구하는 다수의 태스크들을 선점형(preemption)으로 멀티태스킹(multitasking) 하기 위해서 특별히 고안된 스케줄링 기법들을 사용하고 있다. 만일 완전한 선점형 스케줄링이 이루어지지 않는다면 엄격한 실시간 반응 시간을 얻지 못하여 시스템의 정확한 동작을 기대할 수 없다. 그림 2는 비선점형(nonpreemption) 스케줄러(a)와 선점형 스케줄러(b)의 비교를 나타내며[3], 다음의 실시간 스케줄링 기법들은 선점성을 내포하고 있다.

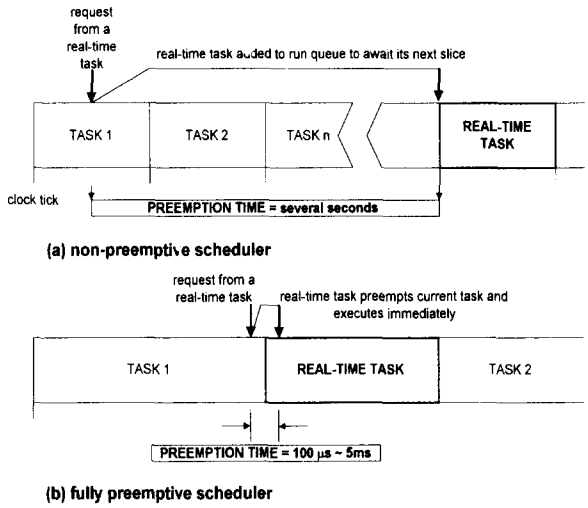


그림 2. 비선점형 스케줄러와 선점형 스케줄러의 비교.

Fig. 2. Non-preemptive scheduler and preemptive scheduler.

### 시간분할 스케줄링(time-slice scheduling)

실시간 멀티태스킹을 위해 일정한 간격으로 시간을 분할하여 분할된 시간마다 각각의 태스크들을 일정량씩 실행한다. 태스크들을 실행하는 순서는 라운드 로빈(round-robin) 방식을 많이 사용하며, 여기에 태스크의 성질에 따라 우선 순위를 각각 부여하여 그 우선 순위에 따른 라운드 로빈 스케줄링도 가능하다. 이 방법은 연성 실시간 시스템에 자주 사용되며, 규칙적이고 비교적 긴 반응 시간을 요구하는 태스크들을 후면(background)으로 스케줄링할 때 적합하다[5].

### 우선 순위 스케줄링(priority scheduling)

각각의 태스크에 주어진 우선 순위를 기반으로 라운드 로빈 방식에 의해 스케줄링을 행한다. 이 방법에는 기본적으로 처음부터 부여된 우선 순위를 바꿀 수 없는 고정 우선 순위(fixed-priority/static-priority) 스케줄링과 실시간 처리 과정 중에 상황에 따라 우선 순위를 재 할당하는 동적 우선 순위(dynamic-priority) 스케줄링의 두 가지 방식이 있다[6]. 동적 우선 순위 방법은 기존의 실시간 로봇 제어기 소프트웨어 설계에서 많이 사용되어 온 방법이다[7].

### 고정표본시간 스케줄링(fixed-sample-time scheduling)

시간을 동일한 크기로 분할하지 않고 태스크의 성질에 따라 알맞은 크기로 분할하여 할당한다. 다시 말하면, 모든 태스크에 동일한 크기의 시간을 할당할 경우, 그 시간 내에 완전히 서비스를 받지 못하는 태스크들은 정해진 시간 내에 결과를 도출하지 못하므로 폐기될 수밖에 없다. 이를 해결하기 위해 실시간 처리 프로그램의 태스크 정의 단계에서 각 태스크에 적절한 표본 시간을 기술하고, 이 표본 시간의 크기에 상응하는 개별적인 소프트웨어 타이머를 이용하여 스케줄링한다[5].

### 최소잔여시간우선 스케줄링(shortest-task-first scheduling)

고정표본시간 스케줄링 기법에서 각기 다른 표본 시간으로 인해 초래되는 오버헤드를 줄이기 위해서 고안된 방법이다. 기본적으로 분할된 시간에 의한 스케줄링을 하면서 표본 시간 종료점으로부터의 상대적 차이가 가장 작은 태스크가 현재 서비스를 받고 있는 태스크를 보류(suspend)시킴과 동시에 선점적으로 서비스 권리를 획득하여 정해진 시간 내에 실행을 마치게 된다[3, 8, 9]. 그림 3에서는 고정 우선 순위 방법과 최소 잔여 시간 우선 방법의 원리를 도시하였는데, 10ms의 문맥 전환 시간을 가지는 시스템에서 최소 실행 시간이 10ms이고 표본 시간이 20ms인 A 태스크와 최소 실행 시간이 25ms이고 표본 시간이 50ms인 B 태스크를 (a)와 (b)에서는 고정 우선 순위 방법으로 (c)에서는 최소 잔여 시간 우선 방법으로 스케줄링한 것을 보여준다. 이때, (a)의 경우 B 태스크의 우선 순위가 A 태스크의 우선 순위보다 높기 때문에 A 태스크는 항상 마감 시간을 지나쳐서 서비스를 받게되며, (b)의 경우는 A 태스크의 우선 순위가 B 태스크의 우선 순위보다 높기 때문에 B 태스크는 항상 A 태스크에 의해 선점을 당하므로 마감 시간을 넘어서 실행을 완료하게 된다. 이렇듯 고정 우선 순위 스케줄링은 최악의 경우 실시간성을 보장할 수 없게 되므로 이를 최소 잔여 시간 우선 스케줄링으로 보완하는 것이 바람직하다. 즉, (c)에서는 우선 순위에 상관없이 마감 시간에 가까운 태스크가 높은 우선 순위를 가지는 효과와 같으므로 경우에 따라서 항상 적절한 우선 순위를 부여받아 마감 시간 내에 실행을 완료하게 된다.

