Project #4: 키보드 활성화, 셸 프로그램, 인터럽트 처리 및 예외 처리 구현

신정은, 조재호, 조한주
School of Software, Soongsil University

1. 구현 목표

1.1 키보드 디바이스 드라이버 추가

지금까지 만든 OS는 커널이 정상적으로 로딩되었다는 메시지를 화면에 출력해줄 뿐이었다. 하지만 키보드 디바이스 드라이버를 OS에 추가하게 되면 키보드로 문자를 입력하여 화면에 출력할 수 있다. 우리가 만드는 OS의 사용자와의 상호작용의 첫 번째 단계이다. 이를 위해 키보드와 OS 커널 간의 데이터 송수신을 위한 키보드 컨트롤러는 만든다. 그리고 입력한 문자를 화면에 출력하는 간단한 셸 프로그램으로 키보드 컨트롤러가 정상 작동하는지 확인한다.

1.2 GDT, IDT 테이블, TSS 세그먼트 추가

OS에서 처리해야 하는 이벤트인 인터럽트와 예외에 대해 다뤄본다. 인터럽트가 발생했을 때 프로세서가 이를 처리하기 위해 IDT라는 벡터 테이블의 인덱스에 해당하는 어드레스로 이동하여 처리 함수를 수행하는데, 이 과정을 직접 구현해 보며 인터럽트와 예외 핸들러가 수행되는 과정을 이해한다. 특히 이 과정에서 스택 스위칭이 왜 사용되는지 이해하고 이를 위한 IST, TSS세그먼트 등을 구현한다.

1.3 PIC 컨트롤러, 인터럽트 핸들러를 이용한 인터럽트 처리

PIC컨트롤러를 사용해 PC디바이스에서 인터럽트가 발생했을 때 프로세서에 전달하는 방법을 알아본다. OS는 인터럽트나 예외가 발생한 경우, 처리를 한 후, 실행 중이던 코드로 복귀할 능력이 있어야한다. 이를 위해서는 프로세서의 전체 상태를 저장해야 하는데, 이것을 인터럽트 컨트롤러인 PIC와 인터럽트 핸들러를 이용해 구현한다. 그리고 인터럽트와 예외를 구분하여 개별적으로 핸들러를 등록한다. 콘텍스트를 저장하고 복원하는 역할을 하는 ISR함수마다 비슷한 과정을 반복하므로 매크로를 사용해 구현해본다.

1.4 키보드 디바이스 드라이버 업그레이드

키보드 디바이스 드라이버를 구현하고, 인터럽트와 예외 처리에 필요한 자료구조와 컨트롤러를 구축했다. 이제 키보드 디바이스 드라이버를 인터럽트 기반으로 업그레이드한다. 기존의 키보드 드라이버는 프로세서가 주기적으로 드라이버를 확인해야 한다. 폴링 방식은 주기적으로 키보드로부터 수신하는 데이터를 확인해서 프로세서의 소모가 심하고 주기에 비례해서 수신되는 시간이 길어진다. 하지만 인터럽트 방식으로 키보드 드라이버를 구현하면 컨트롤러에 데이터가 있거나, 입출력이 가능할 때 인터럽트를 통해 이를 알리므로 프로세서의 사용이 줄어든다.

멀티태스킹 환경에서 더욱 인터럽트 방식이 빛을 발한다. 멀티태스킹 환경은 여러 프로그램이 프로세서를 사용하므로 프로그램의 수가 늘어날 수록 프로세서를 할당 받는 주기가 길어진다. 폴링 방식은 작업 수에 비례해 주기가 길어지는 반면, 인터럽트 방식은 프로그램의 수와 관계 없이 데이터를 처리할 수 있으므로 키보드 디바이스 드라이버도 인터럽트 방식으로 변경하여 성능을 개선하기로 한다.

1.5 Exception Handler 추가 구현

Exception Handler를 통해서 paging과 관련된 대표적인 두 가지 exception (page fault exception, protection fault exception)을 확인한다. Exception이 발생할 때 page table entry를 수정하여 재부팅 없이 코드가 정상 동작하도록 한다.

