Linux 에서의 Interrupt 처리 메커니즘

2018112749, 전현승

I. 들어가며

우리가 일상 생활에서 사용하는 대부분의 컴퓨터는 한 개의 프로세서를 사용하고 있다. 하나의 프로세서로 여러 concurrent 한 작업을 수행하기 위해 Operating System 에서는 Multiprogramming 이나 Time-sharing 과 같은 방법을 사용하지만, 근본적으로 하나의 프로세서는 순간에는 하나의 작업밖에 처리할 수 없다. 따라서, 어떤 작업을 처리하는 도중에 우선순위가 급한 작업을 처리할 필요가 있거나, 처리 도중 문제가 생겨 현재 작업을 멈추고 다른 작업을 시작해야 할 때, 처리 도중 입력 장치의 입력이 들어왔을 때 등, 이러한 상황에 대처할 수 있는 수단이 필요하다. 이러한 때에 사용할 수 있는 기능이 바로 인터럽트(Interrupt)이다.

컴퓨터 구조 전반에서, 특히 운영체제에서 인터럽트는 상당히 중요한 부분을 차지한다. 인터럽트의 존재 자체만으로 컴퓨터 전반적으로 유저에게 더 나은 control 을 제공하며[1], 심지어 운영체제 자체가 인터럽트 중심으로 구동된다고 생각하는 시각도 있다. 실행할 프로세스도 없고, 서비스할 입출력 장치도 없으며, 응답할 사용자도 없다면 운영체제는 어떤 일이 일어날 때까지 무한히 대기한다[2]. 현대의 운영체제들이 "Interrupt-driven"하다는 주장도 어느 정도 일리가 있다.

이 보고서에서는 다음과 같은 순서와 구성으로 인터럽트에 대한 전반적인 부분을 '리눅스 운영체제에서의 인터럽트 처리 메커니즘'을 중심으로 살펴본다.

- "II. 인터럽트의 정의, 종류와 필요성" 에서는, 우선 인터럽트라 불리는 개념이 무엇인지 알아보고, 인터럽트의 여러 종류 중 일부와 인터럽트의 존재의의와 필요성에 대해 알아본다.
- "III. 리눅스 운영체제에서의 인터럽트 처리 메커니즘" 에서는, 인터럽트의 발생부터 인터럽트가 처리되는 과정, 인터럽트 처리 후 기존의 작업으로 다시 돌아가는 과정까지, 인터럽트의 생명 주기를 리눅스 운영체제 측면에서 자세히 알아본다.
- "IV. 결론 및 의문" 에서는, 조사 및 학습, 분석 과정에서 새롭게 배운 점들과 이 과정에서 생긴 의문점, 그리고 제 나름의 결론을 소개하면서 보고서를 마무리한다.

II. 인터럽트의 정의, 종류와 필요성

우선 리눅스 운영체제에서 인터럽트를 어떻게 처리하는지에 대해 알아보기 전에, 인터럽트가 무엇인지, 인터럽트에는 어떠한 종류가 있는지, 그리고 인터럽트라는 개념이 왜 필요한지에 대해 각각 간략히 알아볼 필요가 있다.

A. 인터럽트의 정의

인터럽트 자체는 단순히 추상적인 개념, 또는 메커니즘을 부르는 용어이고, 이의 정의는 책마다, 그리고 인터럽트에 대해 말하는 사람마다 다르게 나타난다. 어떤 이는 인터럽트를 "프로세서의 주의를 끌기 위해 하드웨어가 보내는 신호"[3]라고 정의하기도 하고, 또 다른 이는 "컴퓨터가 그 주변기기로부터 받는 요구의 일종"이라고 정의하기도 한다.

다만 이러한 모든 주장을 통틀어봤을 때, 이들이 공통으로 포함하는 핵심 내용을 정리해보면 인터럽트는 "프로세서의 구동 흐름을 방해하는 행위"라 할 수 있겠다. 영어단어 'interrupt'가 '방해하다', '가로막다', '끼어들다'[4] 정도의 사전적 의미가 있다는 것을 고려했을 때 이는 어느 정도 합리적인 생각이다.