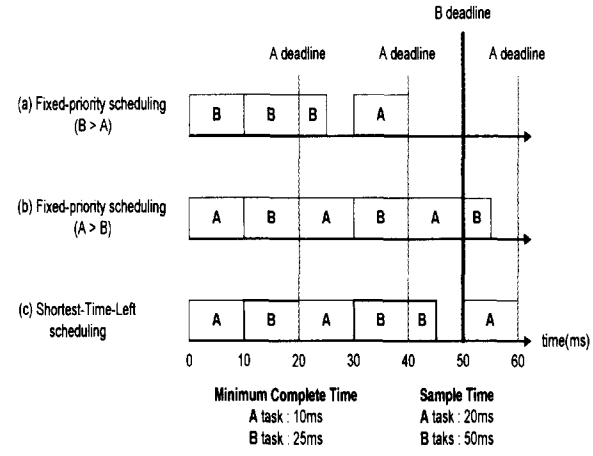


그림 3. 고정 우선 순위 스케줄링과 최소 잔여 시간 우선 스케줄링.

Fig. 3. Fixed priority scheduling and shortest-task-first scheduling.

사건본위 스케줄링(event-driven scheduling)

위의 스케줄링 기법들은 주기적인 태스크 서비스를 전제로 하고 있는 반면에 사건 본위 스케줄링은 비주기적인 태스크 발생을 위해 사용된다. 그러므로 이 방법은 센서를 이용하여 피드백 제어를 행할 경우, 센서를 통해 추출된 데이터에 의해 유발되는 태스크의 스케줄링에 적합하다[5].

3. 태스크 디스패칭과 문맥교환(context switching)

일반적으로 멀티태스킹 환경에서의 태스크들은 네 가지의 상태 중 하나를 가지게 된다. 즉, 실행(executing)상태, 보류(suspended)상태, 준비(ready)상태, 소멸(dormant)상태 등인데 그 상태들이 주어진 조건하에서 천이 하면서 규정된 스케줄링 기법에 따라 멀티태스킹을 수행한다. 문맥 교환은 태스크가 시스템에 서비스를 요구할 경우와 타이머 인터럽트가 발생했을 경우, 그리고 외부 인터럽트 신호가 발생했을 경우에 일어나는 실행상의 변화이다. 이 때 현재 실행중인 태스크는 나중에 실행을 재개하기 위해서 보류되는데 이는 태스크 스택에서 시스템 스택으로의 교환을 수반하며 스택상의 인자들을 시스템 영역으로 전달하는 복잡한 작업이 요구된다. 또한 전체 반응 시간 중에서 문맥 교환이 차지하는 비중이 크므로 시스템 타이머의 주파수를 향상시킴으로써 문맥 교환의 기회를 많이 부여함과 동시에 문맥 교환에 소요되는 시간을 감소시켜야 한다.

## Ⅲ. 실시간 커널의 설계

본 연구의 커널 스케줄링 시스템은 스케줄러와 디스패처의 두 부분으로 나뉘며, 스케줄러는 주어진 태스크들의 형태에 따라 실행 순서를 결정하고, 디스패처는 결정된 순서에 의해 태스크들을 실행한다. 그리고 스케줄링 방법은 제어 작업의 특성 및 태스크의 형태에 따라 적절한 방법들을 선택하여 사용한다. 다음은 실시간 커널의 제작에 있어서 반드시 필요한 사항들이다.

```
struct _task {
    int t_id;                       /* 태스크 id 번호 */
    char t_name[TNAMELEN];          /* 태스크 이름 */
          -------- 실행 제어 데이터 --------
    int t_pri;                      /* 유효 우선 순위 */
    int t_reg[NOREG];               /* 태스크 실행 상태 */
    int t_flags;                    /* 플래그 */
    void (*t_func)(); /* 태스크 쉘에 의해 실행되는 루틴 */
    void *t_argp; /* 태스크 아규먼트 구조 지정 포인터 */
          ------ 태스크 형태와 상태 표시 ------
    int t_tasktype;                 /* 태스크 타입*/
    int t_state;                    /* 태스크 상태 코드 */
          -------- 태스크 제어 패러미터 --------
    long t_tleft;                   /* 실행까지 남은 시간 */
    long t_tsamp;                   /* 샘플링 시간 */
          ------ 실행시간 스택 할당 정보 ------
    int *t_usrstack;    /* 할당된 스택 메모리의 포인터 */
    unsigned t_ustacksize; /* 할당된 사용자 스택 크기 */
    struct _task *t_next; /* 다음 태스크에 대한 포인터 */
};
typedef struct _task TASK;
```

그림 4. 태스크의 구조.

Fig. 4. Task structure.