2. 구현 내용

2.1 키보드 디바이스 드라이버 추가

2.1.1 키보드 컨트롤러의 구조와 기능 설정

키보드 컨트롤러는 PC 내부 버스와 포트 I/O 방식으로 연결되도록 구현한다. 이때 PC 내부 버스와의 통신은 포트 어드레스 0x60과 0x64를 사용한다. 2가지 포트를 사용하지만 각 포트에서 읽기/쓰기 시 접근하는 레지스터가 달라, 논리적으로는 4가지의 레지스터가 연결되어있다. 키보드 컨트롤러를 제어하는 컨트롤 레지스터와 키보드 컨트롤러의 상태를 나타내는 상태 레지스터를 0x64포트 위에서 구현한다. 그리고 프로세서에서 키보드로 향하는 데이터를 저장하는 입력 버퍼와 키보드에서 프로세서로 가는 데이터를 저장하는 출력 버퍼 레지스터를 0x60 포트에 구현한다.

가장 단순한 키보드 컨트롤러의 동작은 키 값을 얻기 위해 상태 레지스터를 읽어서 출력 버퍼가 남아있는지 검사하고, 출력 버퍼 레지스터를 읽는 것이다. 하지만 키보드 컨트롤러를 통해 키보드와 마우스를 제어하는 것이 이번 구현의 목표이므로, 키보드 컨트롤러의 커맨드가 필요하다. 키보드 컨트롤러 커맨드로는 입출력 버퍼의 동작, 디바이스의 활성화 여부를 관리할 수 있다.

2.1.2 키보드와 키보드 컨트롤러 활성화

키보드 컨트롤러의 기본적인 구조를 확인한 다음에는 키보드와 키보드 컨트롤러를 활성화해야 한다. 키보드 컨트롤러에서 키보드 디바이스를 활성화하기 위해서 커맨드 포트로 키보드 디바이스 활성화 커맨드 0xAE를 보낸다. 이 커맨드로 인해 키보드 컨트롤러에서 키보드가 활성화 동작을 한다. 키보드 컨트롤러의 입력 버퍼에 키보드 활성화 커맨

드를 써서 키보드로 전송한다. 정상적으로 키보드가 활성화되면 키보드는 ACK를 리턴한다. 이 때, 키보드와 키보드 컨트롤러는 프로세서에 비해 느리게 동작하므로 커맨드를 전송하고 대기한다. 키보드 컨트롤러의 상태 레지스터의 출력 버퍼 상태 비트를 통해 출력 버퍼를 확인하고 실행 결과를 읽어 대기 과정을 끝낼 수 있다. 이를 위해 입력 포트의 상태와 출력 포트의 상태를 확인하는 코드를 작성한다. 이는 자주 사용되므로 함수로 분리하여 작성한다. 출력 버퍼에서 ACK(0xFA)를 수신하면 정상적으로 키보드 및 키보드 컨트롤러 활성화 과정이 완료된다.

2.1.3 IA-32e 모드의 함수 호출 규약

포트 I/O 어드레스로부터 데이터를 읽고 쓰기 위해 어셈블리어 함수를 작성해 C 코드에서 호출한다. IA-32e 모드에서는 보호 모드와 함수 호출 규약이 서로 다르기 때문에 IA-32e 모드의 함수 호출 규약을 확인해야 한다. 보호 모드와 다른 점만 살펴보자면, 첫 번째로, IA-32e 모드에서는 파라미터를 전달할 때 레지스터를 우선으로 사용한다. 정수 타입 6개, 실수 타입 8개의 레지스터를 사용한다. 두 번째로, 레지스터나 스택에 파라미터를 삽입하는 순서가 보호 모드와 반대로, 파라미터 리스트의 왼 쪽에서 오른 쪽으로 이동하면서 파라미터를 삽입한다. 마지막으로 함수의 반환 값으로 사용하는 레지스터가 다르다. 이러한 차이로 보호 모드에 비해 IA-32e 모드에서는 스택을 비교적 덜 사용한다.

