Project #5: 타이머 인터럽트 활성화, 멀티태스킹 구현 및 셸 업그레이드

신정은, 조재호, 조한주
School of Software, Soongsil University

1. 구현 목표

1.1 콘솔 셸 구현

업그레이드 된 키보드 디바이스 드라이버를 이용하여 커맨드를 입력 받아 작업을 수행하는 셸을 만든다. 세 가지 기본 기능을 수행하는 셸 프로그램을 만든다. 첫 번째는 키를 입력 받아 커맨드 버퍼를 관리하고 프로그램을 실행하는 것이다. 이는 셸의 기본 기능이다. 이 때 커맨드 버퍼에서 커맨드와 옵션을 구분하여 실행하는 것이 주요하다. 두 번째는 텍스트 화면을 관리하는 부분이다. 커서의 위치를 제어하고 화면에 데이터를 출력하는 기능을 구현한다. 세 번째는 커맨드에 해당하는 프로그램을 실행하고 커맨드의 처리 권한을 실행할 프로그램으로 넘긴다. 셸 자체 뿐 만 아니라 입출력을 담당하는 라이브러리를 작성한다.

1.2 타이머 디바이스 드라이버 구현

타이머 디바이스는 PIT 컨트롤러의 IRQ 0에 연결되어 있으며, 일정한 주기로 인터럽트를 발생한다. MINT64 OS에서 시간 측정, 시분할 멀티태스킹에 사용한다. 타임 스탬프 카운터는 프로세서의 내부에 있고 프로세서의 클록을 기준으로 카운터가 증가한다. 다른 카운터보다 클록이 빨라서 정밀한 시간 측정이 가능하고, 함수의 수행 시간 측정에 사용한다. RTC는 PC의 시계로 실제 시간을 기록한다. PC가 꺼져 있어도 시간을 계산하도록 별도의 전원이 들어있다. 현재 시간은 RTC 컨트롤러의 값을 사용한다. 따라서 PIT 컨트롤러, 타임 스탬프 카운터, RTC 컨트롤러를 사용하는 코드를 작성하여 타이머 디바이스 드라이버를 구현한다.

1.3 멀티태스킹 구현

멀티태스킹은 멀티와 태스킹의 합성어로 다수의 태스크를 동시에 실행하는 것이다. OS에서 멀티태스킹은 여러 개의 프로그램을 동시에 수행하는 것을 의미한다. 태스크는 프로세스의 관점에서 본 작업의 단위로 메모리에 로드하여 실행할 수 있는 코드, 데이터, 컨텍스트를 저장하는 자료 구조가 있다. OS의 관점에서 태스크는 유저 프로그램, 프로세스, 커널이 있고 멀티스레딩이 있다면 개별 스레드도 태스크이다. 간단한 멀티태스킹을 구현하여 가장 기본적인 기능을 테스트한다.

1.4 셸 업그레이드

셸에 새로운 명령어를 추가하고 기능을 보강한다. 먼저 bash 셸, zsh 셸 등에 구현된 유용한 기능인 history 기능을 구현한다. 위/아래 화살표 키를 누르면 기존의 입력했던 명령어가 자동으로 입력할 수 있다. 그리고 명령어의 수행 시간을 측정하는 time 커맨드를 구현한다. time <command>를 입력하면 해당 명령어가 수행되면서 동시에 해당 명령어의 수행 시간이 마지막에 출력된다. 그리고 현재 시간 및 직전에 수행한 task의 수행시간을 출력하는 기능을 추가한다.

2. 구현 내용

2.1 콘솔 셸 구현

2.1.1 sprintf()와 가변 인자 처리

sprintf()는 주어진 형식에 맞추어 문자를 버퍼에 줄력한다. 이는 콘솔 모드 뿐만 아니라 GUI 모드에서도 쓰인다. 이 때 갯수가 정해지지 않은 가변 인자가 주어진다.