그림 4에 태스크의 구조가 제안되어 있는데 태스크의 형태에는 스케줄링 방식에 대한 정보가 포함되어 있고, 또 태스크의 상태를 나타내는 필드는 크게 실행상태, 준비 상태, 인터럽트에 의해 보류된 상태, 대기 상태, 소멸 상태 등을 나타내며, 세분화하여 입출력에 대한 대기 상태, 어떤 사건에 대한 대기 상태, 세마포어에 의한 블록 상태, 다른 태스크에 의해 보류된 상태 등을 표시할 수 있다. 태스크의 이름은 최대 16자이고 상주할 수 있는 태스크는 최대 16개이다. 태스크의 정의 함수는 *NewTask()*이고, 이를 이용하여 태스크 구조에 대한 모든 정보들을 정의한다. 마지막으로 태스크 종료 함수는 *ExitTask()*이다.

태스크 큐는 태스크 구조내의 *t_next* 필드에 의해 스케줄러에서 단일 연결 형태의 큐로 사용된다. 태스크 큐의 구조는 다음 그림 5에 나타나 있으며, 스케줄링 방법에 따라 다수의 큐들을 규정된 구문에 맞추어 시스템 내에서 생성시킬 수 있다. 표 2는 시스템에 정의되어 있는 5가지의 큐들을 나타낸다. 태스크들은 큐 내에서 기본적으로 잔여 시간에 의해서 정렬되어지나 잔여 시간에 의해 정렬되어질 수 없는 특별한 예외 상황에는 우선 순위를 기준으로 정렬된다. 처음부터 잔여 시간이 아닌 우선 순위를 기준으로 스케줄링하기를 원한다면, 태스크 정의 단계에서 태스크들의 잔여 시간을 동일하게 설정하면 된다. 일단 큐 내에 정렬되어진 태스크들은 그 순서에 의해 스케줄링된다. 또한 각 태스크 큐들의 식별을 위해 16비트의 고유 번호를 가지고 있다. 태스크 큐를 위한 함수에는 새로운 큐를 생성시키는 *NewTqueue()*, 큐의 선두 포인터를 돌려주는 *HeadTqueue()*, 선두에 태스크를 진입시키는 *StackTqueue()*, 후미에 태스크를 추가하는 *PutTqueue()*, 큐의 선두에서 태스크를 끄집어 내는 *GetTqueue()*, 특정 키(key)를 갖고 있는 태스크를 진입시키는 *EnterTqueue()*, 큐를 정렬하는 *SortTqueue()*등이 있다.

```
typedef struct _tqueue {
        int tq_id;                  /* 큐 id 번호 */
        TASK *tq_head;  /* 큐 시작을 나타내는 포인터 */
        TASK *tq_tail;  /* 큐 끝을 나타내는 포인터 */
}TQUEUE;
```

그림 5. 태스크 큐의 구조.

Fig. 5. Structure of task queue.

표 2. 시스템 큐의 종류.

Table 2. Types of system queue.

| | |
|---|---|
| **CwaitQueue** | 대기 상태 태스크 큐 |
| **ReadyQueue** | 실행 준비 태스크 큐 |
| **CntrlQueue** | 고정 표본 시간 스케줄링 태스크 큐 |
| **SliceQueue** | 시간 분할 스케줄링 태스크 큐 |
| **SuspdQueue** | 보류 상태 태스크 큐 |

태스크간의 통신 기능의 구현에 있어서 데이터 버퍼는 원형 큐나 링 버퍼로 구성되어 있다. 여기에는 큐의 선두(front)와 후미(rear)가 있으며 선두의 포인터는 큐에서 첫 번째 데이터가 위치한 바로 앞을 가리키는 반면 후미는 큐의 마지막 데이터 위치를 가리킨다. 또한 PUT 오퍼레이션(*putcbuf()*)의 내용은 *((rear + 1) % buffer_size) = data이며, GET 오퍼레이션(*getcbuf()*)의 내용은 data = *((front + 1) % buffer_size)이다. 이 외에도 새로운 버퍼를 생성시키는 함수인 *newcbuf()*와 버퍼의 개수를 세는 함수인 *countcbuf()*, 그리고 버퍼를 초기화하는 함수로 *resetcbuf()*가 있다. 메시지 전달은 *NewMessage()*에 의해 할당되고 *PostMessage()* 함수에 의해 메시지가 전달

된다. 전달된 메시지는 *AcceptMessage()* 함수에 의해 수신된다. 이들 함수는 비 블록킹(nonblocking) 송수신 함수이다. 즉, 블록킹 송신은 수신자가 메시지를 수신하여 통신이 완전히 종결된 것이 확인되기까지 송신자가 기다려야하는 반면, 비 블록킹 송신에서는 수신자가 메시지를 받기 전이라도 송신자가 이를 확인하는 과정을 거치지 않고 바로 다른 프로세싱을 계속할 수 있게 해준다. 이를 위해 수신자가 언제 준비되어 이미 송신된 메시지를 가져갈지 알 수 없으므로 그때까지 송신자로부터 받은 메시지를 저장할 메시지 버퍼링 기법이 사용된다. 즉, 앞에서 기술한 메일 박스 메카니즘에 메시지의 생산률에서 소비율을 뺀 값에 버퍼가 가득 차는 시간을 곱한 값에 대응되는 크기의 데이터 버퍼를 설치하여 메시지의 전송을 관리한다. 그리고, 예상하지 못한 요인으로 버퍼가 가득 차는 최악의 경우에는 버퍼를 초기화한다. 블록킹 송신은 동기식 통신이고, 비 블록킹 송신은 비동기식 통신의 한 예이다. 비 블록킹 송신을 사용한 비동기식 통신은 병행성(concurrency)의 수준을 증가시킨다[10]. 세마포어는 *NewSem()* 함수에 의해 할당되며, *Psem()*와 *Vsem()* 함수는 사용자 영역에서 호출된다. 반면에 커널 영역에서는 *Psemaphore()*와 *Vsemaphore()* 함수를 호출한다.