2.1.4 키보드 컨트롤러로부터 키 값 읽어 오기

이제 활성화된 키보드 컨트롤러로부터 키 값을 읽어온다. 키가 눌리거나 떨어질 때 값이 키보드 컨트롤러로 전달이 되며 이 값이 스캔코드이다. 키보드 컨트롤러의 출력 버퍼에 키보드로부터 수신된 데이터가 저장된다. 따라서, 상태 레지스터를 읽어 출력 버퍼에 데이터가 있는지 확인하고, 데이터가 있다면, 출력 버퍼를 읽는다. 키보드 컨트롤러의 출력 포트는 키보드와 마우스 이외에, A20 게이트와 프로세서 리셋에 관련된 라인과도 연결되어 있다. 출력 포트의 해당 비트를 1로 설정해 A20 게이트를 활성화하거나 프로세서를 리셋 할 수 있다. A20 게이트 비트는 출력 포트의 비트1, 프로세서 리셋 비트는 비트0에 해당한다. 그리고 키보드 컨트롤러의 출력 포트는 0xD0 0xD1로 접근한다. A20 게이트는 어드레스의 20번 째 비트를 활성화하는 역할을 하며 과거 컴퓨터의 어드레스와의 호환성을 유지하기 위해 사용한다.

2.1.5 키보드 LED 제어

키보드의 LED 상태는 입력 버퍼만을 이용하여 제어한다. 먼저 입력 버퍼 0x60으로 0xED 커맨드를 전송해서 키보드에 LED 상태 데이터가 전송될 예정임을 알린다. 그리고 키보드가 커맨드에 대한 응답으로 ACK를 보내오면 LED 상태를 나타내는 데이터를 전송한다. LED 상태 데이터는 1 바이트 중 하위 3 비트 만을 사용하고 각각 Caps Lock은 2, Num Lock은 1, Scroll Lock은 비트 0에 할당 되어 있다. LED를 켜려면 해당 비트를 1로 설정하면 된다.

2.1.6 키보드 스캔 코드 처리

키보드를 활성화하고 키보드에 전달된 데이터를 처리할 준비가 끝났으니, 키보드로부터 수신한 데이터를 ASCII 코드 형태로 변환하여 화면에 출력한다. 키보드는 스캔 코드를 전달하기 때문에 두 코드 간의 변환이 필요하다. 키보드의 모든 키는 각자의 고유 코드를 가지고 키에 액션이 이루어질 때마다 키보드 컨트롤러로 스캔 코드가 전송된다. 키가 눌렸을 때의 값에 최상위 비트를 1로 취하면 키가 떨어질 때의 코드이다. 확장 키의 스캔 코드는 구형 키보드에 없던, 후에 추가된 키인데 일반 키와 달리 2개 이상의 코드로 구성된다. 0xE0 혹은 0xE1 로 시작하는 공통점이 있으므로 이에 해당하는 스캔 코드가 수신될 경우에 확장 키로 처리한다.

스캔 코드와 ASCII 값이 대응 하는 테이블을 만들어 스캔 코드를 ASCII 값으로 변환한다. 확장 키를 제외한 스캔 코드 88 개의 코드 값을 테이블 인덱스로 사용하여 ASCII 값을 구하는 변환 테이블을 만들어 사용한다.

테이블을 준비하고 실제 키 값을 입력 받아 ASCII 문자로 변환할 때 조합 키의 상태를 고려해야 한다. Shift, Caps Lock, Num Lock의 상태에 따라 ASCII 코드를 매핑할 때 일반 키를 반환할지, 조합 키를 반환할지 판단한다. 뿐 만 아니라 확장 키도 처리해야 한다. 이러한 부분을 반영한 키보드 자료구조를 만들어 키 값을 처리하여 ASCII 문자로 변환한다. 이 때 조합 키 여부와, 확장 키 여부를 판단하고 처리하는 함수를 만들어 처리한다.

