論文2003-40CI-4-5

소규모 임베디드 시스템을 위한 우선 순위 기반 라운드 로빈 스케줄링 운영체제의 설계 및 구현

(Design and Implementation of The Priority based Round Robin Scheduling Operating System for Compact Size Embedded System)

南 相 曄 * * *, 李 相 杭 * *, 朴 仁 政 *

(Sang Yup Nam, Sang Won Lee, and In Jung Park)

요 약

디지털기술과 반도체기술의 발전은 모든 전자제품의 발전을 가지고 왔다. 특히 이동 통신과 컴퓨터의 발전은 전자제품은 물론이고 가전제품까지도 네트워크화와 고성능화를 가져오게 하였다. 따라서 기존의 전자제품들이 단순제어와 독자적인 동작에서 벗어나서 복잡한 제어와 네트워크에 접속이 되며, 원격으로 조종되는 기능이 추가가 되어지고 있다. 따라서 내장형 기기(Embedded System)는 예전의 단일 Task의 단순 루프 제어 방식과는 달리 다중 Task의 실시간 처리가 필요하게 되었다. 이에 따라서 중·소형 기기에도 실시간 운영체제의 필요성이 대두되고 있다. 본 논문에서는 소형 내장형 기기를 위한 실시간 운영체제를 설계하고 구현하였다. 소형 내장형 기기는 자원과 CPU의 성능에서 한계가 있기 때문에 기존의 상업용 운영체제의 기능 중에서 가장 필요한 기능을 중심으로 구현하였다.

Abstract

In This paper, the operating system using priority based round robin scheduling system is designed and implemented. Using this scheduler, Real-Time operation is possible because High priority Task is running first and the other Task is running in parallel. Also Intertask Communication, Device Driver and operating system suitable for using the compact sized embedded system were implemented. Therefore this Operating system provides efficient and rapid implementation for the compact sized embedded system application.

Keywords : 스케줄링, 임베디드 시스템, 운영체제, 실시간

1. 서 론

현재의 디지털 기술은 급격히 발전하고 있다. 기존가전제품의 디지털화가 급격히 이루어지고 있고, 또한, 이동통신의 발달로 언제 어디서나 필요시 인터넷 검색, Email 수신, 정보 검색 등이 가능하게 되었다. 이로 인해 PDA를 비롯한 개인용 정보 단말기가 개발되고 상용화되고 있으며, 점차 기능도 진화하고 있다.

따라서 이러한 기능의 추가와 처리속도의 증가로 임베디드 분야의 프로그래밍도 갈수록 복잡해지고, 코드의 재사용이나 관리가 힘들어지고 있다. 이제까지의 단

* 正會員, 京文大學 情報通信科
(Dept. of Information and Communication, Kyeongmoon College)
** 正會員, 電子部品研究院
(Korea Electronics Technology Institute)
* 正會員, 檀國大學校 電子工學科
(Dept. of Electronics Engineering, DanKook Univ.)
接受日字:2002年7月2日, 수정완료일:2003年6月18日

일 Task 시스템은 더 이상 진화하는 임베디드 장비의 프로그래밍을 하기에는 힘들게 되었다.

이러한 단점을 보완하기 위해서 기본적인 자원의 관리와 타스크(Task)의 관리, 디바이스의 관리 등을 전담하고, 하드웨어 기반이 바뀌더라도 동일한 함수 인터페이스를 제공하는 상용 Operating System Software 가 나오게 되었다. 이러한 상용 OS는 기본적으로 다수의 고성능 프로세서 지원과 관리, 주변 자원의 관리, Network Interface Library, Graphic User Interface Library 등을 지원하여 개발자로 하여금 빠른 시간 내에 효과적으로 임베디드 장비의 프로그래밍이 가능하도록 지원하고 있다[1-9].

하지만 이러한 OS는 몇 가지 단점을 가지고 있다. 높은 개발비용, 개발 후 License fee 그리고 사용자가 필요로 하는 정보를 얻기가 매우 힘들다. 또한 이러한 운영체제들은 고성능의 프로세서를 중심으로 작성이 되었기 때문에 저 사향 CPU에서는 운용하기가 힘든 상황이다. 반면, 저 사향용으로 작성된 운영체제들은 일반적인 목적이 아니라 단일한 제어나 관리를 목적으로 작성이 되었기 때문에 목적하는 시스템 중심으로 작성이 되어 범용으로 사용하기 힘든 상황이다. 따라서 연구목표는 저 사향(8~16bit) Controller 용 OS, Preemptive Multitasking Kernel, Task Management, Priority Base Scheduling, Round-Robin Scheduling, Intertask Communication, Device Driver의 구현을 목표로 한다.

Ⅱ. 임베디드시스템 에 대한 운영체계 이론 배경

운영체제는 응용프로그램과 시스템 사이를 분리하고, 추상화하여 하드웨어를 단순하고 사용하기 간편하게 해준다. 이러한 추상화는 하드웨어가 교환이 되어도 운영체제에서 동일한 인터페이스를 제공하기 때문에 응용프로그램이 최소한의 변경으로 다른 시스템에 사용이 가능하다는 장점이 있다. 따라서 현재와 같은 다양한 시스템이 존재하는 상황에서는 이러한 점이 매우 중요하게 부각이 되고 있다.

