효율적 인터럽트 처리를 위한 인터럽트 서비스 프로세스의 구현*

양희권[0], 조희남, 성영락[†], 이철훈
충남대학교 컴퓨터공학과, [†]국민대학교 전자정보통신공학부
{hkyang, hncho, chlee}@ce.cnu.ac.kr, [†] yeong@mail.kookmin.ac.kr

Implementation of Interrupt Service Process for Efficient Interrupt Handling

Hui-Kwon Yang[0], Hee-Nam Jo, Yeong Rak Seong[†], and Cheol-Hoon Lee
Dept. of Computer Engineering, Chungnam National Univ.
[†] School of Electrical Engineering, Kookmin Univ.

요 약

인터럽트는 시스템의 내·외부의 프로그램 또는 장치에 의해 발생하는 신호로서 운영체제가 하던 일을 멈추고 인터럽트 서비스 루틴(Interrupt Service Routine)을 통해 적합한 동작을 수행하도록 한다. 실시간 시스템을 포함한 대부분의 컴퓨팅 시스템에서 인터럽트의 발생 빈도와 인터럽트 서비스 루틴의 수행시간에 따라 Response Time 이 길어질 수 있는데, 이는 시스템에 커다란 오버헤드가 된다. 본 논문에서는 실시간 운영체제에서 Response Time 을 줄이고 효율적으로 인터럽트를 서비스하기 위한 인터럽트 서비스 프로세스의 구현에 대해 기술한다.

1. 서 론

인터럽트가 발생하면 운영체제는 하던 일을 멈추고 ISR(Interrupt Service Routine)을 통해 각 인터럽트에 맞는 모듈을 실행하게 된다. 따라서 인터럽트의 발생빈도와 각 ISR 의 수행시간에 따라 단위시간당 Response Time 이 길어질 수 있다. 인터럽트는 시스템에 있어서 꼭 필요한 요소이지만 인터럽트가 잦거나 Response Time 이 길어지면 그 만큼 다른 프로세스들의 작업에 방해가 되는 셈이다. 특히 실시간 운영체제에서는 시간 결정성(Determimism)을 저해하는 요인이 된다.

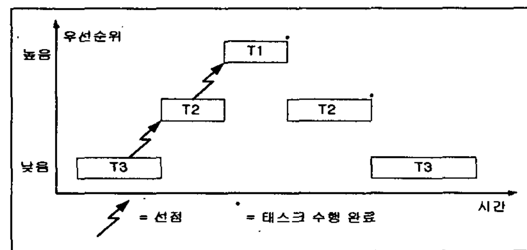
실시간 운영체제는 멀티 태스킹을 지원하며 우선순위 기반의 선점 또는 비 선점형 스케줄러를 제공한다. 본 연구팀에서는 실시간 운영체제의 이와 같은 특징을 이용하여 선점형 스케줄링 방식을 지원하는 실시간 운영체제에서 Response Time 을 줄이고 효율적으로 인터럽트를 처리할 수 있는 방안에 대해 연구하였다. 즉, ISR 에서는 어떤 인터럽트가 발생하였는지 감지하고 있다가, 별도의 프로세스가 일괄적으로 각 인터럽트와 관련된 모듈을 수행할 수 있도록 인터럽트 서비스 프로세스(Interrupt Service Process)를 설계하고 구현하였다.

본 논문의 구성은 2 장에서 기반연구로서 실시간 운영체제의 선점형 스케줄링과 인터럽트에 대하여 설명하고 3 장에서는 인터럽트 서비스 프로세스의 구현에 관해 설명한다. 4 장에서는 테스트 환경 및 결과를 보이고 5 장에서는 결론 및 향후 과제를 기술한다.