B. 인터럽트의 종류와 원인

인터럽트는 대표적으로 인터럽트 신호가 하드웨어 이벤트에 의해 생성되는지, 소프트웨어 이벤트에 의해 생성되는지에 따라 크게 두 가지로 나눌 수 있다.

- 하드웨어 인터럽트 (외부 인터럽트)는 입출력 장치, 등 프로세서 외부에 있는 장치들이 운영체제로부터 주의를 끌기 위해 사용된다. 쉬운 예로, 키보드 키를 누르거나 마우스 포인터를 움직이는 것과 같은 행위들은 모두 하드웨어 인터럽트를 일으켜, 프로세서가 키 입력이나 마우스 위치를 읽게끔 한다. 하드웨어 인터럽트 요청은 주로 'IRQ(Interrupt request)'라는 인터럽트 신호로 나타난다.
- 소프트웨어 인터럽트 (내부 인터럽트)는 프로세서가 인스트럭션을 수행하는 도중에, 또는 특정한 조건을 만족했을 때 프로세서 자체가 발생시키는 인터럽트를 말한다. 실행 중인 프로그램상의 예상치 못한 오류들 또한 소프트웨어 인터럽트라 할 수 있는데, 특히 이러한 경우의 인터럽트는 'trap' 또는 'exception'이라 부르기도 한다. 일반적으로 'interrupt'라는 용어 자체가 'interrupts'와 'exceptions' 두 용어를 가리키는 데에 사용된다는 주장도 있다[5].

이외에도 인터럽트에는 다양한 종류가 있고 분류를 나누는 기준도 다양하지만, 이와 관련한 더 자세한 내용은 보고서의 주제에서 벗어나므로 생략한다.

C. 인터럽트의 존재의의와 필요성

인터럽트의 존재의의, 즉 인터럽트의 주요 목적 중 하나는 다른 작업을 처리하면서도 프로세서 외부 장치로부터의 정보를 누락 없이 수신하는 것이라 할 수 있다. 그러나 앞서 말했듯 인터럽트는 입출력 장치 처리 외에도 프로그램상의 예외 처리 등 다양한 방면에서 사용될 수 있다. 인터럽트를 사용함으로써 얻을 수 있는 구체적인 효과들은 다음과 같은 것들이 있다:

- 프로세서 자원을 더 효율적으로 이용할 수 있다. 외부 장치의 처리 속도는 프로세서의 처리 속도보다 훨씬 느리기 때문에, 외부 장치가 작업을 처리하는 동안 프로세서는 다른 작업을 수행하는 것이 바람직하다. 따라서, 프로세서가 다른 작업을 처리하고 있을 때, 외부 장치에서 이벤트가 발생했거나, 작업이 필요하거나, 작업이 종료되는 등 외부 장치에서 프로세서의 처리가 필요할 경우, 인터럽트라는 신호를 발생시켜 프로세서에게 알리는 것이다. 참고로 이전에는 일정 주기마다 외부 장치의 상태를 정기적으로 일일이 확인하는 '폴링(Polling)'이라는 방식을 사용하기도 했지만, 이는 방금 설명한 인터럽트와 비교하면 상대적으로 비효율적인 방법이라 할 수 있다.
- 앞서 말했듯, 인터럽트는 프로그램의 예외 처리에 사용될 수 있다. 프로그램 수행 중에 소프트웨어적인 에러가 발생했거나 외부 장치의 고장이 발생한 경우, 또는 정전과 같은 비정상적인 상황이 발생한 경우, 프로세서에서는 현재 처리하고 있는 작업을 잠시 중단하고 다른 작업을 처리해야 할 필요가 있다. 일반적으로는 구현하기 어렵지만, 인터럽트의 개념을 이용하면 프로그램이나 외부 장치에서 프로세서에 인터럽트를 보내고, 프로세서가 이를 확인하고 현재 수행 중인 작업을 교체하는 식으로 프로세서에서 예외에 신속하게 대응할 수 있다는 장점이 있다.

이외에도 인터럽트를 도입함으로써 얻을 수 있는 이점은 많지만, 마찬가지로 이와 관련한 더 자세한 내용은 보고서의 주제에서 벗어나므로 생략한다.