구현된 여러 가지 스케줄링 방법에서 시간분할 스케줄링은 일정하게 분할된 시간을 가진 라운드 로빈 방식 하에서 태스크를 스케줄링한다. 이때 사용되는 태스크 큐는 SliceQueue이며 여기에서 정렬된 순서대로 *RrbDispatch()* 함수에 의해 태스크를 디스패치한다. 다음으로 *RrbStart()* 함수가 라운드 로빈 디스패쳐 시스템의 스케줄링을 시작한다. 최소잔여시간 스케줄링은 CntrlQueue를 사용하며 여기에서 정렬된 순서대로 *StlDispatch()* 함수에 의해 태스크를 디스패치함과 동시에 스케줄링한다. 또한 스케줄링을 위한 타이머를 초기화하는 *StlStart()* 함수와 샘플링 순간을 대기하기 위한 *StlWait()* 함수가 있다. 그 외에도 큐 내에 있는 태스크들의 태스크 제어 블록내의 *t_tleft* 값을 감소시키는 *dec_tleft()* 함수, 다음 샘플링 인스턴스 *t_next*에 시간을 할당하는 *set_tnext()* 함수, 그리고 다음 태스크에 대해 재스케줄링(rescheduling)하는 *next_task()* 함수가 있다. 사건 본위 스케줄링에는 사건 디스크립터 노드(event descriptor node)를 할당하는 *NewEvent()* 함수와 사건을 보류 상태로 전환시키는 *PendEvent()* 함수, 그리고 사건의 발생을 알리는 신호를 만들어 내는 *NotifyEvent()* 함수 등이 있다.

디스패처는 스케줄링 방법에 의해 결정된 순서에 따라 태스크 큐로부터 태스크를 추출하여 실행한다. 여기에는 주 함수인 *Dispatch()*와 타이머에 의해 호출되는 디스패쳐인 *TimeDispatch()* 함수, 그리고 현재 실행될 순서의 태스크를 실행하는 *Execute()* 함수가 있다. 그리고, *TimeDispatch()* 함수 내부에 *StlDispatch()*와 *RrbDispatch()* 함수가 포함되어 있어서 스케줄링 방법에 따라 선택적으로 사용되며, 이때 태스크 큐들은 표 2에 나타난 바와 같이 그 시점의 태스크 상태에 의해 적절하게 선택되어진다. 결과적으로 태스크 구조에서 태스크 상태를 나타내는 *t_state* 필드 값에 의해 그림 6와 같이 상태 천이가 발생된다.

문맥 교환 코드는 시스템의 반응 시간을 최소한으로 하기 위하여 어셈블리 언어로 작성하였다(*kernel.asm*). 본 실시간 커널에서는 두 가지 형태의 엔트리(entry)를 인식하는데, 시스템 호출과 인터럽트가 이에 해당된다. 시스템 호출은 세마포어상의 보류(*pend()*)와 같은 서비스를 요구하는 도구로써 어셈블리 언어로 작성된 *syscall* 함수에 의해 수행되고, 인터럽트는 소프트웨어 인터럽트를 통

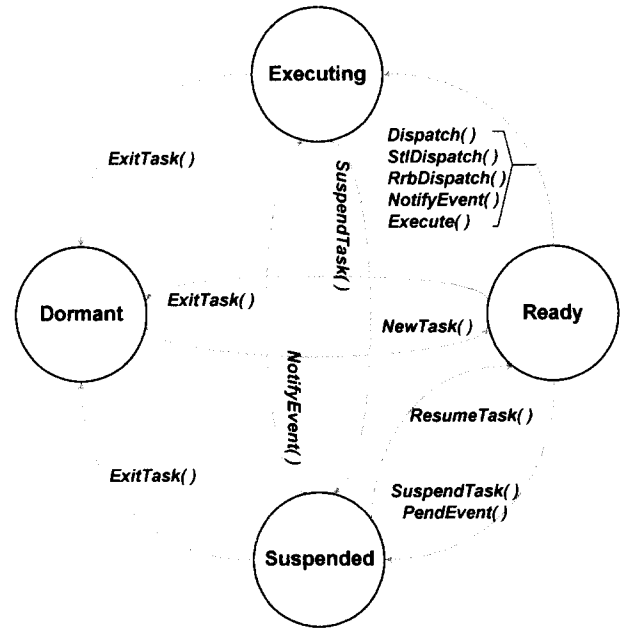


그림 6. 실시간 태스크의 상태 천이도.

Fig. 6. State transition diagram for real time tasks.