2. 기반연구

2.1 선점형 스케줄링(Preemptive Scheduling)

실시간 운영체제는 선점형(Preemptive) 과 비선점형(Non-Preemptive) 또는 복합된 형태의 스케줄링을 제공하는데, 본 연구는 선점형 스케줄링을 제공하는 실시간 운영체제를 기반으로 진행되었다. 선점형 스케줄링이란 프로세스의 우선순위에 기반하여 가장 높은 우선순위를 가지는 실행준비상태의(Ready State) 프로세스에게 CPU 사용권을 부여하는 것으로서 응답성(Responsiveness)이 중요시되는 시스템에 적합하다. 다음의 [그림 1]에서와 같이 현재 CPU 를 점유하고 있던 프로세스(또는 태스크)보다 우선순위가 높은 프로세스가 실행준비 상태가 되면 CPU 사용권은 해당 프로세스에게 넘어간다.



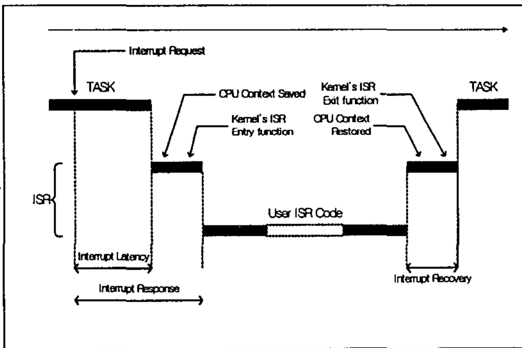
[그림 1] 선점형 스케줄링

*본 논문은 산업자원부 중기거점과제 연구비 지원에 의한 것임

2.2 인터럽트의 처리

일반적으로 CPU 는 FIQ(Fast Interrupt Request)와 IRQ(Interrupt Request) 두가지의 인터럽트 모드를 지원한다. IRQ 는 일반적인 인터럽트로 중첩(Nesting)가능하며, FIQ 는 즉시 처리되어야 하는 인터럽트로서 중첩되지 않도록 가장 중요한 한, 두개의 인터럽트만 등록하여 사용한다

인터럽트는 CPU 의 특정 레지스터의 플래그를 조정하거나 상태레지스터를 통해 몇몇 또는 전체 인터럽트를 Disable 또는 Enable 할 수 있다. Enable 즉, 허용가능한 인터럽트가 발생하면 CPU 는 인터럽트 모드로 들어가게 된다. 이때 운영체제는 현재 수행중이던 프로세스의 Context 를 저장(save)하고 발생한 인터럽트에 해당하는 모듈(user ISR code)을 실행한다. 모듈의 실행이 종료되면, 선점형 실시간 운영체제의 경우 재스케줄링을 통해 우선순위가 가장 높은 프로세스를 선택하여 Context 를 적재(restore)함으로써 인터럽트의 처리를 마치게 된다. 이와 같이 인터럽트와 관련한 일련의 처리를 담당하는 것을 ISR(Interrupt Service Routine)이라고 한다. 아래 [그림 2]는 선점형 실시간 운영체제에서의 Interrupt Latency, Response, Recovery Time 을 예로 든 것이다.



[그림 2] Interrupt Latency, Response, Recovery

3. 인터럽트 서비스 프로세스의 구현

기본적으로 FIQ 모드를 지원하지 않는 대신 8 단계의 인터럽트 등급을 두고 각 단계마다 발생한 인터럽트의 User ISR Code 를 리스트 구조로 관리할 수 있도록 하였으며, 발생시간에 관계 없이 등급이 높은 단계의 리스트에 있는 User ISR Code 를 우선으로 실행하도록 설계하였다. 또한 모든 프로세스에 우선하여 실행할 수 있도록 최상위 우선순위(0)를 배정하고 다른 프로세스가 이 우선순위를 함부로 사용할 수 없도록 처리하였다. 또한 수용 가능한 범위를 지정하여 무한정 리스트에 넣을 수 없도록 하였다. 최악의 경우 리스트가 포화상태가 되면 이후 발생하는 인터럽트는 무시된다.