이러한 운영체제 중 산업용으로 많이 쓰이는 방식이 실시간 운영체제이다. 이 운영체제는 기본적으로 운영체제의 역할은 동일하게 수행하지만 원하는 작업을 정해진 시간에 완료하는 것을 보장해야 한다. 실시간 시스템은 Soft Real-Time system과 Hard Real-Time system이다. Hard Real-Time system은 정해진 시간에 작업의 결과가 절대적으로 나와야 하는 경우에 사용하게 된다. 주로 군사용이나 의료용 등의 결과를 보장해야 하는 곳에 사용된다. Soft Real-Time system은 이보다 덜 치명적인 상황에서 사용하게 된다. 따라서 운영체제의 설계는 사용될 시스템과 상황, 그리고 자원 등을 모두 고려해야 한다.

1. Task Management

컴퓨터의 작업단위로서 사용되는 Task는 일련의 합쳐진 컴퓨터 조작을 의미하고 주로 Task 나 Thread라는 용어를 사용한다. 각 Task는 CPU register Set을 가지고 있으며, 자신의 Stack 영역을 가지고 있다. Multitask 운영체제는 Task가 동시에 여러 개가 실행이 될 수 있다. 하지만 CPU의 입장에서는 한번에 하나의 Task 만 실행이 될 수 있다. 따라서 각 Task는 Scheduler에 의해서 번갈아 가며 실행이 된다. 이러한 전이를 Context Switching이라고 한다. 우선순위를 기반으로 한 시스템에서는 각 Task가 우선순위를 가지고 있어서 이 Scheduler는 Context Switching시에 우선순위가 높은 Task를 먼저 실행할 수 있다. 따라서 각 타스크는 중요도에 따라서 실행 순위가 결정된다.

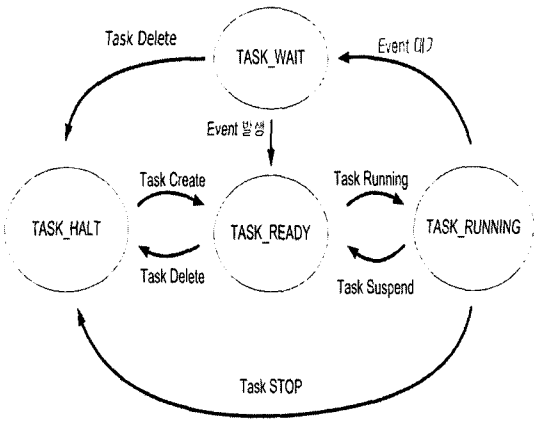


그림 1. 타스크 상태 흐름
Fig. 1. Task Status Flow.

<그림 1>는 Task의 흐름을 나타낸다. 일반적으로 Task는 Scheduler에 의해서 Context Switching 되거나 Signal을 기다리기 위해서 상태가 전이된다. 일반적으로 사용자 프로그램에 의해서 메모리에 적재되는 상태를 TASK_HALT 라고 한다. 프로그램이 실행이 되면

서 Task의 생성 시 TASK_READY 상태로 들어가게 되며, Scheduler에 의해서 실행되기를 기다린다. Scheduler에 의해서 Task가 실행이 되면 TASK_RUNNING 상태로 들어가게 되고 필요한 작업을 수행하게 된다. 정해진 시간이 경과하거나 Signal, Interrupt를 기다리는 상황이 되면 TASK_WAITING 이나 다시 TASK_READY 상태로 전이하게 된다. Task가 모든 동작을 끝마치게 되면 Task는 TASK_HALT 상태로 전이하여 작업을 마친다. TASK_HALT 상태로 전이된 후에는 Task는 더 이상 사용되지 않고 또한 어떤 Signal이나 Interrupt에 의해서도 상태가 전이되지 않는다.

2. Context Switching

<그림 2>은 Context Switching의 동작을 나타낸 그림이다. 멀티태스크 커널이 다른 Task를 실행하기로 결정되면, (1) 현재 Task의 Context (CPU register)를 현재 Task의 Stack 영역에 저장하게 된다. 이렇게 함으로서 현재 실행되고 있는 Task의 정보를 저장하게 되고, (2) 그 후에 Scheduler에 의해서 선택된 새로운 Task의 Context를 선택된 Task의 Stack 영역에서 CPU의 레지스터와 상태 레지스터 등을 복구한다. 그리고 새로운 Task를 수행하게 되면 이전에 작업이 된 이후부터 변화 없이 작업이 계속 진행이 된다.

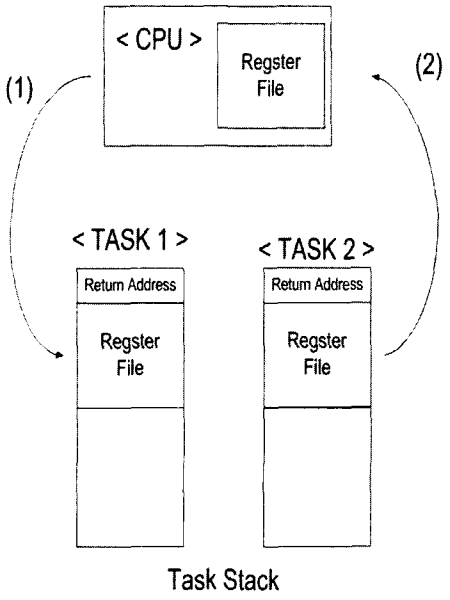


그림 2. 컨텍스트 스위칭
Fig. 2. Context Switching.