III. 리눅스 운영체제에서의 인터럽트 처리 메커니즘

이전 챕터에서는 인터럽트에 대한 전반적인 사실들을 알아보았다. 그래서, 프로세서가 다른 프로그램을 수행하는 도중 인터럽트가 발생하면, 리눅스 운영체제 내부에서는 실제로 어떤 일이 일어나는가? 인터럽트가 발생하고, 프로세서가 이를 처리하고 다시 원래 수행하던 프로그램으로 돌아오기까지의 자세한 과정을 리눅스 운영체제 측면에서 알아보도록 하자.

들어가기에 앞서, 리눅스 커널은 일반적인 인터럽트 처리 메커니즘과 인터페이스를 가지고 있고, 이는 리눅스 커널을 탑재한 운영체제라면 거의 비슷하지만, 인터럽트 처리에 관련된 세부 사항들은 프로세서 아키텍처에 따라 달라질 수 있음을 양지하라.

A. 대략적인 인터럽트 처리 과정

우선, 리눅스에서의 인터럽트 처리가 전체적으로 어떠한 흐름으로 일어나는지 알아볼 필요가 있다. 전반적인 인터럽트의 처리 과정은 Fig 1.과 같고, 다음과 같이 크게 세 단계로 나누어볼 수 있다. (각 과정에 대한 자세한 내용은 후술한다.)

a) 프로세서가 기존의 프로그램을 처리하는 도중 인터럽트가 발생하면, 우선 프로세서는 현재 실행 중인 프로그램의 실행을 중단하고, 현재 프로그램의 상태를 보존한다.

b) 해당하는 인터럽트를 처리하는 루틴을 찾아, 이를 실행한다.

c) 해당 인터럽트 처리가 끝나면, 아까 저장했던 프로그램의 상태를 복구하고, 이전에 실행하던 프로그램 위치로 돌아가 실행을 재개한다..

인터럽트 처리에 필요한 각 과정을 상세하게 알아보도록 하자.

B. 현재 프로그램 중단 및 상태 저장

앞에서 설명한 바와 같이, 프로세서는 인터럽트 신호를 받으면 우선 현재 실행 중인 프로그램 상태를 저장하고, 인터럽트 처리가 끝나면 기존에 실행하던 프로그램을 원래 상태로 복구해야 한다. 그리고 이는 인터럽트를 처리하는 데 있어 핵심적인 부분이라 할 수 있다.

프로세서는 현재까지 실행 중이었던 상태를 해당 프로세스의 PCB(Process Control Block)에 저장한다. PCB 는 프로세스의 정보를 담는 리눅스 커널의 자료구조의 일종으로, 프로세스 상태, 현재 수행 중인 인스트럭션의 메모리 주소, 해당 프로세스의 주소 공간, 레지스터 값 등 프로세스를 표현하는 데에 필요한 거의 모든 정보를 담고 있다. PCB 는 "운영체제가 프로세스를 표현하는 방법"[6]이라 할 수 있다. 실행 중인 프로세스가 교체되면 레지스터 값은 당연히 덮어씌워질 것이기 때문에, 프로세서는 이러한 레지스터 값들이 바뀌기 전에 해당 프로세스의 PCB 에 프로세스의 정보를 저장하고 인터럽트를 처리하기 위한 준비를 한다.