3.1 환경변수

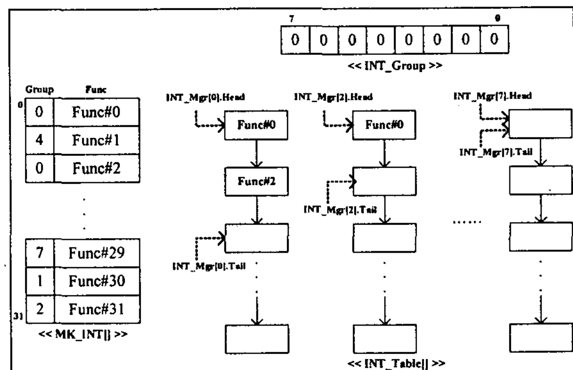
[그림 3]은 인터럽트 서비스 프로세스를 위한 추가된 구조체 및 환경변수이다. INT_Group 은 8bit 변수로 bit 별로 8 단계의 인터럽트 등급을 나타내며 1 로 set 된 bit 의 등급에 처리할 인터럽트가 발생되어 있음을 의미한다. 각 등급의 인터럽트는 16(MAX_INT_Group_Entry)개의 INT_Entry 로 구성된 리스트와 8 개의 배열로 선언된 INT_Manager 에 의해 관리할 수 있도록 하였다. MK_INT[32]는 각 인터럽트에 대한 등급과 User ISR Code 를 등록하기 위한 변수이다.

```
#define MAX_INT_Group_Entry     16
unsigned char INT_Group;

struct INT_Entry {
        MK_voidFuncVoid   Func;
        struct INT_Entry   *Next;
}
typedef struct int_list_manager {
        struct INT_Entry   *Head;
        struct INT_Entry   *Tail;
} INT_List_Manager;
typedef struct int_table {
        unsigned char Group;
        MK_voidFuncVoid   Func;
} INT_Table;

struct INT_Entry        *INT_Table[8];
INT_List_Manager        INT_Mgr[8];
INT_Registry            MK_INT[32];
```

[그림 3] 추가된 환경변수

3.2 인터럽트 서비스 프로세스

인터럽트 서비스 프로세스를 구현하기 위해 기존의 인터럽트 초기화, 등록 함수 및 ISR 의 내용을 추가, 수정하고 인터럽트 서비스 프로세스를 위한 함수를 구현 하였으며 함수의 구성 및 세부 내용은 다음의 [그림 5]와 같다. 인터럽트 서비스 프로세스는 Interrupt_Service 함수로 구현하였는데, 인터럽트가 발생한 Group 중 높은 등급을 판별하고 INT_Group 를 초기화하는 연산의 효율을 높이기 위해 μ COS 를 비롯하여 본 연구팀에서 자체 개발한 iRTOS™에서 스케줄링에 사용하는 UnMapTable 과 MapTable 을 이용하였다. 아래 [그림 4]는 본 논문에서 구현한 인터럽트 서비스 프로세스를 이용한 인터럽트 서비스의 실행 예를 도식화 한 것이다.