3. Scheduling

Multitasking 운영체제는 두개이상의 Task를 동시에 실행 할 수 있다. 이런 상황에서 운영체제는 먼저 실행해야 할 Task를 결정해야 한다. 이러한 결정에 관계된 운영체제의 일부분을 스케줄러(Scheduler) 라 부르고, 그것이 사용하는 알고리즘을 스케줄링 알고리즘이라고 한다.

좋은 스케줄링 알고리즘의 조건은 다음과 같다.

(1) 공정성 : 각각의 Task에게 공정하게 CPU의 시간을 할당한다.
(2) 효율성 : CPU는 그 시간의 100%를 사용하도록 한다.
(3) 응답시간 : 상호작용 하는 사용자에 대해 응답시간을 최소화 한다.
(4) 반환시간 : 일괄처리 사용자들이 출력을 기다려야 하는 시간을 최소화한다.
(5) 처리량 : 시간당 처리되는 작업의 수를 최소화한다.

이러한 스케줄링의 방식으로는 선점 스케줄링 (Preemptive scheduling)과 비 선점 스케줄링 (Non-preemptive Scheduling)이 있다.

4. Intertask Communication

Task는 다른 Task와 의 정보의 교환이나 동기화 등을 위해서 서로간의 통신이 필요하게 된다. 이러한 방법으로는 단순한 메시지를 주고받는 Signal, 블록 단위의 데이터 교환을 위한 Message Queue가 있다. Signal은 Task간의 전달되는 짧은 메시지를 의미한다. 미리 정의된 메시지를 간단하게 주고받는 형식으로 Task의 상태에 관계없이 전달이 된다. 따라서 Signal은 Task간의 정보를 전달하거나 Task를 준비 상태로 만드는 역할과 하나의 Task가 다른 Task의 신호를 받아 동기를 맞춰 동작시키는 역할도 할 수 있다.

Ⅲ. 새로운 운영체제의 설계

운영체제의 설계는 소형 시스템에 필요하지 않는 기능을 제거하고 실제 필요한 기능을 중심으로 구성을 하였다. 기본적으로 다중 작업을 위해서 멀티태스킹 스케줄러를 설계하였고, 이 스케줄러는 우선순위를 기반으로 동일한 우선순위를 가진 Task는 라운드 로빈 방식으로 스케줄링을 하게 된다. 이러한 Task들의 통신을 위해서 Signal과 Message Queue를 지원하게 된다.

Signal은 짧은 메시지를 전달하기 위해 사용되고, Message Queue는 한 블록의 데이터를 전달하기 위해 사용된다. 또한 각 Task가 디바이스 사용 시 충돌을 막기 위해 Semaphore를 이용하여 사용여부를 체크하게 되며, 디바이스를 선점한 Task는 안전하게 작업을 할 수 있게 된다. 디바이스 드라이버는 소프트웨어 계층과 하드웨어 계층을 연결하는 몇 가지 함수들로 구성이 되어, 디바이스를 추가로 장착 할 경우 커널이나 어셈블리 언어 영역의 수정 없이 드라이버를 작성할 수 있도록 하였다.

표 1. OS 셋업 파라미터
Table 1. OS Setup parameter.

```
#define   Max_Prio_Num      16
#define   Max_Task_Num      16
#define   Stack_Size 64
#define   Msg_Que_Size      64
```

<표 1>은 운영체제의 초기 설정을 조절 할 수 있는 Parameter 이다. 이 Parameter는 Task의 최대 개수, Task 우선순위의 Level 수, Stack 크기와 Message Queue 크기를 가변적으로 조절할 수 있다. 따라서 운영체제 위에서 돌아가는 응용 프로그램을 작성할 때 프로그램의 상황에 맞춰서 조절함으로써 불필요한 메모리의 낭비를 줄이고, 운영체제가 효율적으로 동작할 수 있게 할 수 있다.

1. Task Management

Task는 Operating System에서 하나의 작업 단위를 가리킨다. 각 Task는 각각의 우선 순위가 할당되어 있고, CPU register Set을 가지고 있으며, 자신의 Stack 영역을 가지고 있다. 따라서 각 Task별로 독립적인 프로그램의 수행을 할 수 있게 된다. 이러한 관리를 위해서 Operating System 에서는 Task Management를 하게 된다.

각 Task는 CPU Register Set, Stack 영역을 따로 가지고 있다. High Level Operating System 에서는 각 Task 별로 Memory 영역을 따로 가지고 있고, 이 메모리 관리를 CPU의 MMU(Memory Management Unit)에 의해서 관리를 하고 있지만, 본 논문에서는 이러한 기능을 사용하지 않고 Memory 영역을 공통 영역을 사용한다.

표 2. 타스크 제어 블록
Table 2. Task Control Block.

```
typedef struct _TCB{
 U4BYTE *Cur_Stk_Ptr;     // Stack의 현재 위치
 U4BYTE Stack[Stack_Size];
 struct _TCB              *nextTCB; // 다음 TCB
 U4BYTE *TaskPtr;         // 현재 Task pointer
 S4BYTE TCB_Usage; // TCB 의 사용여부
 S4BYTE prio;                       // Task 의 priority
 S4BYTE Task_Status;      // Task 의 상태를 저장
 S4BYTE M_Signal_Flag;    // Signal Message Flag
 U4BYTE M_Signal;         // Signal Message
 S4BYTE M_Que_Flag;       // Message Que Flag
} TCB;
```

<표 2>는 Task를 관리하기 위한 Task Control Block 의 구조를 나타낸다. 내부에 Task의 Stack을 가지고 있어서 Stack과 TCB 와 동시에 관리가 가능하여 따로 Stack 구조나 영역을 만들 필요가 없게 설계하였다. Task 간의 통신을 위해서 Signal을 저장하는 부분과 Flag를 만들어서 Signal 이 발생 시 TCB의 Flag를 ON 시킴으로서 Signal의 발생을 알릴 수 있다.