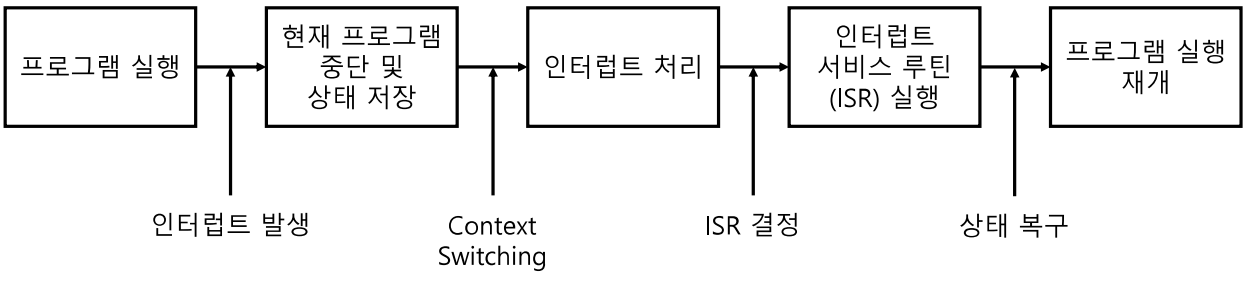

Fig. 1. 대략적인 인터럽트 처리 과정

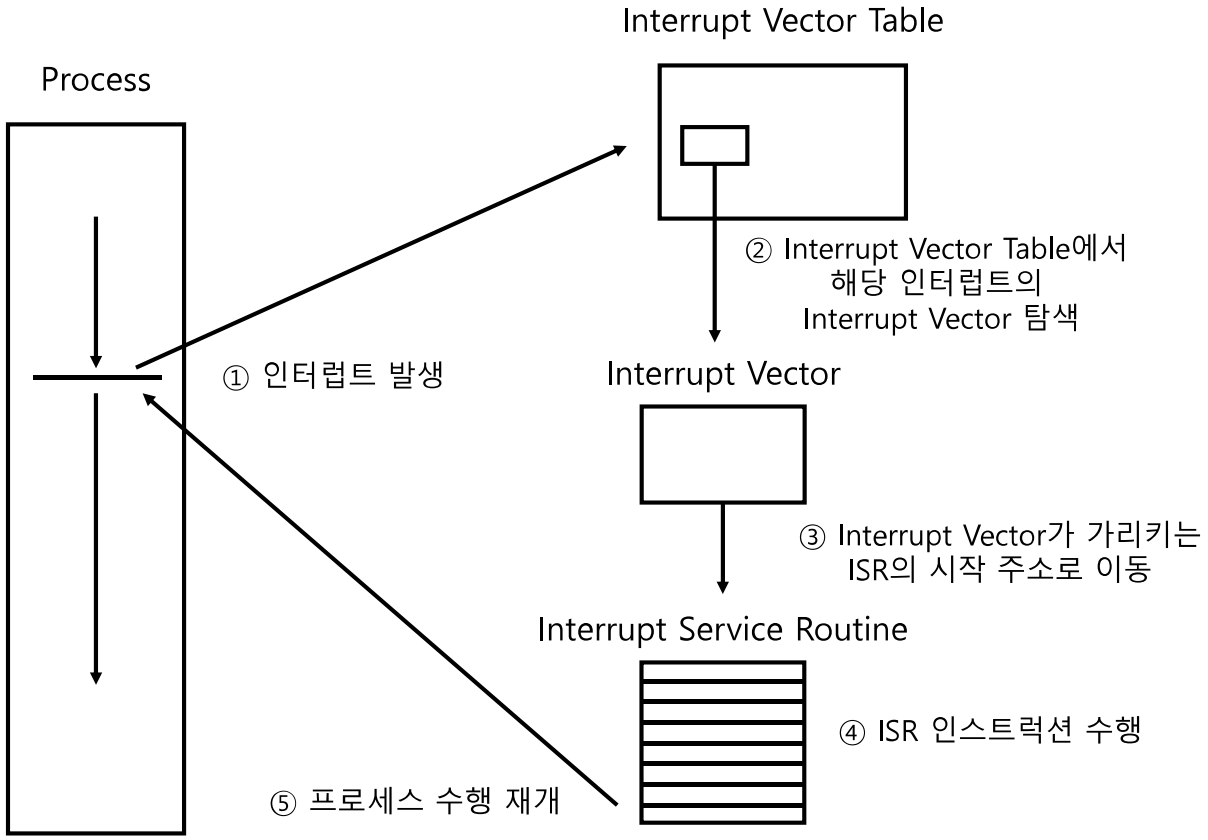

Fig. 2. 리눅스에서 인터럽트가 처리되는 전반적인 과정

그리고, 프로세서는 PC(Program Counter) 레지스터에 다음에 실행할 인스트럭션의 주소를 저장한다. 이는 당연히 인터럽트 처리가 종료되고 기존에 수행하던 프로세스의 인스트럭션으로 돌아오기 위함으로, 어셈블리 계열에서 돌아올 인스트럭션의 주소를 보존하는 CALL 인스트럭션 등과 비슷한 맥락이다.