해 제어권을 시스템으로 전달하기 위해서 게이트웨이(*gateway*)를 호출하는 인터럽트 서비스 루틴에 의해 실행된다. 여기에서 게이트웨이는 소프트웨어 인터럽트로 구현된다. 또한 *syscall*은 시스템 호출 인자들을 처리하는 게이트웨이에 대한 전위(front-end) 함수로서의 역할도 수행한다. 게이트웨이는 현재 태스크 구조상의 *t_reg[]* 필드에 표시된 태스크 스택을 포함하여 태스크의 모든 현재 상태를 저장한다. 문맥 교환에 소요되는 시간을 감소시키기 위하여 80x87 수치 연산 보조 처리기의 존재를 검사하여 만약 존재한다면 그 상태까지도 태스크 스택에 저장하도록 하였다. 다음으로 게이트웨이는 태스크 스택에서 시스템 스택으로 인자들을 복사하고 시스템 엔트리 포인트(system entry point)로 간접 호출을 행함으로써 실행상의 문맥을 시스템 영역으로 전환한다. 후에 호출 복귀가 이루어지면 *runuser()* 함수에 의해 현재 태스크는 디스패치된다. 즉, 수치 연산 보조 처리기의 상태를 복원(restore)하고 태스크 스택으로 전환하여 현재 태스크에서 문맥 교환이 발생한 시점으로부터 남아 있는 뒷부분의 실행을 재개(resume)한다. 문맥 교환에서 주의해야 할 점은 시스템 영역에서 실행 중인 오퍼레이션에 대해서는 인터럽트가 디스에이블(disable)되어야 한다는 점이다. 즉, 커널 자체는 인터럽트가 불가능하다. 이는 인터럽트 지연시간(latency)과 외부 사건에 대한 반응시간의 최소화에 제약을 가하지만, 커널 함수가 완벽하게 재진입(reentrance)을 실현하지 못했을 경우에도 커널을 단일화할 수가 있다.

도스에서는 초당 최대 18.2회(≈55㎳)의 클락 틱(clock tick)이 발생되지만 로봇 제어용 소프트웨어에서는 이보다 높은 횟수의 클락 틱이 필수적이다. 왜냐하면 클락 틱의 발생률은 시스템의 성능과 직결되기 때문이다. 또한 IBM-PC에서는 3개의 카운트다운 타이머(countdown timer)가 있지만 프로그램에서 사용할 수 있는 타이머는 단 하나뿐이며, 그 외는 동적 기억장치 리프레쉬(dynamic memory refresh)와 소리(sound) 생성에 사용된다. 이를 보완하기 위하여 두개의 소프트웨어 타이머를 추가하여

하드웨어 타이머에 의해 인터럽트 되면서 클락 틱 발생률을 1ms정도로 정밀화하였다.

기본적으로 C 언어에서 제공되는 입출력 함수들은 재진입(reentrance)이 불가능하므로 BIOS를 호출하여 재진입이 가능하도록 설계된 루틴들을 어셈블리 언어로 작성해야 할 필요성이 있다. 즉, 이들 함수들도 실행 중에 문맥 교환이 일어날 수 있도록 하여 실시간 반응 시간을 최소화하며, 실시간 스케줄링이 태스크 코드(code) 단위로 이루질 수 있도록 하여 실시간 소프트웨어 제작의 신뢰도 및 유연성을 향상시킨다. 예를 들면, *sprintf()* 함수 대신에 주어진 속성(attribute)으로 문자열을 출력하는 *Video_WriteString()* 함수를 사용한다. 이와 같은 함수에는 화면의 명시된 위치에서 문자를 그 속성과 함께 읽어들이는 *Video_ReadChar()* 함수, 주어진 속성으로 명시된 위치에 문자를 출력하는 *Video_WriteChar()* 함수, 명시된 위치에 주어진 색깔로 점(pixel dot)을 찍는 *Video_WritePixel()* 함수, 그리고 BIOS의 비디오 스크롤링(scrolling) 서비스를 행하는 *Video_Scroll()* 함수 등이 있다. VRTX에서도 *sc_putc()* 함수와 *sc_getc()* 함수가 있는데 이들은 각각 문자를 출력하고 입력한다. *sc_getc()* 함수는 64바이트 버퍼에 문자를 입력받는데 버퍼가 완전히 비어 있으면 문자가 도착할 때까지 보류 상태를 유지한다. 그러나 이 함수는 출력 장치에 문자를 나타내지는 않으므로 특정 인터럽트 서비스 루틴을 이용하여 화면에 문자를 나타낼 수 있다. *sc_putc()* 함수도 역시 64 바이트 버퍼를 사용하는데 버퍼가 완전히 차 있으면 한 문자가 빠져나갈 때까지 보류 상태를 유지한다.