표 3. 타스크 상태
Table 3. Task Status.

```
#define   TASK_HALT         0
#define   TASK_READY        1
#define   TASK_RUNNING      2
#define   TASK_WAITING      3
```

<표 3>은 Task의 상태를 나타낸다. 각 상태는 Task의 동작을 제어하거나 Scheduling 시 Task의 선택 등에 사용이 된다. Task가 생성이 되면 TASK_READY 상태로 들어가게 되고, 여기서 Scheduler에게 제어 권을 받게 되면, TASK_RUNNING 상태로 들어가게 된다. Task가 동작을 수행하다가 Scheduler에 의해서 제어권을 넘기게 되면 Task 상태가 TASK_READY로 돌아가게 되고, 다시 다음 제어권이 넘어오기를 기다리게 된다. Task가 어떠한 Signal, Message, Interrupt 등을 기다리기 위해서 잠시 대기상태에 들어가는 경우 Task는 TASK_WAITING 상태를 나타내게 된다. 이때에는 Scheduler에 의해서 Task가 실행되지 않으며, 원하는 행동이 취해질 때 Task는 다시 TASK_

RUNNING 이나 TASK_READY 상태로 들어가서 Scheduling을 기다리게 된다. Task는 자신이 원하는 실행을 완료했을 경우와 더 이상 실행이 필요치 않은 경우에는 TASK_HALT 상태로 들어간다. Task가 이 상태로 들어갈 경우 Task를 다시 생성할 경우를 제외하고는 다시 수행할 수 없다.

2. Scheduling

우선 스케줄링 방식으로 우선순위를 바탕으로 한 Round Robin 방식을 사용하였다. 우선순위 방식은 각 Task별로 우선순위를 할당하여 중요한 Task부터 스케줄러에 의해 먼저 실행될 수 있도록 하는 방식이다. 이렇게 함으로서 실시간 처리에 필요한 Task의 우선순위를 높게 할당하여 원하는 시간 내에 Task를 처리할 수 있도록 하였다. 같은 우선순위를 가지는 Task들은 Round Robin 방식으로 Context Switching이 이루어질 수 있게 하였다. 이 방식은 동일한 우선순위의 Task는 병렬로 처리를 하게 된다. 따라서 Task들이 병렬로 처리되면서 우선순위가 높은 Task는 먼저 실행할 수 있도록 하여 실시간 성이 중요한 내장형 기기에 적합하게 구현하였다.

<그림 4>는 Scheduler의 Flowchart를 나타낸다. 우

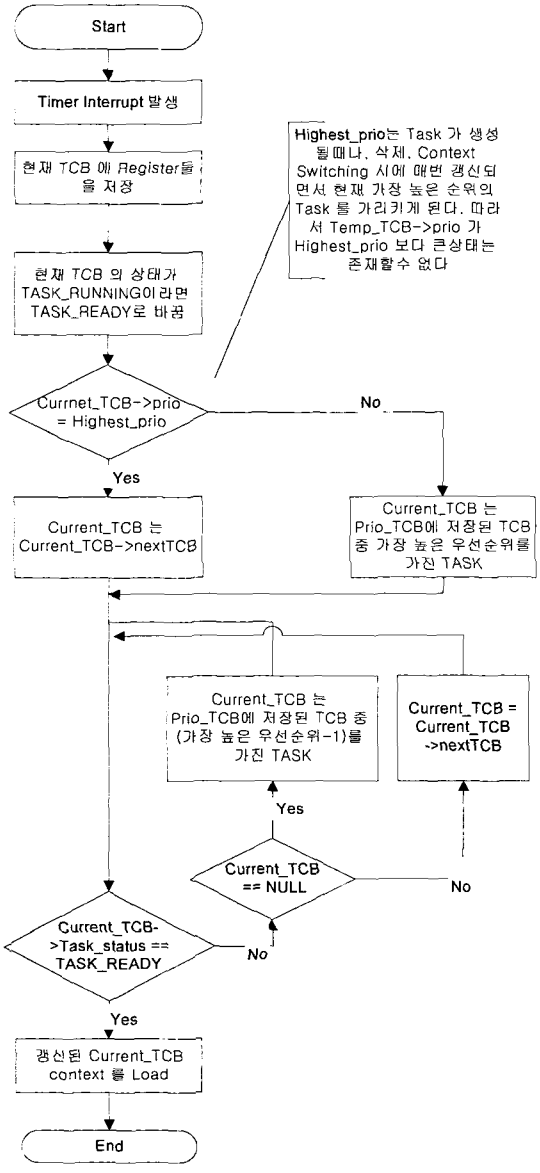


그림 4. 스케줄러의 흐름도
Fig. 4. Scheduler Flowchart.