현재 실행 중인 프로세스의 정보 저장이 끝나고 성공적으로 종료되고 나면, 운영체제에서는 control 권한을 인터럽트 요청 핸들러 (interrupt-request handler)에게 넘겨주고, 현재 수행 중인 프로세스를 후술할 인터럽트를 처리하는 루틴으로 교체하게 된다. 이렇게 프로세서가 이전의 프로세스 상태를 PCB 에 저장하고, 다른 프로세스의 정보를 PCB 에서 읽어들임으로써 프로세스를 교체하는 과정을 '문맥교환(Context switch)'이라 한다.

C. 인터럽트 처리 : IRQ, IPR, ISR

Control 권한이 인터럽트 핸들러에게 주어졌다. 이제 프로세서는 인터럽트 종류를 보고 이에 대응하여 미리 정해진 대로 행동해야 한다. 인터럽트에 대응하여 수행해야 할 프로그램은 어디에 정의되어 있으며, 운영체제는 인터럽트 종류를 가지고 이러한 정보들에 어떻게 접근해야 하는가?

특정한 인터럽트에 대해 프로세서가 수행해야 할 프로그램, 루틴이 바로 ISR(Interrupt Service Routine, 인터럽트 서비스 루틴)이다. ISR 자체도 또한 프로그램으로, 인스트럭션의 집합이다. ISR 은 인터럽트 처리 과정에서 가장 핵심적인 부분이라 할 수 있다. ISR 이 어떻게 구현되어 있느냐에 따라 프로세서가 인터럽트에 어떻게 대응하고 어떻게 행동할지가 결정되며, 인터럽트에 대한 실질적인 처리를 담당하는 부분이 바로 ISR 이라 할 수 있다.

TABLE I. CORTEX-M3 프로세서의 IVT 예시[7]

Exception Type	Vector Number	Priority	Vector Address or Offset
-	0	-	0x0000.0000
Reset	1	-3	0x0000.0004
Non-Maskable Interrupt (NMI)	2	-2	0x0000.0008
Hard Fault	3	-1	0x0000.000C
Memory Management	4	Programmable	0x0000.0010
Bus Fault	5	Programmable	0x0000.0014
Usage Fault	6	Programmable	0x0000.0018
-	7-10	-	-
SVCall	11	Programmable	0x0000.002C
Debug Monitor	12	Programmable	0x0000.0030
-	13	-	-
PendSV	14	Programmable	0x0000.0038
SysTick	15	Programmable	0x0000.003C
Interrupts	16 and above	Programmable	0x0000.0040 and above

그리고, 이러한 인터럽트를 처리할 수 있는 루틴인 ISR 의 시작 주소를 담고 있는 공간이 바로 인터럽트 벡터(Interrupt Vector)이며, 인터럽트 벡터를 모아둔 테이블이 바로 인터럽트 벡터 테이블(Interrupt Vector Table, IVT)이다. 대부분의 프로세서들은 IVT 를 가지고 있으며, 프로세서는 인터럽트 번호를 통해 IVT 에서 해당하는 인터럽트의 인터럽트 벡터를 찾고, 인터럽트 벡터가 가리키는 ISR 의 주소로 이동하여 해당 루틴을 수행함으로써 인터럽트를 처리한다.

지금까지의 전체적인 인터럽트 처리 흐름을 그림으로 정리하면 Fig 2. 와 같다.

IV. 결론 및 의문

운영체제 과목에서 깊게 다루지 않은 '리눅스 운영체제에서의 인터럽트 처리 과정'에 대해 여러 레퍼런스들을 찾고, 혼자서 공부함으로써 기존에 잘못 알고 있던 지식도 바로잡을 수 있었고, 새로운 지식도 알 수 있었다.