2.1.7 키 값을 확인하는 셸 구현

키보드 컨트롤러로부터 스캔 코드를 얻어 ASCII 코드로 변경하는 부분 까지 완성했다. 키보드 처리를 구현했지만 아직 화면으로 출력할 수 없으므로 키 값을 화면에 출력하는 간단한 셸 프로그램을 구현한다. 셸은 사용자의 명령을 받아 작업을 수행하는 프로그램으로 OS와 유저 사이에 위치하는 프로그램이다. 제작한 키보드 드라이버를 테스트하는 것이 주목적이므로 키보드로부터 입력된 스캔 코드를 변환하여 화면에 출력하는 기능만 구현한다.

키보드 디바이스 드라이버와 셸 프로그램을 OS에 추가하여 확인한다. 이 때 C 언어 커널 엔트리 포인트를 수정하여 키보드 컨트롤러와 키보드를 활성화하고셸 프로그램을 실행한다. 오류가 없이 빌드가 되었다면 프로그램을 실행하여 키보드를 눌렀을 때 화면에 정상적으로 키 값이 출력되는 것을 확인할 수 있다.

2.2 GDT, IDT테이블, TSS세그먼트 추가

2.2.1 GDT 테이블 생성과 TSS세그먼트 디스크립터 추가

TSS세그먼트 디스크립터를 추가하기 전에, 기존의 GDT테이블이 있는 보호모드 커널 엔트리 포인트 영역은 104바이트인 TSS세그먼트 디스크립터가 들어가기엔 작을 수 있다. 그리고 멀티코어를 활성화하게 된다면 코어별로 TSS세그먼트가 필

요하므로 1MB이상의 공간에 GDT테이블과 GDT정보를 나타내는 자료구조를 생성한다. 보호모드때 구현했던 코드와 데이터 세그먼트 디스크립터, 그리고 이번에 구현할 TSS세그먼트 디스크립터와 TSS세그먼트로 사용할 자료구조와 매크로를 C언어로 구현한다. 이때 디스크립터를 생성하는 함수는 파라미터로 넘어온 필드 값을 세그먼트 디스크립터의 구조에 맞춰 삽입하는 역할을 한다.

생성한 후에는 TSS세그먼트를 초기화해야 한다. MINT64 0S는 I/O맵을 사용하지 않고 스택 스위칭 방식으로 IST방식을 사용하므로 이에 맞추어 TSS 세그먼트 필드를 설정한다. I/O 맵 기준주소를 TSS 세그먼트 디스크립터에서 설정한 Limit 필드 값보다 크게 설정하여 I/O맵을 사용하지 않게 설정한다. 그리고 IDT 게이트 디스크립터의 IST필드 값을 0이 아닌 값으로 설정하고 TSS 세그먼트의 해당 IST영역에 핸들러가 사용할 스택 어드레스를 설정하여 IST를 설정한다.

2.2.2 GDT테이블 교체와 TSS세그먼트 로드

GDT테이블은 LGDT명령어를 사용하여 갱신할 수 있다. TSS세그먼트를 로드하려면 태스크에 관련된 정보를 저장하는 TR(Task Register)에 LTR명령어를 이용하여 TSS세그먼트 인덱스인 0x18을 지정하면 된다. 이렇게 TSS세그먼트를 프로세스에 설정한다. 이 과정에서 사용하는 LGDT, LTR명령어는 어셈블리어 명령이므로 어셈블리어 함수를 만들어 C코드에서 호출한다.

2.2.3 IDT테이블 생성, 인터럽트, 예외 핸들러 등록

인터럽트나 예외처리 방법이 정의된 IDT테이블을 생성해보자. IDT테이블은 IDT게이트 디스크립터로 구성된다. IDT게이트 디스크립터의 자료구조와 매크로와 IDT게이트 디스크립터를 생성한다. 디스크립터를 생성하는 함수에서 postEntry파라미터는 IDT게이트 디스크립터의 어드레스를 넘겨주는 용도로 사용한다. bIST파라미터를 1로 설정해 IST를 사용하도록 명시한다. 또한 우리의 MINT64 OS의 모든 핸들러는 커널 레벨에서 동작하고, 핸들러 수행 중에 인터럽트가 발생하는 것을 허락하지 않으므로 앞에서 정의한 매크로도 파라미터로 넘겨 flag로 설정해 준다.