2.1.2. 포맷 스트링과 가변 인자

포맷 스트링에 다른 파라미터를 어떻게 해석하는지를 기술한다. 첫 번째 파라미터 외의 파라미터는 개수와 데이터 타입이 정해져 있지 않고, 대신 포맷 스트링에 전달된 문자열에 포함된 요소들에 따라 파라미터 개수와 데이터 타입이 결정된다. 함수를 가변적으로 처리하려면, 호출하는 함수에서 인자를 모두 스택에 넣어 전달하고, 호출되는

함수는 포맷 스트림을 참조하여 스택에서 파라미터를 직접 참조한다. C 언어는 가변 인자를 처리하는 리스트 데이터 타입과 3 가지 매크로를 제공한다. va_list 자료구조에 파라미터들을 담는다. va_start()로 va_list 를 초기화한다. va_arg()로 가변 인자를 꺼낸다. va_end()로 가변 인자 사용 종료를 한다. 이러한 매크로와 자료 구조는 GCC의 stdarg.h에 선언되어 있는데 자료 구조는 int로, 매크로는 내부 함수를 사용하기 때문에 라이브러리가 따로 필요 없다. 따라서 가변 인자 매크로를 사용하여 MINT64 OS를 구현할 수 있다.

2.1.3 sprintf(), vsprintf() 구현

sprintf()는 printf()와 거의 같지만 결과를 문자열 버퍼에 출력한다는 것이 다르다. printf()는 vsprintf()로 구현할 수 있다. vsprintf()는, sprintf()의 내부에서 동작하는 함수로, 가변 인자(...) 대신 va_list 를 인자로 받아 작업을 한다. 가변 인자(...) 대신 가변인자 리스트를 받는 이유는 가변 인자(...)를 다른 함수 호출시 사용할 수 없기 때문이다. vsprintf()는 포맷 스트링과 가변 인자 리스트를 받아 포맷 스트링 형식에 따라 출력 버퍼를 채운다. 원래 C 라이브러리의 포맷 스트링은 다양한 타입을 지원하지만 빠른 구현을 위해 몇가지 타입으로 제한하여 구현한다. 이번에 구현할 자료형은 문자, 문자열, 정수, 포인터 타입이다. 지원하는 데이터 타입에 다라 처리하는 방법이 다르다. 문자와 문자열 타입은 파라미터가 이미 ASCII이기 때문에 그대로 출력 버퍼에 복사한다. 정수나 포인터 타입의 경우에는 부호의 유무와 데이터의 크기에 따라 처리가 다르므로 itoa()와 atoi()를 구현하여 이를 vsprintf() 내부에서 사용한다. vsprintf()를 위한 데이터 타입, 처리 방법을 준비 했으므로, vsprintf()를 구현할 수 있다. vsprintf()는 포맷 스트링을 따라가면서 데이터 타입 문자열은 가변 인자의 값으로 확장하고 나머지는 출력 버퍼에 그대로 복사하는 방법으로 구현한다.

2.1.4 itoa() 와 atoi() 구현

itoa()는 정수를 문자열로 변환한다. atoi()는 문자열을 정수 값으로 변환한다. 즉, 서로 반대이므로 하나를 구현하고 코드를 역순으로 처리하는 식으로 구현할 수 있다. itoa()는 정수를 10진수, 16진수 문자열로 변환하므로 진법을 파라미터로 받는다. C 라이브러리는 여러 진법을 지원하지만 축소하여 10진수, 16진수 변환만 구현한다. itoa()는 정수 값, 출력 버퍼, 그리고 진법을 파라미터로 전달받아 문자열로 변환한 길이를 반환한다. 내부에서 HexToString()과 DecimalToString()이 실질적인 역할을 한다. 각 함수가 정수를 문자열로 변환한다. 정수 값을 문자로 변환할 때 1의 자리를 나머지 연산으로 추출하여 버퍼에 담는 방식을 사용한다. itoa()의 반대의 경우 인 atioi() 는 HexStringToQword()와 DecimalStringToLong() 이 실질적인 기능을 한다. 각각 16진수 문자열과 10진수 문자열을 정수로 변환한다. 이는 itoa() 내부 함수들의 내용을 역순으로 작성하면 된다.

2.1.5 콘솔 입출력 처리

콘솔은 텍스트 방식이나 명령줄 방식으로 작업을 수행하는 입출력 장치다. MINT64 OS 에서 모니터가 출력 장치이며, 키를 입력하는 키보드가 입력 장치이다. 셸은 이러한 콘솔 환경에서 동작하므로 콘솔 입출력을 담당하느 라이브러리를 구현한다.