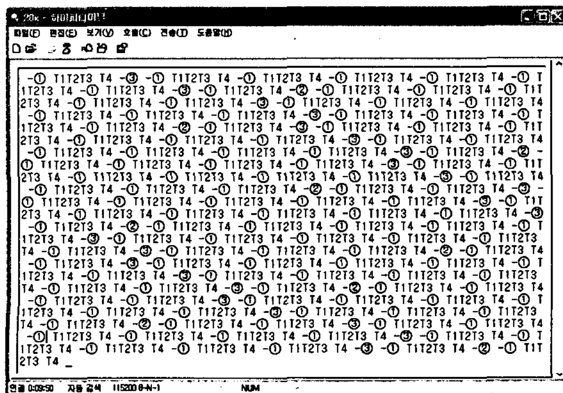


[그림 4] 인터럽트 서비스 프로세스의 실행 예

```
void MK_InterruptInitialize(void) {
    //중략
    INT_Group = 0x00L;
    /* INT_Table[] 초기화 */
    ...
}
int MK_IRQInstall(int Num, int Group, void (*pFunc)(void)) {
    //중략
    MK_INT[Num].Group = Group;
    MK_INT[Num].Func = pFunc;
    //중략
}
void MK_ServiceIRQ(void) {
    //중략
    for(Num=0; Num<31; Num++) {
      if( ((SRCPND&(1<<Num))&INTMSK) ) {
        Group = MK_INT[Num].Group;
            /* 리스트가 Full 이면 Skip*/
            ...
        INT_Mgr[Group].Tail->Func = MK_INT[Num].Func;
        INT_Mgr[Group].Tail = INT_Mgr[Group].Tail->Next;
        INT_Group |= MK_MapTable[Group];
        SRCPND = 1<<Num;
      }
    }
    if (INT_Group)
      MK_Resume(&TaskISP);
}
PROCESS Interrupt_Service {
    //중략
      while(INT_Group) {
        Group = MK_UnMapTable[INT_Group];
        (*INT_Mgr[Group].Head->Func)();
        INT_Mgr[Group].Head->Func = NULL;
        INT_Mgr[Group].Head=INT_Mgr[Group].Head-
>Next;
        /* 리스트에 있는 나머지 User ISR Code 실행 */
        ...
        INT_Group &= ~MK_MapTable[Group];
      }
      MK_Suspend(&TaskISP);
    }
}
```

[그림 5] 인터럽트 서비스 프로세스 관련 모듈

4. 테스트 환경 및 결과



[그림 6] 테스트 결과

본 연구는 ARM-920T CPU 가 탑재된 S3C2800 보드 기반으로 SDT2.51 ARM 용 통합개발환경을 사용하여 진행하였다. 커널은 본 연구팀에서 자체 개발한 iRTOS ™을 이용하여 인터럽트 서비스 관련 부분만을 재구성하였다. [그림 6]은 인터럽트 서비스 프로세스를 적용한 iRTOS™에서 테스트한 결과이다. 3 개의 타이머 인터럽트 ①, ②, ③을 각각 10, 255, 30 CLK 의 주기로 활성화하고 <2, 1, 1>, <1, 3, 2>등으로 인터럽트 처리 등급에 변화를 주어 테스트하였다. 화면에서 ①, ②, ③은 타이머 인터럽트를 나타내며, T1, T2, T3, T4 는 시스템에서 활동중인 프로세스이다.

5. 결론 및 향후 과제

본 논문에서는 실시간 운영체제에서 인터럽트를 처리하기 위한 별도의 프로세스를 설계 구현함으로써 각각의 인터럽트를 처리하는 데 필요한 Context Switching 시간을 줄이고, User ISR Code 의 수행이 길어질 가능성에 대해서도 유연하게 되었다. 또한 동등한 수준의 인터럽트 사이에서는 발생한 순서에 따라 순차적으로 수행되기 때문에 인터럽트 중첩(Nesting)으로 인한 오버헤드가 줄어드는 효과를 얻을 수 있었다. 향후 좀더 심도있고 정밀한 테스트와 면밀한 분석을 통한 안정화의 노력이 계속되어야 할 것이다.

6. 참고문헌

[1] http://www.inestech.com

[2] C.M.Krishna, Kang G.Shin, “Real-Time Systems”, McGraw-Hill, 1997

[3] Jean, J. Labrosse, “μC/OS The Real-Time Kernel”, R&D Publications, 1995.

[4] David Stepner 외 2 명, “Embedded Application Design Using a Real-Time OS”, IEEE, 1999.

[5] University of York, “Real-Time Systems and Programming Languages”, Addison Wesley, 1997

[6] http://www.acceleratedtechnology.com/embedded/nucleus.html