IDT테이블을 생성했으면 LIDT명령어를 통해 프로세서에 로드시켜 인터럽트나 예외가 발생했을 때 참조하도록 한다.

2.3 PIC컨트롤러와 인터럽트 핸들러로 인터럽트 처리

2.3.1 PIC컨트롤러 제어

지금의 MINT64 OS의 인터럽트와 예외 핸들러는 아직 복귀 능력이 없다. 따라서 인터럽트 처리를 목적으로 PIC컨트롤러를 통해 PC디바이스의 인터럽트를 프로세서에 전달하는 것을 구현해 본다. 우선 ICW커맨드를 통해 마스터PIC와 슬레이브PIC를 초기화 한다. ICW1커맨드는 트리거모드와 캐스케이드여부, ICW4의 사용여부를 설정하고, ICW2커맨드는 인터럽트가 발생했을 때 프로세서에 전달할 벡터번호를 설정하는 역할을 한다. ICW3커맨드는 마스터 PIC컨트롤러의 몇 번 핀에 슬레이브 PIC컨트롤러가 연결되었는지를 설정한다. 마지막으로 ICW4커맨드는 EOI전송모드와 8086모드, 확장기능을 설정한다. 이렇게 4가지의 ICW커맨드의 필드를 설정하면 PIC컨트롤러의 초기화 작업이 끝난다.

OCW1커맨드를 통해 특정 인터럽트 설정 기능을 추가한다. 특정 인터럽트를 설정하려면 IMR 레지스터에 무시할 인터럽트를 1로 설정해야 한다. 마스크를 설정하려면 IRQ번호에 따라 마스터나 슬레이브 PIC컨트롤러를 구분하여 처리해야 한다.

인터럽트가 발생했음을 PIC컨트롤러가 프로세서에 알려주면 프로세서는 핸들러를 실행하여 인터럽트 처리를 완료한 후 다시 PIC컨트롤러에 이를 알려야 하므로 OCW2커맨드를 사용해 PIC컨트롤러에 EOI를 전송한다. 우리는 PIC컨트롤러가 알려준 인터럽트의 종료여부만 전송하면 되므로 EOI 비트만 1로 설정하여 대상을 지정하지 않는 EOI커맨드만 사용한다. 이 때 슬레이브 PIC컨트롤러에 EOI를 전송하면 마스터 PIC컨트롤러에도 EOI를 전송해야 한다.

2.3.2 콘텍스트 저장과 복원

인터럽트나 예외가 발생하고 핸들러가 수행된 후 다시 실행 중이던 코드로 복귀해야 한다. 이를 위해서는 핸들러로 이동하기 전 프로세서의 상태를 저장하고, 핸들러 처리가 끝난 후 복원해야 한다. 이렇게 프로세서의 상태와 관계된 레지스터의 집합인 콘텍스트 중 프로세서가 처리하는 부분을 제외하고 핸들러가 처리하는 부분을 저장하고 복원하도록 한다.

프로세서가 복원하는 콘텍스트가 이전에 저장한 콘텍스트와 같은 것인지 비교하는 기능으로 인터럽트와 예외 핸들러를 업그레이드 해 본다. 콘텍스트를 저장하고 복원하는 역할을 하는 ISR함수는 인터럽트 처리를 위해 16개를 작성해야 하는데 반복되는 코드가 많으므로 매크로를 활용한다. NASM에서 매크로를 정의하기 위해 %macro와 %endmacro사이에 매크로 이름과 파라미터 수, 코드를 삽입한다.

이렇게 인터럽트와 예외 핸들러를 추가했으므로 IDT테이블에 등록하는 핸들러 함수를 변경한다. 마지막으로 프로세서의 RFLAGS레지스터의 IF비트는 인터럽트 발생 가능여부를 알려준다. STI명령어와 CLI명령어를 통해 인터럽트 활성화, 비활성화하는 함수를 구현하고, RFLAGS레지스터를 스택에 저장하는 PUSHF명령어를 통해 프로세서의 상태를 반환하는 함수를 구현하여 OS가 인터럽트를 처리하게 한다.