2.1.6 콘솔 자료구조 생성과 printf() 함수 구현

콘솔에는 커서가 있는데, 커서는 현재 문자가 출력될 위치를 나타낸다. '-'처럼 생긴 깜빡이는 기호이다. 커서의 위치를 나타내는 자료 구조의 구현이 필요하다. 자료 구조에는 출력할 문자의 위치 정보만 포함된다. 그리고 그외 비디오 작업에 필요한 설정들을 정리한다. 콘솔 자료구조를 바탕으로 printf()를 구현한다. 앞의 vsprintf()를 사용하고, 추가로 화면 출력 제어문자를 구현한다. ₩n, ₩t 등은 현재 커서의 위치를 옮김으로써 구현한다. 화면 스크롤 처리도 모든 텍스트를 위아래로 옮겨서 구현할 수 있다. 제어 문자 처리와 스크롤 기능을 ConsolePrintString()으로 구현한다. 이제 vsprintf(), ConsolePrintString()과 후에 나올 SetCursor()로 printf()를 구현한다.

2.1.7 커서 제어

커서는 모니터에 '_' 문자로 표시되며 입력하는 텍스트가 출력될 위치를 표시한다. 커서는 모니터 출력을 담당하는 VGA 컨트롤러를 사용하여 구현한다. 커서 제어와 관련된 기능은 VGA 컨트롤러의 CRTC 컨트롤러 어드레스 레지스터, CRTC 컨트롤러 데이터 레지스터가 담당한다. 각각 I/O 포트 0x3D4, 0x3D5를 사용한다. CRTC 컨트롤러 어드레스

레지스터는 CRT 컨트롤러를 지정한다. 커서의 위치는 0xE와 0xF를 전달하여 상위 커서, 하위 커서 위치 레지스터를 선택하고 CRTC 컨트롤러의 데이터 레지스터에 상위 바위트와 하위 바이트로 나누어 커서 위치를 전달하면 해당 위치로 커서가 이동한다. CRT 컨트롤러는 비디오 메모리 내에 위치할 오프셋을 사용한다. SetCursor() 함수에서 커서의 위치를 설정하고 GetCursor 함수에서 현재 커서의 위치를 반환한다.

2.1.8 getch() 구현

기존의 키 처리 코드는 큐로부터 데이터를 직접 읽어 처리했지만 셸은 키가 눌렸다는 것과 키의 ASCII 코드만 필요하므로 getch() 로 키가 눌렸을때 ASCII 값만 반환받는다.

2.1.9 프롬프트, 커맨드 버퍼, 사용자 입력 처리

지금까지 셸을 준비하기 위한 라이브러리를 만들었다. 이제 셸을 본격적으로 구현한다. 프롬프트는 셸이 사용자로부터 키를 입력받을 준비가 되어있다는 표시이다. 리눅스에서는 '$_₩ >' 식으로 나타난다. MINT 64 OS 에서 셸이 사용자의 입력을 3 가지로 나누어 처리한다. 첫 번째는 알파벳이나 숫자 같은 셸 커맨드를 입력하는 것이다. 셸은 이를 화면에 표시하여 정상 처리를 나타낸다. 두 번째는 엔터 키와 백스페이스 처럼 입력된 커맨드를 조작한다. 해당 키가 입력되면 명령을 실행하거나 출력된 문자를 삭제한다. 세 번째는 셸에서 사용하지 않는 키이다. 셸은 이런 키를 무시한다. 이번에는 Shift, Caps Lock, Num Lock, Scroll Lock 키를 제외한 모든 키를 사용한다. 앞에서 작성한 라이브러리 덕분에 간단하게 셸 화면을 구현할 수 있다. 하지만 화면에 출력하는 것 외에 커맨드를 실행해야 한다. 사용자의 입력을 프롬프트에 표시함과 동시에 커맨드 버퍼에 삽입하여 관리해야 한다. 커맨드 버퍼는 사용자의 입력을 저장해 실행할 때 참조하려고 관리하는 버퍼이다. 키를 입력할 때 마다 셸이 화면과 커맨드 버퍼에 데이터를 반영하고, 커맨드 버퍼의 상태를 확인하여 데이터 추가와 삭제 가능 여부를 판단한다. 셸 루프에 커맨드용 버퍼를 생성하고, 키를 추가하거나 삭제할 때마다 버퍼에 동시 반영한다. 그리고 엔터 키를 입력했을 때 버퍼를 참조하여 커맨드를 실행하는 코드까지 넣어 루프를 완성한다.