## Ⅳ. 실시간 커널의 구현 및 응용

본 연구에서의 커널은 로봇 제어기 소프트웨어용이므로 반응 시간이 1ms에서 10ms사이의 중앙 집중식 경성 실시간 시스템에 맞도록 제작하였다. 태스크의 정의에서부터 스케줄링 기법, 타이머의 설계, 문맥 교환, 메모리 관리 및 간단한 사용자 인터페이스 등을 지원하는 함수들을 모아서 하나의 커널 라이브러리를 작성하고 그 외의 네트워크(network) 기능과 실제 하드웨어 제어를 위한 입출력(input-output) 기능은 제어 시뮬레이션 단계에서 필요하지 않기 때문에 구현에서 제외한다. 다음으로, 작성된 라이브러리를 이용하여 제어 프로그램 작성시 필요한 실시간 기본형을 함수 단위별로 기술하고 컴파일 단계에서 커널 라이브러리와 링크 시키면 실행 가능한 실시간 제어 코드가 생성된다. 이는 실시간 제어 프로그래밍이 쉬워지면서 시간이 절약되는 결과를 가져온다. 그림 7은 제작 과정을 도식화한 것이다. 제작 환경은 도스 6.0이 탑재된 100MHz의 IBM-PC 호환기종이며 마이크로소프트 C/C++ 버전 7.0 및 매크로 어셈블러 6.0 컴파일러를 사용하여 앞서 기술한 사항들을 중심으로 실시간 커널 라이브러리를 제작하였다. 이때 대부분의 모듈은 C 언어로 기술되지만, 세마포어와 스택 조작 및 커널의 기본적인 기능부는 어셈블리 언어로 기술하였다. 최소잔여시간 스케줄링을 위해 먼저 CntrlQueue를 마련하고, 최소잔여시간 스케줄러인 *StlDispatch()* 함수를 구현하였다. 이때 *WaitNextSamp()*와 *StlWait()*에 의해 CntrlQueue의 순서가 갱신되므로 적절한 스케줄링이 이루어진다. 실시간 제어기 소프트웨어 프로그래머는 *SetCntrlTask()*로 태스크를 정의함으로써 *NewTask()* 함수를 호출함과 동시에 STF 스케줄링 효과를 얻을 수 있다. 시간분할 스케줄링도 역시 SliceQueue의 순서에 의해 스케줄링되며 *SetBkgndTask()* 함수에 의해 조작된다. 사건 본위 스케줄링에서는 사건 발생이 *NotifyEvent()* 함수에 의해 사건

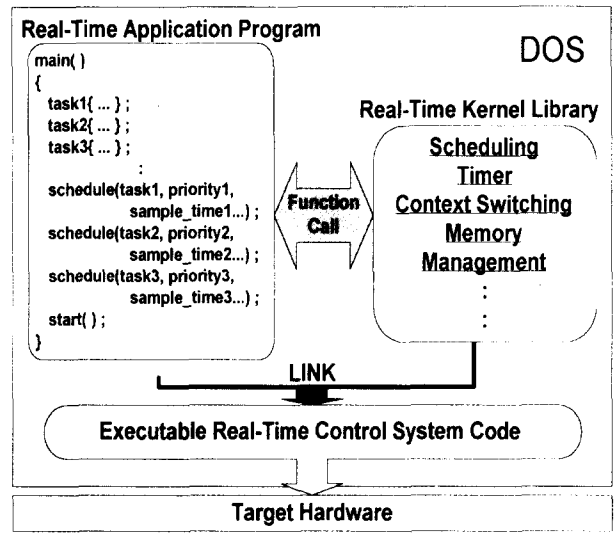


그림 7. 구현된 실시간 시스템의 구조.

Fig. 7. Structure of presented real-time systems.

관리자에게 통보되며 동시에 실행 중이던 태스크는 *PendEvent()* 함수에 의해 ReadyQueue에 들어가게 된다. 이는 다시 *Resume()* 함수에 의해 실행을 재개하게 된다. 이를 위해 *SetEventTask()* 함수가 준비되어 있다. 실시간 커널 구현 결과 문맥 교환에 소요되는 시간은 클락 틱 1000회 정도이며 0.1ms에서 1ms사이의 값으로 나타났다. 기본적으로 실시간 병행 처리가 가능한 최대 태스크 개수를 16개로, 시스템 버퍼 크기는 64바이트로, 태스크의 기본 우선 순위는 1000으로 설정한다. 여기에서 시스템의 전체 속도 증가를 위해 컴파일 단계에서 스택 검사를 하지 않도록 하며, 커널의 코드들을 완전한 재진입 코드로 구현하여 실시간 병행 처리의 신뢰도를 향상시켰다.

완성된 커널의 성능을 측정하기 위해 이 커널을 이용하여 간단한 로봇 제어시스템을 구현한 다음 실시간 시스템에서 중요한 요소중의 하나인 태스크 전환 시간을 측정하여 본다. 적용 로봇은 6축 다관절형이며, 로봇의 기구학(kinematics)은 데너빗-하텐버그(Denavit-Hartenberg) 방식으로, 역기구학(inverse kinematics)은 기하학적 접근법으로 정해진 시간 내에 해를 도출하도록 구현하였다. 그림 8과 같이 시스템 클락이 1ms로 발생되는 환경에서 대표적인 태스크들의 표본 시간을 다음과 같이 설정한다. 예를 들어 기구학 태스크는 30ms, 역기구학 태스크는 30ms, 그리고 시스템의 현재 클락을 표시하는 태스크는 1sec로 설정하여 우선 순위 스케줄링과 최소잔여시간 스케줄링 방법을 혼합하여 멀티태스킹한다. 추가적으로 키보드의 상태를 검사하여 ESC키가 눌러졌을 때 전체 시스템을 정지시키는 태스크를 사건본위 스케줄링 방법으로 시뮬레이션 한다. 로봇 제어기 소프트웨어의 커널을 작성하는 과정과 완성된 커널의 성능 측정에서 가장 주된 태스크인 기구학 태스크들간의 태스크 전환 시간을 측정하여 이 값이 신뢰 구간을 만족하는지를 검사하기 위해 시뮬레이션 소프트웨어의 주 기능부를 그림 9와 같이 기술하고 이를 제작된 실시간 커널 라이브러리와 링크하여 실행 코드를 생성한다. 생성된 제어기 소프트웨어에서 로봇 제어를 위해서는 필수적인 로봇 기구학 계산 태

스크들이 정확한 해를 표본시간내에 출력하는 상황하에서의 태스크 전환에 걸리는 시간을 측정한 결과 이들 태스크의 전환 시간이 0.2ms로 나타났다. 그러므로, 이 경우에 한해서는 완전한 선점형 실시간 커널이라고 할 수 있다.