2.4 키보드 디바이스 드라이버 업그레이드

키보드 디바이스 드라이버를 인터럽트 방식으로 변경하여 프로세서를 사용하는 빈도를 줄여 전체적인 OS의 성능을 개선한다. 이를 위해 셸 코드의 키 처리 부분을 인터럽트 핸들러로 옮긴다. 인터럽트는 예측할 수 없으므로 코드의 어느 부분을 실행하는지 알 수 없다. 하지만 셸 코드가 키를 처

리하려면 인터럽트 핸들러가 읽은 키 값이 필요하다. 인터럽트 핸들러가 디바이스에서 읽은 키 값을 버퍼에 저장하고 프로그램이 버퍼를 확인하여 이를 처리한다. 이렇게 키 값을 전달할 수 있다. 버퍼를 계속 확인하는 것과 기존 폴링 방식의 차이는, 외부의 컨트롤러 I/O에 비해 메모리와의 입출력이 훨씬 빠르다는 것이다. 따라서 인터럽트 방식이 폴링 방식보다 유리하다.

2.4.1 범용 큐 자료구조 구현

인터럽트 핸들러가 사용할 버퍼를 구현하기 위해 큐 자료구조를 사용한다. 큐는 앞으로 OS에서 다양한 목적으로 사용하므로 범용으로 사용할 수 있는 큐 자료구조를 설계하고 구현한다. 그리고 자료구조에 맞추어 초기화, 삽입 및 제거를 수행하는 함수를 작성한다. 큐는 반드시 사용 전에 초기화를 수행한다. 초기화가 되지 않으면 잘못된 어드레스에 접근해 OS가 오동작할 수 있다.

2.4.2 키 정보를 저장하는 자료구조와 큐 생성

언제 인터럽트를 처리할지 모른다는 인터럽트 방식의 문제를 해결하기 위해, 버퍼로 사용할 자료구조로, 큐를 준비했다. 이제 키보드의 키 정보를 저장하는 큐를 생성한다. 그리고 폴링 방식으로 작성된 키보드 디바이스 드라이버를 인터럽트 방식으로 변경한다. 키 정보를 전달하는데 사용할 자료구조를 정의하고, 키를 저장하는 큐를 생성한다. 키를 저장하는 자료구조에는 ASCII 값 필드와, 키 상태 필드, 그리고 키보드에서 전달되는 스캔 코드 필드가 있다. 키를 저장하는 큐를 생성할 때는 키보드를 활성화하는 함수를 큐를 생성하는 함수와 하나로 통합한다. 기존의 키보드 핸들러에 키를 읽어 큐에 삽입하는 역할을 추가한다. 셸 코드의 키 처리 부분을 키보드 핸들러로 옮겨주고 키를 큐에 삽입하는 코드를 추가한다. 키 값을 처리하는 부분이 셸 코드에서 키보드 핸들러 함수로 이동한다. 따라서 키보드 컨트롤러 대신 큐를 참조하게 셸 코드를 수정한다.

2.4.3 인터럽트 제어와 키보드 컨트롤러의 충돌 해결

지금까지 구현한 인터럽트 기반의 키보드 디바이스 드라이버는 문제가 있다. 먼저 큐에 입출력이 있을 때 인덱스에 문제가 생길 수 있다. 예를 들어, 셸 코드가 제거 동작을 수행하여 제거 인덱스가 변경된 순간, 인터럽트가 발생하여 인터럽트 핸들러가 수행되어 키보드 컨트롤러에서 수신한 키 값을 큐에 삽입한다. 이 때 마지막으로 수행한 명령이 삽입이라는 플래그로 설정된다. 인터럽트 처리가 끝나고 복귀할 때 아직 셸 코드가 끝나지 않아, 수행 명령 플래그를 제거로 바꾼다. 제거 후 삽입 동작이 수행되었으므로 큐가 가득 찬 상태여야 하지만, 인터럽트 때문에 큐에 수행한 하지만 명령이 제거로 변경되어 큐의 상태는 가득 찬 상태가 아닌 완전히 비어 있는 상태가 된다.