2.1.10 커맨드 비교와 커맨드 실행

커맨드 버퍼에 있는 커맨드를 추출하여 실행하는데 함수 포인터를 사용한다. SHELLCOMMANDENMTRY 라는 자료구조에 커맨드와 사용 방법, 커맨드가 일치했을 때 실행하는 함수의 어드레스를 포함한다. 따라서 커맨드를 실행하는 함수는 해당 자료구조만 참조하고 커맨드의 개수에 의존하지 않는다. 자료구조를 테이블 형태로 만들어 커맨드만 계속 추가하면 된다.

2.1.11 코드 통합과 빌드

콘솔 라이브러리와 셸 코드를 MINT64 OS에 통합하고 결과를 확인한다. 콘솔 파일에는 콘솔 라이브러리에 관련된 함수와 매크로가 정의되어 있다. 앞에서 정의 한 함수 외에도, 화면을 지우는 함수, 라인에 문자열을 출력하는 함수, 전원을 종료하는 함수를 추가하여 PC를 재시작할 수 도 있다. 콘솔 셸 파일에는 실제 셸 기능을 하는 함수와 매크로를 정의하고 커맨드 버퍼에서 파라미터를 추출하는 함수를 추가한다. printf()와 sprintf()를 구현하는데 사용되는 부가 함수를 정의한다. 그리고 보호 모드에서 작성한 메모리 크기 검사 함수를 기반으로 PC에 설치된 전체 메모리 크기를 검사하는 함수를 추가한다. 기존의 kPrintString() 함수가 콘솔 셸로 옮겨지고 kPrintStringXY()로 이름이 변경되었으므로 인터럽트 핸들러를 수정해야 한다. 모든 메시지 출력부를 이번에 작성한 콘솔 라이브러리로 변경한다. 이제 빌드하면 shutdown, hel, strtod, totalram 등 커맨드를 실행하고 결과를 확인할 수 있다.

2.2 타이머 디바이스 추가

2.2.1 PIT 컨트롤러, I/O 포트, 레지스터

PIT 컨트롤러는 1개의 컨트롤 레지스터와 3개의 카운터로 구성된다. 각각 0x43, 0x40, 0x41, 0x42에 연결되어 있는데 컨트롤 레지스터는 쓰기 전용이다. PIT 컨트롤러의 컨트롤러 레지스터의 크기는 1바이트이고 카운터 레지스터의 크기는 2바이트이다. I/O 포트는 PIT 컨트롤러와 1바이트 단위로 전송하고, 컨트롤 레지스터에 전달하는 커맨드에 따라 카운터 I/O 포트로 읽고 쓰는 바이트 수가 결정된다. 컨트롤 레지스터는 4개의 필드로 구성되어 있다.

PIT 컨트롤러는 내부 클록에 맞게 매회마다 각 카운터의 값을 1씩 감소시켜 0이 되었을 때

신호를 발생시킨다. 신호를 발생시킨 이후에 PIT 컨트롤러는 설정된 모드에 따라 다르게 동작한다. PIT의 모드중 MINT 64 OS 에서는 모드 0과 모드 2를 사용한다.

모드 0은 카운터에 입력된 값을 매 회마다 1씩 감소시켜 카운터가 0이 되면 외부로 신호를 발생시킨다. 일정 시간이 지난 것을 체크할 때 사용한다. 모드 2는 모드 0과 같지만 카운터에 새로운 값을 넣지 않아도 자동 반복하기 때문에 반복 작업에 사용한다.

PIT 컨트롤러에 카운터가 3개 있지만 MINT64 OS에서는 카운터 0만 사용한다. 카운터 0은 완료 신호를 출력하는 OUT 핀이 PIC 컨트롤러의 IRQ 0에 연결되었다는 의미이다. 이제 카운터 0을 사용하면 일정한 주기로 인터럽트를 발생시킬 수 있다.