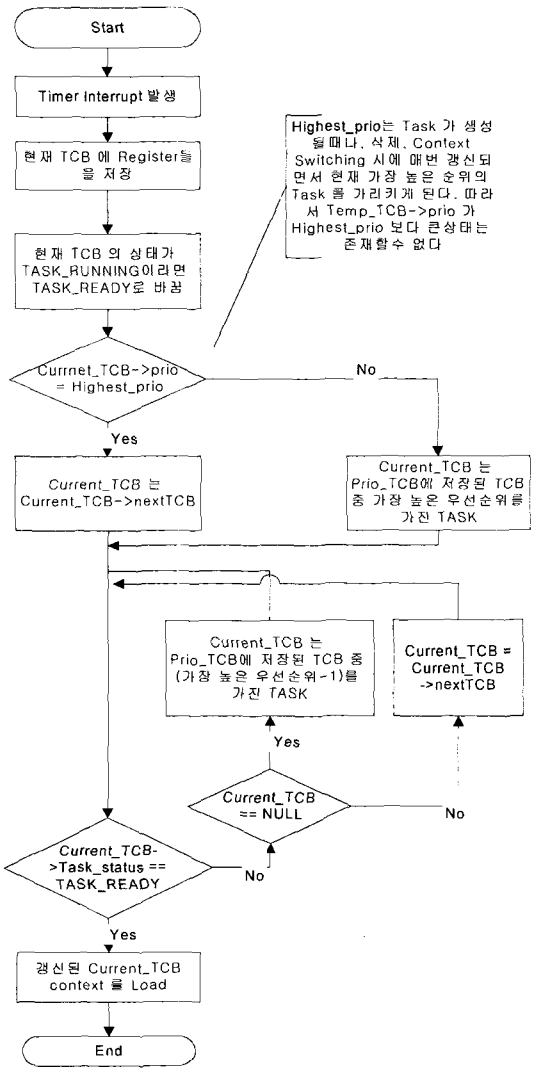


그림 5. 스케줄러의 흐름도
Fig. 5. Scheduler Flowchart.

선 Timer Interrupt에 의해서 스케줄러가 실행이 되면, 우선 Highest_prio의 값을 검사하게 된다. 현재 실행되고 있는 Task의 우선순위가 Highest_prio 값과 동일하게 되면, 동일 우선순위의 다음 Task에게 제어권을 넘기게 된다. 만일 동일 Task중에 TASK_READY상태에 들어가 있는 TASK가 없을 경우에는 다음단계의 우선순위 Task에게 제어권을 넘겨서 TASK_READY 상태의 Task를 찾게 된다. 이렇게 함으로써 우선순위가 높은 Task는 다음 우선순위에게 제어권을 양보할 수 있게 되고, Task가 외부 입력을 받기 위해 대기하는 경우 외부 입력이 없을 경우에 제어권을 받는 것을 막을 수 있다.

3. Intertask Communication

Signal의 전달은 Task의 상태와 관계없이 전달이 되며, 전달된 후 Task가 TASK_WAITING 상태라면 Task를 TASK_READY 상태로 만들어 다음의 Scheduling 시 실행되게 한다. 이러한 Signal은 데이터의 길이가 짧기 때문에 TCB 내에 Signal의 전송을 알리기 위한 Flag와 Data의 내용을 저장하기 위한 M_Signal을 설정하였다. Signal 전송 시 전송을 받는 Task의 TCB에 Flag를 1로 설정하여 Signal이 도착했음을 알리고, Signal의 내용을 입력한다. 이때 해당 Task가 TASK_WAITING 상태에 있을 경우도 있으므로 TASK_READY로 상태를 바꿔서 다음 Scheduling에 제어권을 받도록 설정한다.

Message Queue는 일정 단위의 데이터를 Task간에 전달하기 위해 사용한다. <표 4>는 Message Queue를

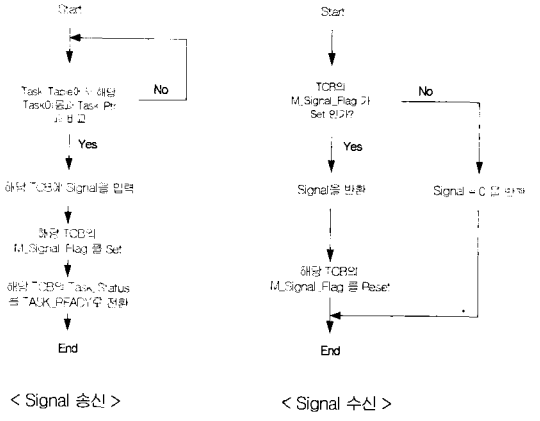


그림 6. 신호 메시지 송신/수신 함수
Fig. 6. Signal Message Transmitting/Receiving Function.

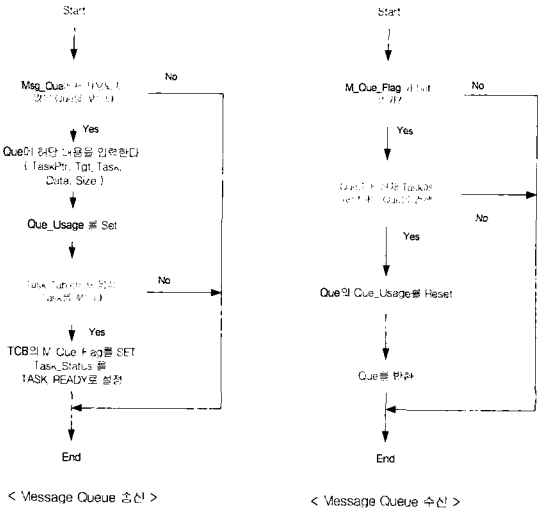


그림 7. 메시지 큐 송신/수신 함수
Fig. 6. Message Queue Transmitting/Receiving Function.