두 번째 문제는 키보드 활성화나 LED 컨트롤 같은 제어 명령의 ACK 를 인터럽트가 가로챌 수 있다는 것이다. 이전 코드는 키보드로 제어 명령을 전송하고 나서 일정 시간 ACK 를 기다려 성공 여부를 판단했다. 하지만 ACK 를 인터럽트 핸들러가 처리한다면 키보드 활성화 함수와 LED 컨트롤 함수는 ACK를 수신하지 못해 실패한다.

위의 두 가지 문제의 공통점은 인터럽트가 필요하지 않은 동작에서 인터럽트가 발생한다는 것이다. 이는 해당 동작을 할 때 인터럽트를 발생하지 못하게 해야한다. 하지만 인터럽트가 이미 비활성화인지 알아야하므로 이전의 인터럽트 상태를 저장했다가 복원한다. 활성화나 비활성화를 수행할 때 이전 인터럽트 플래그의 상태를 반환하므로 이를 활용하여 인터럽트 플래그를 제어한다.

2.4.4 키보드 디바이스 드라이버 수정 및 업그레이드

인터럽트 방식의 문제점을 해결했으므로 키보드 디바이스 드라이버를 인터럽트 기반으로 수정한다. 큐에 데이터가 입출력 될 때 인터럽트 플래그를 조작해서 인터럽트가 발생하지 않게 한다. ACK 처리와 관련된 부분은 ACK 를 대기하는 동안 수신된 키를 확인하여 큐에 삽입하는 함수를 추가한다. 그리고 인터럽트를 비활성화하는 부분을 추가하고 ACK 처리하는 부분을 수정한다.

이제 인터럽트 기반으로 변경된 키보드 디바이스 드라이버를 OS에 통합하고 결과를 확인한다. 기존의 화면과 비교해서 변한 것은 없지만 이제 멀티태스킹 환경에서도 키보드 이벤트를 효율적으로 처리할 수 있다.

2.5 Exception Handler 추가 구현

Page fault, Protection fault 모두 아래의 흐름을 따른다.

(1). 0x1ff000를 매핑하고 있는 페이지 테이블 엔트리를 수정하여 에러가 발생하게 수정한다.
(2). IDT 테이블의 index 14 에 있는 예외 핸들러를 수정한다.
(3). Error code로 page fault와 Protection fault를 구분하여 각각 예외 핸들러를 호출한다.
(4). 코드가 정상 수행되도록 페이지 테이블 엔트리를 수정하고 invlpg 함수를 호출하여 해당 page와 관련하여 TLB에 cache된 것을 invalidation한다.

(ㄱ). Page fault

주소 0x1ff000에서부터 4K만큼 매핑하고 있는 페이지 테이블 엔트리를 non-present한다. 숫자 ‘0’을 누르면 유효하지 않은 주소값 0x1ff000에 ‘0’이라는 값을 쓰도록 하여 Page fault를 발생시킨다. Page fault가 발생하면 IDT 테이블의 index 14에 있는 예외 핸들러를 수행한다. 따라서 ISR.asm 파일에서 Page Fault ISR 함수가 호출된다.

```
mov rdi, cr2
mov rsi, qword [ rbp + 8 ]
call kDistinguishException
```

이 함수에서 어떠한 exception이 발생했는지 구분해 주는 kDistinguishException 함수를 호출한다. 이 때 exception이 발생한 위치를 알려주는 cr2레지스터와 어떠한 exception이 발생했는지 알려주는 Error Code를 파라미터로 보낸다.

kDistinuishException 함수에서 파라미터로 받은 Error Code와 페이지가 유효한지 나타내는 0x00000001를 &연산하여 0 이면 Page fault이므로 예외 핸들러 kPageFaultExceptionHandler를 호출한다.

```
void kDistinguishException(int iVectorNumber, QWORD qwErrorCode)
{
    int page_mask = 0x00000001;
    int protection_mask = 0x00000002;

    if((qwErrorCode & page_mask) == 0 )
    {
        kPageFaultExceptionHandler(iVectorNumber,qwErrorCode);
    }
    else if((qwErrorCode & protection_mask) == 2)
    {
        kProtectionFaultExceptionHandler(iVectorNumber,qwErrorCode);
    }

}
```