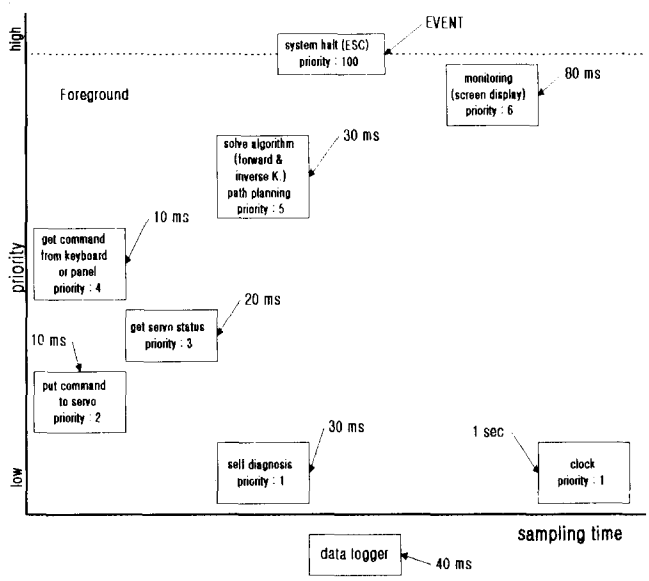


그림 8. 로봇 제어 태스크의 정의.
Fig. 8. Tasks for robot control.

```
main( )
{ prologue( ) ;                    /* 커널 초기화 */
   /*--태스크 정의 및 스케줄링 전략 명시--*/
SetCntrlTask_S("CLK",clock,(void*)NULL,
        PCTRL,CLK_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("DIAGNOSIS",diag,(void*)NULL,
        PCTRL,DIAG_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("PUT_COM",put_com,(void*)NULL,
    PCTRL+1,PUT_COM_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("SERVO",servo,(void*)NULL,
        PCTRL+2,SERVO_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("GET_COM",get_com,(void*)NULL,
    PCTRL+3,GET_COM_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("F_KINEMAT",forward_kine,
        (void*)&d_fg1,PCTRL+4,
        FORWARD_TSAMP,SMALL_STACK);
SetCntrlTask_S("I_KINEMAT",inverse_kine,
        (void*)&d_fg2,PCTRL+4,
        INVERSE_TSAMP,SMALL_STACK) ;
SetCntrlTask_S("MONITOR",monitor,(void*)NULL,
        PCTRL+5,MONIT_TSAMP,SMALL_STACK) ;
SetBkgndTask_S("FILER",filer,(void*)stdout,
        PSLICE+10,SMALL_STACK) ;
escape = SetEventTask_S("ESCAPE",escape_handler,
        (void *)NULL,PCTRL+99,SMALL_STACK) ;
Intset(INT_KB, kbsig) ;
  /*-------------------------------------*/
start( ) ;      /* 스케줄러 기동 */
return(-1) ; }
```

그림 9. 주 프로그램 리스트.
Fig. 9. Main program.

## V. 결론

본 연구에서는 현재 로봇 제어용 소프트웨어 개발분야에서 절실히 요구되는 실시간 처리 기법을 구현하기 위하여 단일 태스크 처리 기반인 도스 운영체제상에서 실시간 커널을 구현하는 체계적인 방법을 제안한다. 과거의 로봇 제어기용 소프트웨어 개발에서는 간단한 운영체제를 직접 디자인하여 사용하였으므로 상당히 많은 시간과 노력이 요구되었을 뿐만 아니라, 대개의 경우 구현된 운영체제가 실시간 멀티태스킹을 지원하지 않으므로 하드웨어 인터럽트를 빈번하게 이용하여야 했다. 이는 제어를 위한 응용 프로그램 작성을 힘들게 했으며, 실시간의 특성을 갖고 있는 제어기의 기능을 여러 면에서 제약하는 요인이 되었다. 따라서 본 논문에서는 위의 특성들을 충분히 반영함과 동시에 저가의 비용으로 효율적인 실시간 시스템을 구성하고 이를 실제 프로그램 작성에 적용함으로써 본 시스템의 효용성을 검사하였다. 이와 같은 로봇제어기용 실시간 운영체제를 구현하기 위한 실시간 커널의 이용으로 로봇에 여러 가지 다양한 기능을 쉽게 구현 할 수 있다.

그러나 일종의 소프트웨어 인터럽터의 사용이기 때문에 실시간 반응성에 대한 충분한 검증이 있어야 하며 기존의 상용화된 제품과의 성능비교가 필요하다. 또한 네트워크를 통한 분산 실시간 환경과 이를 이용한 분산 실시간 시스템의 다양한 응용 분야에 적합한 최적의 스케줄링 기법에 관한 연구가 필요하다.