표 4. 메시지 큐
Table 4. Message Queue.

```
typedef struct        _M_Que {
        U4BYTE                  *Task_Name;
        U4BYTE                  *Tgt_Name;
        struct _M_Que *Next_Que_Ptr;
        S4BYTE                  *Que_Ptr;
        S4BYTE                  Que_Size;
        S4BYTE                  Que_Usage;
}M_Que;
```

위한 구조체 이다. 운영체제에서 사용할 수 있는 Message Queue의 길이는 사용자 프로그램에 따라서 설정이 가능하고, 이 Message Queue는 연결 리스트로 관리가 된다. 그러므로 한번 사용한 Message Queue는 필요한 정보가 전달이 되면 Que_Usage의 Flag를 0으로 만들어서 다음에 사용할 수 있게 하였다.

4. Device Driver

현재 운영체제에서 사용할 수 있는 디바이스는 DeviceList 라는 구조체를 사용하여 관리하게 된다. 이 구조체는 각 디바이스의 이름과 사용중인 Task, 그리고 Semaphore를 가지게 된다. 디바이스의 초기화가 시작이 되면 이 구조체에 초기화된 디바이스가 등록이 되고, 이 리스트를 이용해서 각 Task는 디바이스의 Access 정보를 얻게 된다. Multitask System에서는 하나의 디바이스를 여러 개의 Task가 동시에 Access 할

경우가 생기게 된다. 이러한 경우를 막기 위해서 각 Task가 Device를 Access하기 전에 디바이스에 대한 Semaphore를 얻게 된다. 이 Semaphore란 현재 디바이스가 사용되고 있는지 아닌지를 검사하는 Flag 같은 역할을 하게 된다.

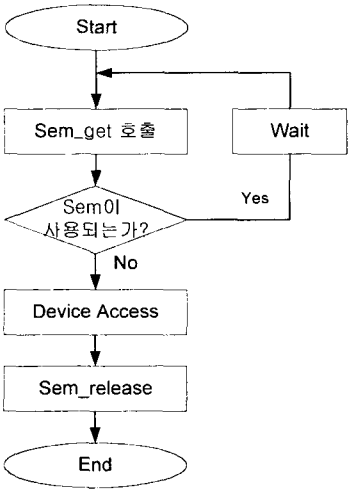


그림 7. 장치 접근 순서
Fig. 7. Device Access Order.

각 Task는 <그림 7>과 같은 절차로 Device를 사용하게 된다. Device가 Sem_get 함수를 이용하여 Device의 Semaphore를 검사하게 되고, 현재 사용중이 아니면 디바이스 리스트에 사용할 Task의 TCB를 등록하고, Semaphore를 증가시켜 디바이스가 사용중임을 나타내게 된다. 사용이 끝나게 되면 Task의 TCB를 제거하고, Semaphore를 감소시켜 디바이스를 해제하게 된다.

Ⅳ. 구 현

구현은 ARM7TDMI Microprocessor를 사용한 삼성 S3C3410 칩을 사용하였다(<표 5> 참조)[10, 11]. 운영체제의 구현은 기존의 라이브러리를 사용할 경우 불필요한 부분의 라이브러리를 추가하는 경우가 생겨 전체 크기를 증가시킬 수 있다. 따라서 기존의 라이브러리는 사용을 하지 않고 필요한 부분은 직접 제작을 하여 사용하였다. 가장 중요한 기능이 디버깅 정보를 제공해줄 수 있는 직렬 포트의 구동이다. 운영체제는 기존의 응용프로그래밍과 달리 사용할 수 있는 디버깅 환경이 제한적이기 때문에 이 직렬포트를 사용하여 필요한 정보를 개발 시스템에 제공해야 한다.

표 5. 구현에 사용된 시스템과 Target 사양
Table 5. Spec. of System and Target used for Implementation.

	구현에 사용된 시스템의 사양
운영체제	Windows 2000
컴파일러	ARM Development Suit 1.0

	Target System
프로세서	ARM7TDMI(Samsung S3C3410)
메모리	8-bit FLASH 512Kbyte X 2 SRAM memory bank with 256 K x 16 bit size
Serial Port	9-pin serial ports (CN8-RS232)

표 6. uC/OS-II의 예제 코드
Table 6. Example Code for uC/OS-II.

```
void Task1(void *i)
{
        INT8U Reply;

        for (;;)
        {
            OSSemPend(Sem2,0, &Reply);
            uHALr_printf("(1)₩n");
            OSSemPost(Sem1);
        }
}
void Task2(void *i)
{
        INT8U Reply;
        for (;;)
        {
            OSSemPend(Sem1,0, &Reply);
            uHALr_printf("(2)₩n");
            OSSemPost(Sem2);
        }
}
```

<표 5>는 구현에 사용된 시스템의 정보를 나타낸다. 개발에 사용된 Target은 가장 간단한 기능만을 가진 Evaluation Board 이다. 따라서 Ethernet Port를 제공하기 위해서 추가적인 작업을 하였다.