2.2.2 PIT 컨트롤러 초기화

PIT 컨트롤러를 이용해서 시간을 측정하려면 시간을 PIT 컨트롤러의 카운터 값으로 변환해야 한다. PIT 컨트롤러의 내부 클록이 1.193182MHZ로 동작하므로 한 클록은 1초를 내부 클록으로 나눈 약 838.095ns이다. 따라서 측정할 시간을 이 값으로 나누면 카운터에 설정할 값을 얻는다. 또 다른 방법은 1초에 1193182번 카운터가 증가하므로 이 값에 시간을 곱한다. PIT 컨트롤러의 카운터 최대 값은 0x10000으로 최대 시간은 54.93ms이다.

PIT 컨트롤러 초기화에는 반복 여부를 설정해야 한다. 한 번만 확인한다면 모드 0 을 주기적으로 반복한다면 2를 설정한다. PIT 컨트롤러를 초기화한 후에는 타이머가 완료되었음을 인터럽트를 통해 확인한다.

2.2.3 카운터를 읽어 직접 시간 계산

인터럽트를 할 수 없는 상황에서는 카운터를 직접 읽어 시간을 계산한다. PIT 컨트롤러는 카운터가 초기화되면 내부 클록에 따라 카운터의 값을 1씩 감소시킨다. 카운터의 값을 직접 읽어 각 시점의 카운터의 차를 이용해 일정 시간이 경과했는지 여부를 판단한다.

래치 커맨드를 PIT 컨트롤러로 전송해 카운터를 읽는다. PIT 컨트롤러는 래치 커맨드가 지정한 카운터 레지스터에 현재 감소 중인 카운터 값을 저장한다. 커맨드를 전송한 후 카운터의 I/O 포트를 하위 바이트와 상위 바인트 순으로 두번에 걸쳐 읽는다.

카운터를 2바이트 부호 없는 정수형으로 저장해서 카운터의 차를 쉽게 구한다.

2.2.4 타임 스탬프 카운터와 RTC

타임 스탬프 카운터는 프로세서의 클록에 따라 값이 증가하기 때문에 시간을 정확하게 측정할 때 사용한다. MINT 64 OS의 타겟이 되는 멀티코어 프로세서는 매 클록마다 프로세서 최대 클록을 버스 클록으로 나눈만큼 타임 스탬프 카운터가 증가한다. 타임스탬프 카운터는 RDTSC 명령어를 사용하여 상위 32비트를 RDX 레지스터에 넣고 하위 32비트를 RAX 레지스터에 넣는다. RDTSC 명령어는 레지스터를 직접 제어하므로 64비트 어셈블리어 함수로 작성한다. 코드는 길지 않다. rdtsc 명령을 실행하고 각 레지스터를 읽어 들이면 된다.

2.2.5 RTC 컨트롤러와 CMOS 메모리

RTC 컨트롤러는 PC가 꺼져도 현재 일자와 시간을 기록하는데, 전원을 따로 사용한다. 그리고 RTC 컨트롤러는 현재 시간을 BIOS와 공통으로 사용하는 CMOS 메모리에 저장한다. CMOS 메모리에는 0x70, 0x71 포트로 접근한다. CMOS 메모리에는 시스템 전반에 대한 정보가 들어있지만 RTC 컨트롤러에 관한 레지스터 어드레스는 0x0A, 0B, 0C, 0D 이다. MINT64 OS는 현재 시간과 일자만 얻으면 되므로 Seconds, Minutes, Hours, Day, Month, Year 레지스터를 읽는다. RTC 컨트롤러는 BCD 포맷으로 데이터를 저장하므로 BCD 포맷을 바이너리 포맷으로 변환해 읽어야 한다. CMOS 메모리의 RTC 정보는 10진수 두 자리를 한 바이트에 저장한다. 상위 4비트, 하위 4비트 나누어 바이너리 포맷으로 변환하면 된다. 요일은 1~7로 변환한 값으로 저장하는데 숫자는 알아보기 어려우므로 ConvertDayOfWeekTostring()에서 문자열로 변환해 저장한다.