## 참고문헌

[1] Stuart Bennett, *Real-Time Computer Control : An Introduction*, Prentice Hall International, 1994.

[2] Brad Adelberg et al., Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers, *Stanford Univ. Department of Computer Science Technical Report CS:TN 94-8,* April, 1994.

[3] Borko Furht et al., *Real-Time UNIX systems : design and application guide*, Kluwer Academic Publishers, 1991.

[4] Hassan Gomma, *Software Design Methods for Concurrent and Real-Time Systems*, SEI series in software engineering, Addison-Wesley, 1993.

[5] D. M. Auslander and C. H. Tham, *Real-Time Software for Control*, Prentice Hall International, 1990.

[6] Krithi Ramamritham et al., "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no.1, pp.55-67, January, 1994.

[7] Sang H. Son, "Advances in real-time systems," *Prentice Hall International*, 1995.

[8] Phillip A Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, IEEE Computer Society Press, 1993.

[9] F. Panzieri and R. Davoli, "Real time systems : a tutorial," *Technical Report UBLCS-93-22*, October 1993.

[10] H. M. Deitel, *An Introduction to Operating Systems, Second Edition*, Addison-Wesley, 1990.

## 부 록

표 3. 태스크 동기화와 스케줄링을 위한 함수(1).
Table 3. Functions for task synchronization and scheduling(1).

| 함수개요 | 파라미터 |
|---|---|
| TASK *NewTask(name, func, argp, tasktype, priority, stacksize, quantum)<br>char *name;<br>void (*func)(), *argp;<br>int tasktype, priority;<br>long quantum;<br>unsigned stacksize; | name : 태스크 이름<br>func : 태스크의 실행되는 루틴<br>argp : 아규먼트 구조 포인터<br>tasktype : 태스크 타입<br>priority : 초기 실행 순위<br>stacksize : 태스크 스택의 필요한 워드 단위의 크기<br>quantum : 표본 시간 |
| void ExitTask(void) | 없음 |
| int NewSem(n)<br>int n; | n : 초기 세마포어 개수 (정상상태:양수, 그외:-1) |
| void Psem(n)<br>int n; | n : 세마포어 번호 |
| void Vsem(n)<br>int n; | n : 세마포어 번호 |
| void Psemaphore(id)<br>int id; | id : 세마포어 번호 |
| void Vsemaphore(id)<br>int id; | id : 세마포어 번호 |
| int NewMessage(void) | 작은 양의 정수인 메시지 핸들러 반환 |
| int PostMessage(mh, size, datum)<br>int mh, size;<br>void *datum; | mh : 메시지 핸들러<br>size : 메시지 블럭의 바이트 단위의 크기<br>datum : 하나의 메시지 데이터 포인터 |
| int AcceptMessage(mh,size,datum)<br>int mh, size;<br>void *datum; | mh : 메시지 핸들러<br>size : 전송된 데이터의 바이트 단위의 크기<br>datum : 하나의 메시지 데이터 포인터 |

표 4. 태스크 동기화와 스케줄링을 위한 함수(2).
Table 4. Functions for task synchronization and scheduling(2).

| 함수개요 | 파라미터 |
|---|---|
| void RrbDispatch(sliceq, timer)<br>TQUEUE *sliceq;<br>int timer; | sliceq : 시분할 태스크 큐<br>timer : 사용중인 타이머 |
| void RrbStart(rrbq, rrbtimer)<br>TQUEUE *rrbq;<br>int rrbtimer; | rrbq : 라운드로빈 태스크 큐<br>rrbtimer : 라운드로빈 스케줄링을 위한 타이머 |
| void StlDispatch(stlq, urgentq, stltimer)<br>TQUEUE *stlq, *urgentq;<br>int stltimer; | stlq : STF 태스크 큐<br>urgentq : 급하게 서비스 받아야 하는 대기 태스크들의 큐<br>stltimer : STF 스케줄링을 위한 타이머 |
| void StlStart(stlq, urgentq, stltimer)<br>TQUEUE *stlq, *urgentq;<br>int stltimer; | stlq : STF 태스크 큐<br>urgentq : 급하게 서비스 받아야 하는 대기 태스크들의 큐<br>stltimer : STF 스케줄링을 위한 타이머 |
| void StlWait(stlq)<br>TQUEUE *stlq; | stlq : STF 태스크 큐 |
| int NewEvent(void) | 정상상태 : 양수<br>그외 : -1 |
| void PendEvent(id, task)<br>int id;<br>TASK *task; | id : 사건 핸들러 번호<br>task : 새로운 태스크 포인터 |
| void NotifyEvent(id, action)<br>int id, action; | id : 사건 핸들러 번호<br>action : notifyONE, notifyALL |
| void Dispatch(void) | 없음 |
| void TimeDispatch(arg)<br>void *arg; | arg : 아규먼트 구조를 가리키는 포인터(정수형) |
| void Execute(void) | 없음 |


**장 호**

1994년 영남대 전산공학과 졸업. 동대학원 석사(1996년). 1997년 ~ 현재 영남대학교 전산공학과 박사과정. 관심분야는 분산 실시간 운영체제 및 실시간 제어시스템.

**이 기 동**

1985년 서울대 제어계측공학과 졸업. 동대학원 석사(1987년), 동대학 박사(1994년). 1995년 ~ 현재 영남대학교 전산공학과 조교수. 관심분야는 로봇 제어시스템 및 지능제어.