다음은 구현한 OS를 사용하여 간단한 프로그램을 만든 것이다. 비교를 위해서 공개된 8/16 bit OS인 uC/OS-II 와 비교를 하였다. uC/OS-II 는 제안한 OS와는 달리 우선순위 방식의 스케줄링을 하게 된다. 이 방식은 각각의 Task가 우선순위를 달리 가지게 되므로 동일한 작업을 하는 Task를 병렬로 실행하기 위해서는 Signal이나 Semaphore 등을 이용하여 다음 실행할

Task를 지정하여 실행을 시켜야 한다. 따라서 네트워크용 서버프로그램이나 단일한 작업을 여러 Task가 실행하는 병렬적인 프로그램에 적용하기에는 무리가 있게 된다. 다음의 <표 6>에서는 uC/OS-II에서 단일한 기능의 Task 2개를 설정하여 병렬로 실행되는 간단한 예이다.

두 개의 Task는 Serial Port로 간단한 문자를 전송하는 기능을 가지고 있다. 이러한 작업을 두 Task는 교대로 실행하게 된다. 위의 uC/OS-II 에서는 각각 다른 우선순위를 가지고 있기 때문에 두 개의 Task가 병렬로 실행되기 위해서는 Semaphore를 이용하여 다른 Task 에게 Signal을 보내게 되고 이러한 작업은 계속 반복적으로 일어나게 된다. 소수의 Task를 이러한 방법으로 번갈아 실행하는 것은 간단하게 이루어지나, Task의 수가 늘어나게 되면 이러한 제어는 매우 힘들어지게 된다. 반면 <표 7>은 제안된 OS를 이용하여 동일한 프로그램을 작성한 것이다. 기본적으로 동일한 우선순위를 가지고 실행이 되고, 동일한 우선순위는 일정시간이 지난 후에는 다른 Task로 교대를 하게 된다. 따라서 위의 <표 6>에서와 같이 Semaphore를 이용하여 다음의 Task에게 Signal을 전달할 필요가 없게 되고, 또 Task의 수가 늘어나도 간단히 제어가 된다는 장점이 있다.

표 7. 제안된 OS의 예제 코드
Table 7. Proposed OS example Code.

```
int Test1_Task(void *data)
{
        for(;;){
            Uart_Printf("(1)₩n");
        }
        return 1;
}

int Test2_Task(void *data)
{
        for(;;){
            Uart_Printf("(2)₩n");
        }
        return 1;
}
```

<표 8>은 uC/OS-II의 Main 함수 부분이다. Main 함수는 프로그램에서 가장 처음 실행하는 부분이므로 각 OS의 초기화와 각 변수, 상수들을 정의하게 된다. uC/OS-II 에서는 Task를 교대로 실행하기 위해 Semaphore를 사용하기 때문에 OSSemCreate() 함수를 사용하여 각 Task의 Semaphore를 생성하게 된다.

표 8. uC/OS-II 의 Main 함수
Table 8. Main Function of uC/OS-II.

```
int Main(int argc, char **argv)
{
 char Id1 = '1';
 char Id2 = '2';

 ARMTargetInit();

 OSInit();

 OSTimeSet(0);

 Sem1 = OSSemCreate(1);
 Sem2 = OSSemCreate(1);

 OSTaskCreate(Task1,    (void    *)&Id1,    (OS_STK
*)&Stack1[STACKSIZE - 1], 1);
 OSTaskCreate(Task2,    (void    *)&Id2,    (OS_STK
*)&Stack2[STACKSIZE - 1], 2);

 ARMTargetStart();

 OSStart();

 return 0;
}
```

반면 제안된 OS의 Main 함수를 나타내는 <표 9>에서는 OS의 최기화 함수들과 각 Task를 생성하는 함수만으로 구성이 되었다.

표 9. 제안된 OS의 Main 함수
Table 9. Main Function of Proposed OS.

```
int Main(void)
{
        Led_Display(0xf);
        Isr_Init();
        Uart_Init(115200);

        Init_OS();
        if (!Create_Task(8,Test1_Task)){
            return 0;
        }
        if (!Create_Task(8,Test2_Task)){
            return 0;
        }

        OS_start();

}
```

위의 차이는 응용프로그램의 성격에 따라서 장·단점이 될 수 있다. 하지만 본 논문에서 제안한 OS의 목적은 소규모 내장형 네트워크 시스템의 구성을 목적으

로 한다. 네트워크에서 서비스를 제공하기 위한 시스템에서는 다수의 동일한 권한을 가진 유저의 접속을 처리하기 때문에 다수의 동일한 우선순위의 Task를 처리해야 한다. 물론 중요한 기능을 하는 Task는 우선순위를 높게 설정하여 필요시에 먼저 처리되도록 설정할 수 있다.

V. 결과 및 검토

<표 10>은 구현된 운영체제의 컴파일 결과를 나타낸다. 총 운영체제의 사이즈는 12,412Byte의 ROM 공간과 9728Byte의 RAM 공간을 사용한다. 물론 운영체제에서 사용되는 파라미터들을 수정하면 코드의 사이즈는 변경될 수 있다.