2.2.6 코드 통합과 빌드

위에 준비한 3가지 디바이스를 MINT64 OS에 통합한다. 인터럽트 속도 조절 기능, 프로세서 속도 측정 기능, 현재 일자와 시간 표시 기능을 구현한다. PIT 컨트롤러 파일에는 컨트롤러를 제어하는 함수와 관련 매크로를 정의 한다. 어셈블리어 유틸리티 파일에 READTSC() 함수를 추가한다. 이는 타

임 스탬프 카운터를 읽어서 반환하는 함수이다. RTC 컨트롤러 파일에 CMOS 메모리에서 RTC 컨트롤러가 저장한 현재 일자와 시간을 읽는 함수와 매크로를 정의한다. 콘솔 셸 파일에는 다음의 다섯 가지 커맨드를 추가한다. 먼저 PIT 컨트롤러 초기화 및 일정시간 대기하는 settimer, wait 커맨드이다. 두 번째는 타임 스탬프 카운터를 읽고 CPU 속도를 간접적으로 측정하는 rdtsc와 cpuspeed 커맨드이다. 그리고 현재 일자와 시간을 출력하는 date 커맨드를 추가한다.

코드를 추가하고 빌드하여 결과를 확인한다. help를 입력하면 추가된 커맨드를 확인할 수 있다. 커맨드의 정상 동작을 확인한다.

1. PIT 컨트롤러를 제어하여 10초 동안 대기하였다가 IRQ 인터럽트의 발생 주기를 50ms 간격으로 설정해본다. wait 및 settimer 커맨드를 사용하는데 정상 동작을 확인할 수 있다.
2. rdtsc 명령어를 사용하여 타임 스탬프 값을 확인하고 cpuspeed를 사용하면 측정을하여 프로세스의 클록을 알 수 있다. 그리고 date 명령으로 현재 일자와 시간을 확인가능하다.

구현한 기능을 사용해 디지털 시계나 달력 등의 프로그램을 만들 수 있다.

2.3 태스크 개념을 추가해 멀티테스킹 구현

2.3.1 태스크 제어 블록 정의

태스크 제어 블록, TCB는 태스크의 종합적인 정보가 들어있는 자료구조이다. 코드와 데이터, 컨텍스트와 스택, 태스크를 구분하는 id와 기타 속성 값이 있다. 태스크의 정보를 한데 모아 관리하면 태스크에 관련된 작업을 편리하게 처리할 수 있다. TCB는 태스크와 1:1로 매핑되어야 한다. TCB에는 컨텍스트 변수, id, 플래그, 스택의 어드레스와 크기를 담는다. 컨텍스트에는 프로세서의 상태, 즉 레지스터를 저장한다. 총 24개의 레지스터를 저장한다. 인터럽트나 예외가 발생했을 때와 같은 순서로 컨텍스트 자료구조에 레지스터를 저장한다.

2.3.2 태스크 생성 처리

태스크의 코드, 데이터, 스택을 할당하고 TCP를 생성한다. 태스크의 엔트리 포인트 함수와 관련 코드를 작성하고 데이터를 정의한다. 스택은 8KB 공간을 할당해 지역 변수를 할당한다. 태스크의 코드, 데이터, 스택이 준비되면 TCB를 생성한다. 넘겨받은 정보를 컨텍스트의 정확한 오프셋에 채워 넣으면 된다. ID, 플래그, 엔트리 포인트, 스택 주소와 스택 크기를 파라미터로 넘겨 받아 TCB 를 설정한다. 먼저 RSP, RBP 레지스터를 초기화해 스택을 초기화한다. 세그먼트 셀렉터 CS, DS, ES, FS, GS는 커널에서 사용하는 커널 코드 디스크립터와 커널 데이터 디스크립터를 각각 설정한다. 엔트리 포인트는 ROP 레지스터에 태스크의 시작인 엔트리 포인트의 어드레스를 저장한다. 인터럽트 발생 가능 여부는 인터럽트 플래그를 1로 설정해 인터럽트가 발생하도록 둔다.