운영체제에 인터럽트 개념이 존재하기 전에는 운영체제가 어떠한 방식으로 외부 입출력 장치와 상호작용하였는지, 사용자 입력을 어떻게 받고 처리하였는지에 대해 평소에 궁금증이 있었는데, 이번 보고서를 작성하는 과정에서 기존에 이를 처리하는 방식 중 하나로 폴링이라는 방식이 있다는 것을 처음 알게 되었다. 그리고 이를 인터럽트 방식과 비교해봤을 때, 얼마나 많은 고민 속에서 인터럽트가 탄생한 것인지도 새삼 느끼게 되었다. 지금 와서야 간단해 보이는 방법이지만, 아무것도 없는 밑바닥에서 새로운 해결책을 만들어내는 문제 해결의 과정이 얼마나 어려운지를 새삼 알기에 더욱 놀랄 수밖에 없었다.

다만, 혼자서 공부하면서 주제와 관련하여 생긴 의문점들이 여럿 있었는데, 그중 몇 가지를 적자면 다음과 같다.

- 인터럽트를 처리하는 도중, 다른 인터럽트가 발생할 경우 어떠한 일이 일어나는가? 이러한 행동들을 사용자가 정의할 수 있는가?
- 정전 등의 이유로 컴퓨터의 전원이 갑자기 차단되는 것과 같이, Power 와 관련된 예상치 못한 일들은 어떻게 처리되는가? 이러한 것들도 인터럽트의 영역에서 처리할 수 있는가? 그렇지 않다면, 인터럽트라는 개념으로 설명되지 않는 또 다른 영역의 예외가 있는 것인가?

이러한 의문점들에 대해 나름대로 많은 시간 생각해보고, 여러 레퍼런스를 찾아본 결과, 다음과 같은 결론을 얻을 수 있었다.

- 인터럽트 처리 도중 다른 인터럽트가 발생하는 경우에 대해서는, 중복 인터럽트를 허용하는 프로세서도 있고 그렇지 않은 프로세서도 있었다. 하드웨어의 영역에서 중복 인터럽트의 허용 여부를 설정할 수 있다는 것은 예측하지 못했다. 또한, ISR 의 설계에도 'Non-nested', 'Nested', 'Re-entrant nested' 등 다양한 방식이 존재하고, 이들 간의 장단점도 각각 분석되어 있다는 것을 알았다[8].
- 일단 대부분의 컴퓨터의 경우, 정전 등이 일어나더라도 내장된 배터리 등의 예비 전원을 통해 컴퓨터의 수명을 일정 기간동안 유지할 수 있다. 그리고 Power 문제의 경우도 'Power failure interrupt'라는 이름으로 이와 관련된 인터럽트가 따로 마련되어 있었고, 예비 전원으로 컴퓨터가 가동되는 동안 운영체제는 이 인터럽트를 발생시켜 로그를 남기고, 다음 컴퓨터 가동 시에 추가적인 작업을 수행할 수 있도록 한다.

운영체제 구동의 핵심이라 할 수 있는 인터럽트에 대해 심도 있게 공부함으로써 운영체제 과목 자체에 대한 이해도가 어느 정도 상승했다고 생각하고, 앞으로 다른 운영체제를 분석하는 시각이 더 넓어지기를 소망하며 보고서를 마무리한다.

REFERENCES

[1] http://faculty.salina.k-state.edu/tim/ossg/Introduction/OSworking.html
[2] http://www.slawinski.ca/courses/new_cs30/unit2/part1.htm
[3] Jonathan Corbet; Alessandro Rubini; Greg Kroah-Hartman, "Linux Device Drivers, Third Edition, Chapter 10. Interrupt Handling", p.269, O'Reilly Media (2005).
[4] "interrupt", Cambridge Dictionary. https://dictionary.cambridge.org/us/dictionary/english/interrupt
[5] Daniel Bovet and Marco Cesati, "Understanding The Linux Kernel", Oreilly & Associates Inc, 2005.
[6] Deitel, Harvey M., "An introduction to operating systems", p.673, Addison-Wesley, 1984.
[7] http://download.mikroe.com/documents/compilers/mikroc/arm/help/interrupts.htm
[8] Peng Zhang, "Advanced Industrial Control Technology", William Andrew, 2010