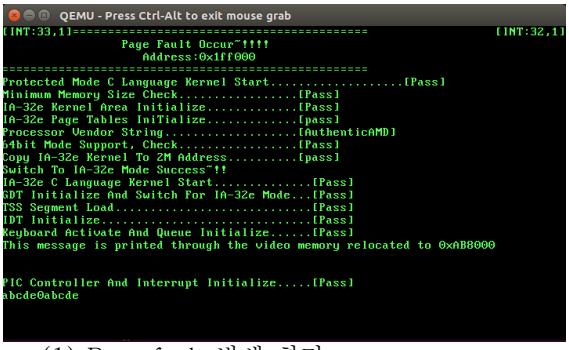
kPageFaultExceptionHandler 함수에서 에러가 발생한 주소를 에러메시지와 함께 출력한다. 에러 메시지 출력 후 해당 page table entry를 유효하게 하고 읽기와 쓰기가 가능하도록 수정한다. invlpg함수를 호출하여 TLB에 cache된 것을 invalidation해 재부팅 없이 코드가 정상 동작하도록 한다.

(ㄴ). Protection fault

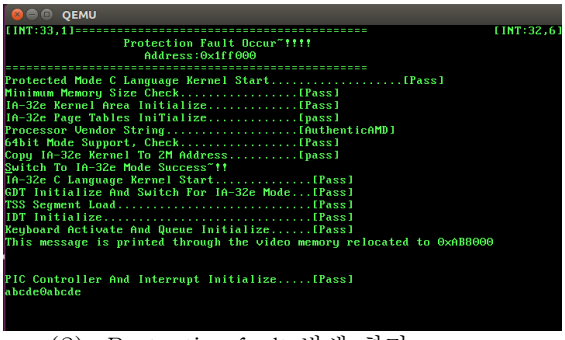
주소 0x1ff000에서부터 4K만큼 매핑하고 있는 페이지 테이블 엔트리를 read-only로 바꾼다. 숫자 '0' 을 누르면 read-only인 0x1ff000에 '0' 이라는 값을 쓰도록 하여 Protection fault를 발생시킨다. Page과 관련된 Protection fault가 발생하면 IDT 테이블의 index 14에 있는 예외 핸들러를 수행한다. 따라서 ISR.asm 파일에서 Page Fault ISR 함수가 호출된다. Page fault와 같이 어떤 exception인지 구분해 주는 kDistinguishException 함수가 호출되고 Error code의 마스킹 연산 결과가 2이면 Protection fault이므로 kProtectionFaultExceptionHandler 함수가 호출된다.

kProtectionFaultExceptionHandler 함수에서 에러가 발생한 주소를 에러메시지와 함께 출력한다. 에러 메시지 출력 후 해당 page table entry를 유효하게 하 읽기와 쓰기가 가능하도록 수정한다. invlpg함수를 호출하여 TLB에 cache된 것을 invalidation해 재부팅 없이 코드가 정상 동작하도록 한다.

(ㄷ). 테스트 결과



(1) Page fault 발생 화면



(2) Protection fault 발생 화면

개발 시 경험한 문제점 및 해결점

1. invlpg명령어의 사용법을 제대로 알지 못한 점

QEMU에서는 TLB가 엄격하게 emulation되지 않아서 invlpg 명령어 없이도 동작하였다. 하지만 정확한 구현을 위해 invlpg 명령어를 사용해야 했다. 하지만 invlpg 명령어를 제대로 알지 못해 어셈블리에서 해야 하는지, c 언어로 해야 하는지 몰라 헤매었다. 이 문제는 c언어로 invlpg 함수를 구현하여 해결하였다.

3. 조원별 기여도

신정은: 33%
조재호: 33%
조한주: 34%