2.3.3 태스크 전환 처리

태스크 전환은 크게 태스크를 저장하는 과정과 복원하는 과정으로 나뉜다. 이 때 컨텍스트를 관리하는 것이 중요하다. 컨텍스트는 24개의 레지스터를 저장하면 되는데 SS, RSP, RFLAGS, CS, RIP 레지스터는 예외로 주의해야 한다. SS, RFLAGS, CS 레지스터는 현재 프로세서에 저장된 값을 그대로 저장한다. 하지만 태스크 전환 이후 실행할 코드의 어드레스를 저장하는 RIP 레지스터는 스택읠 참고해서 설정한다. RSP 레지스터는 레지스터에서 복귀할 어드레스 만큼을 제외한 어드레스를 설정한다. 나머지 레지스터는 SAVECONTEXT라는 인터럽트 및 예외 처리에 사용하는 매크로를 사용해 저장한다. SAVECONTEXT 매크로 사용 전에는 RSP 레지스터의 어드레스를 변경해야 한다. 스택의 어드레스는 총 19개라서 컨텍스트 자료구조의 19번째 레지스터 영역으로 설정한다.

2.3.4 태스크 복원 처리

태스크 복원은 저장하는 방법과 비슷하다. RSP 레지스터에 복원한 컨텍스트의 어드레스가 들어있는 RSI 레지스터를 대입하고 LOADCONTEXT 매크로를 제작해 실행하면 19개의 레지스터를 쉽게 복원할 수 있다. 그리고 SS, RSP, RFLAGS, CS, RIP 레지스터는 인터럽트나 예외가 완료되었을 때 사용하는 IRET 명령어를 통해 복원한다. 이 때 태스크 전환 작업이 모두 끝나고 프로세서는 새로운 태스크를 수행한다.

2.3.5 멀티태스킹 기능 통합과 빌드

태스크 파일에 어셈블리어 유틸리티 파일에 추가된 switchContext() 함수 및 태스크 생성에 관련된 함수와 자료구조, 매크로를 정의한다. 어셈블리어 유틸리티 파일에 태스크 전환에 핵심인 switchcontext() 함수를 추가한다. 콘솔 셸 파일에 추가된 기능을 확인하는 createtask 커맨드, TCB, 스택과 테스트용 태스크 코드를 추가한다. createstack 커맨드는 다른 태스크인 testTAsk를 생성하여 태스크 전환을 수행한다. testTask 는 현재 어느 태스크가 실행 중인지 판단하기 위해 각 태스크의 메시지를 출력하고 키보드 입력을 대기했다가 키가 입력되면 다른 태스크로 전환하는 것을 반복한다. 만약 문제가 있다면 정상 출력되지 않고 정지한다.

정상적으로 빌드하고 실행하면, createtask를 수행했을 때 태스크가 정상적으로 수행되면 'This message is from ConsoleShell...' 메시지가 출력된다. 이 때 아무 키나 누르면 testtask()로 태스크가 전환되고 화면에 'This message from testTask...' 메시지가 출력된다. 이는 정상 적으로 전환된 것이고 또 키를 누르면 createtask 메시지가 표시된다. 멀티태스킹이 정상적으로 이루어짐을 확인할 수 있다.

2.4 셸 기능 업그레이드

2.4.1 history 구현

Char command_list[10][300] 배열을 만들어 최대 10개의 명령어를 저장한다. 위/아래 화살표 키를 누르면 가장 최근의 명령어부터 차례대로 출력이 된다. 화살표 키를 눌러 최근 명령어를 출력하게 되면 기존의 명령어랑 겹치지 않게 하기 위해 해당 라인을 초기화하고 다시 프롬프트를 띄운 후 명령어를 출력한다.