표 10. 구현된 소스의 컴파일 결과
Table 10. Compile Result for Implemented Source.

```
==============================================================
Image component sizes
 Code RO Data RW Data ZI Data Debug      Object Name
 1328 0 0 9548 1904          Task.o
 260 0 0 0 0   sched.o
 684 0 0 96 1660             device.o
 1288 0 4 4 4112             k401lib.o
 996 0 0 0 0   k401init.o
 176 0 0 0 872               timer.o
 828 0 0 12 3632             401mon.o
--------------------------------------------------------------
 5560 0 4 9660 12180 Object Totals
==============================================================
 Code RO Data RW Data ZI Data Debug
 11880 528 4 9724 14696 Grand Totals
==============================================================
 Total RO Size(Code + RO Data) 12408      ( 12.12kB)
 Total RW Size(RW Data + ZI Data) 9728    ( 9.50kB)
 Total ROM Size(Code + RO Data + RW Data) 12412      ( 12.12kB)
==============================================================
```

구현된 운영체제는 특정 하드웨어나 프로세서에 의존적이지 않게 어셈블러로 기술된 소스를 최소로 하여 다른 하드웨어에 적용이 용이하게 구성하였고, 각 Task의 수와 Stack의 크기, Task 간 통신기능 등 대부분의 기능을 선택 가능하게 하여, 코드의 길이를 최적화 할 수 있게 구성되어 있다. 따라서 소형 내장형 기기에서 필요한 기능을 지원하면서 메모리는 최소한으로 사용함으로서 운영체제의 활용범위가 넓어지고 응용프로그램의 개발이 용이하게 될 것이다. 또한 시스템의 하드웨어가 바뀌거나 프로세서가 교환될 때 최소한의 수정으로 기존의 응용프로그램을 사용할 수 있으므로 개발비용이나 시간을 절약할 수 있다.

향후 네트워크 인터페이스 관련 디바이스 드라이버를 구현하여 본 운영체제를 적용하면 본 운영체제의 활용 범위는 훨씬 넓어 질 것이다. 본 운영체제는 아직은 안정성이 보장이 안되고, 응용프로그램을 작성하기에는 디바이스 드라이버 등이 부족하다. 하지만 점차 부족한 점을 개선하고, 성능을 추가한다면 한결 쉽고, 효율적으로 네트워크 상에서 동작하는 소형기기의 구현을 할 수 있을 것이다.

참 고 문 헌

[1] Jean J, Labrosse "Micro C/OS-II", R&D Book

[2] Andrew S. Tanenbaum "Modern Operating System", Prenties-Hall, 1996.

[3] Daniel P. Bovet "Understanding the Linux Kernel", O'REILLY, 2001.

[4] Alessandro Rubini "Linux Device Drivers", O'REILLY, 2000.

[5] 리눅스를 이용한 독립장비 개발, http://kesl.org

[6] Kernel Korea, http://kernelkorea.org

[7] 리눅스 한글문서 프로젝트, http://kldp.org

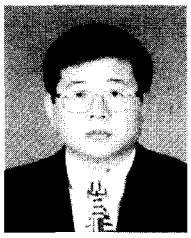
[8] Yanbin Li, Miodrag Potkonjak and Wayne Wolf "Real-Time Operating Systems for Embedded Computering", Department of Electrical Engineering, Princeton University Department of Computer Science, UCLA, 1997.

[9] David Stepner, Nagarajan Rajan, David Hui, "Embedded Application Design Using a Real-Time OS", Integrated System, Inc[8]. ARM, "ARM Software Development Toolkit"

[10] ARM Ltd, http://www.arm.com

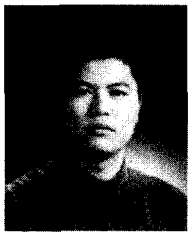
[11] Micrium, Inc http://www.ucos-ii.com

저 자 소 개



南 相 曄(正會員)

1982년 2월 : 단국대학교 공과대학 전자공학과(학사). 1984년 2월 : 단국대학교 대학원 전자공학과(석사). 2002년 2월 : 단국대학교 대학원 전자공학과(박사). 1984년 1월~1992년 1월 : 삼성종합기술원 정보시스템연구소 주임연구원. 1992년 1월~1998년 3월 : 모토로라반도체통신(주) 기술연구소 차장. 1998년 3월~2003년~현재 : 경문대 정보통신과 교수. <주관심분야 : 멀티미디어 신호처리, 컴퓨터통신, 마이크로프로세서, DVD, CD-R/W, D-TV, STB, 멀티미디어콘텐츠, 전자상거래, 3D ,DSP, 영상/음성인식, 인터넷방송. 디지털TV>



李 相 杭(正會員)

1993년 2월 : 단국대학교 전자공학과(학사). 1999년 2월 : 단국대학교 대학원 전자공학과(석사). 1999년 7월~현재 : 단국대학교 대학원 전자공학과 박사과정. 2001년 7월~현재 : 전자부품연구원 전임연구원. <주관심분야 : RTOS, MPEG-2/4, 음성/영상 처리/인식, 디지털 신호처리, 디지털 TV, 임베디드 시스템>

朴 仁 政(正會員)

1974년 2월 : 고려대학교 전자공학과(학사). 1980년 9월 : 고려대학교 대학원 전자공학과(석사). 1986년 2월 : 고려대학교 대학원 전자공학과(박사). 1981년 3월~2003년 현재 : 단국대학교 교수. 1988년 10월~1989년 9월 : Bowling Green주립대 객원교수. 1995년 1월~2000년 : 대한전자공학회 교육이사. 1999년 1월~2000년 : 대한전자공학회 멀티미디어 연구회 전문위원장. 1999년 3월~2003년 현재 : 한국XDSL포럼 의장. 2000년 7월~2003년 현재 : 한국인터넷방송/TV학회 회장. <주관심분야 : 멀티미디어 신호처리, 컴퓨터통신, 이동통신, 인터넷방송 실시간처리시스템, 인터넥컨텐츠, 정보보호기술 등임>