2.4.2 명령어 수행시간 측정 기능

Time <Command>로 들어온 것을 kExecuteCommand함수의 인자로 넘겨주어서 명령어 수행시간을 측정한다. 수행하기 전 kReadTSC()함수를 통해 시작 timestamp counter값을 측정하고, 끝난 후 kReadTSC()함수를 통해 마찬가지로 timestamp counter값을 측정 한다. Rdtsc 어셈블리 명령어를 통해 64비트에 저장되므로 startCounter와 endCounter는 QWORD형으로 정의한다. counter값이 계산되면 적절한 값으로 나눠서 수행시간을 출력한다.

```
void kMeasureCommandTime( const char* pcParameterBuffer )
{
    QWORD startCounter=0;
    QWORD endCounter=0;
    WORD counter = 0;
    WORD hour, min, sec,msec,nsec,micsec;
    unsigned long ini, end=0;


    startCounter = kReadTSC();
    kExecuteCommand( pcParameterBuffer );
    endCounter  = kReadTSC();
    counter = (endCounter - startCounter);

    min = counter / 60000;
    sec = counter / 10000;
    msec = (counter % 10000)/10;
    kPrintf("measure command time : %d\n",counter);
    nsec = (counter *100000)% 1000;
    micsec = (counter * 100 )% 1000;
    kPrintf("real %d:%d:%d:%d:%d\n",min,sec,msec,micsec,nsec);
}
```

2.4.3. 현재 시간 및 태스크 수행시간 출력 기능

출력할 위치가 화면의 최댓값을 벗어 나면 스크롤 되어 올라간다. 이때 마지막 2줄은 고정되어 시간을 출력해야 하므로 화면의 최댓값을 23으로 수정해 나머지 2줄은 스크롤하지 않고, cursor도 가지 않게 한다. 먼저 현재 시간 출력 기능을 구현한다. RTC.c 에 작성한 kReadRTCTime()을 사용하면 RTC 컨트롤러가 저장한 현재 시간을 읽을 수 있다. 이를 사용해 다음과 같은 함수를 구현한다.

```
BYTE printCurrentTime(BYTE before){
    BYTE bSecond, bMinute, bHour;
    char time[8] = "00:00:00";
    kReadRTCTime( &bHour, &bMinute, &bSecond );
    if( bSecond == before + 1 || bSecond == 0){
    time[0] = bHour / 10 + 48;
    time[1] = bHour % 10 + 48;
    time[3] = bMinute / 10 + 48;
    time[4] = bMinute % 10 + 48;
    time[6] = bSecond / 10 + 48;
    time[7] = bSecond % 10 + 48;

    kPrintStringXY(72, 24, time);
    }
    return bSecond;
}
```

현재 RTC time을 읽어서 기존의 시간 보다 1초 증가했거나 60초를 넘어가 0이 되면 출력할 현재 시간이 바뀌었으므로 알맞은 좌표에 시간을 다시금 출력하는 함수이다. 현재 멀티태스킹이 구현되어 있으므로 멀티태스킹으로 이 기능을 사용하면 좋겠지만 아직 스케쥴러가 없어 계속해서 현재 시간 출력을 멀티태스킹하기는 힘들다. 그래서 이번에는 키큐에 데이터가 수신되는 것을 기다릴 때와 그리고 명령어 안에서 반복문이 실행 될 때 위의 함수를 실행하여 현재 시간을 계속 확인하여 출력하는 식으로 구현한다.

Task의 수행시간은 엔터키를 처리할때 시

작한다. 먼저 kInitializePIT()를 통해 초기화를 한다. 그리고 명령어를 수행하기 전 kReadCounter0()를 통해 시작 카운터 값을 저장하고 수행한 후의 카운터도 저장하여 감소된 시간만큼을 출력한다.

개발 시 경험한 문제점 및 해결점

1. 현재 시간 출력 시 경험한 문제점

Context switch 기능은 구현하여 멀티태스킹을 할 수있다. 하지만 스케쥴러를 아직 구현하지 않아, 계속해서 사용자와 상호작용하는 셸과 현재 시간 출력, 그리고 커맨드의 내부 실행 등의 기능을 병렬적으로 수행하기가 힘들다. 이를 위해서는 많은 부분에 있어 수정이 불가피하기 때문에, 이를 우회적으로 해결하고자 했다. 대부분의 런타임을 가져가는 반복 문에 현재 시간을 출력하는 함수를 삽입하여 해결했다.

2. History 출력시 경험한 문제점

계속 위 방향키를 누르면 가장 나중 명령어가 계속 출력되어야 하고 계속 아래 방향키를 누르면 최종적으로 아무것도 출력이 안되야 했다. 리눅스 환경이랑 최대한 같도록 구현하려고 하다보니 이런 부분을 많이 신경쓰게 되었다.

3 조원별 기여도

신정은: 33%

조재호: 34%

조한주: 33%