20163340 강원경

1.

(a) 소스코드는 전처리(preprocessing), 컴파일링(compiling), 어셈블링(assembling), 링킹(linking) 과정을 거쳐 컴퓨터에서 실행 가능한 기계어 코드로 변환한다. 그중에서 문제에서 제시한 Assembly Code까지의 과정은 컴파일러를 거치게 되는 컴파일링 단계이다. 여기까지의 과정을 설명해보도록 하겠다.

우선 전처리 과정에서 전처리기를 통해 사용자가 소스코드에 지시한 지시자(라이브러리 호출, 매크로 함수 정의 등)들을 처리한다. 그 후 컴파일 과정으로 넘어가게 된다. 이 단계에서 컴파일러는 소스코드를 어셈블리 코드로 변환하는 과정을 거치는데, 시스템 환경에 맞는 어셈블리 코드로 변환한다.

시스템에 따라 어셈블리 코드가 다르게 출력되겠지만, 한가지 예시를 통해 a=b+c를 변환해보도록 하겠다. 우선 변수 a,b,c를 메모리에 할당하는 코드가 필요할 것이다. 그 후 b+c의 연산이 필요한데, ADD와 같은 어셈블리 명령어 코드를 통해 b+c의 연산을 수행할 것이다. 그 다음 이 연산 결과를 a라는 변수의 공간에 저장해야 하므로 시스템에 따라 다르겠지만, MOV 또는 STORE라는 어셈블리 명령어를 통해 변수 a에 값이 저장될 것이다. 이렇게 하면 a=b+c라는 소스코드가 어셈블리 코드로 변환된다.

(b) 일반적으로 제어문 코드는 반복문, 조건문을 통틀어 일컫는 말이다. C언어의 구문을 예시로 하면 if, switch, for, while, break 등이 해당되는 것이다. 이 구문의 공통점은 조건에 따라 다음 코드의 실행 위치가 달라진다는 점이다. 따라서 다음 실행할 코드의 라인이 현재 실행중인 코드의 다음 줄이 될 수도 있고, 아니면 소스코드의 특정 라인으로 건너뛸 수도 있다.

이런 특성을 고려할 때, Assembly Code에서 꼭 필요한 구문은 조건을 판별할 수 있는 코드와 실행할 다음 코드의 위치를 지정하는 명령어가 필요하다. SIC 프로그래밍에서의 어셈블리 코드를 예시로 하면, 별도의 조건 없이 다음 위치를 점프하는 J(Jump), 비교 연산을 통해 조건에 맞으면 점프를 수행하는 JEQ, JLT, JGT와 같은 명령어 등이 필요할 것이다.

2.

(a) Shell이란 대부분의 운영체제에서 지원하는 고유의 소프트웨어로, 사용자의 명령어를 해석하고 운영체제(시스템)에서 이해할 수 있도록 해석하여 지시하는 프로그램을 일컫는 말이다. 다시 말해, 사용자와 커널을 이어주는 중간다리의 역할을 수행하는 것이다.

일반적으로 운영체제(또는 컴퓨터 시스템)는 사용자가 내리는 문자 형태의 명령어를 바로 이해할 수 없다. 그래서 사용자가 어떤 명령을 내리는지 이해할 수 있도록 기계어로 된 명령어로 해석할 필요가 있는데, 이 역할을 수행하는 것이 Shell이다.

간혹 Shell과 터미널/콘솔을 혼동하는 경우가 있는데, 둘은 엄밀히 따지면 다른 프로그램이고 각각은 다음과 같이 구분할 수 있다.

- Shell : 사용자로부터 문자로 된 명령을 통해 컴퓨터에 명령할 수 있도록 해석하고 Kernel에 전달하는 프로그램
- 터미널/콘솔 : Shell을 실행하기 위해 문자로 된 명령어를 입력받아 그 입력과 출력을 화면상에 보여주는 프로그램

(b) fork, exec, pipe는 프로세스 간 상호작용을 하는 점에서는 공통인 특징을 가지는 함수들이다. 하지만, 각각의 함수는 세부 동작이 서로 다른데 이것을 정리하면 아래와 같다.

- fork() : 하나의 프로세스에서 그 프로세스가 필요로 하는 부가적인 프로세스를 생성할 때 사용하는 함수로, 이 함수를 호출한 프로세스는 부모 프로세스, 호출되어 생성된 프로세스는 자식 프로세스로 취급한다. 이 때, 메모리에는 자식 프로세스를 실행하기 위한 별도의 공간이 할당되며, 부모 프로세스와 자식 프로세스는 별개의 프로세스로 취급된다.
- exec() : 하나의 프로세스에서 또 다른 부가적인 프로세스를 생성할 때 사용한다는 점은 fork() 함수와 같지만, 새로운 프로세스가 기존 프로세스 위에 덮어 씌워진다는 점이 가장 큰 차이점이다. 즉, exec()를 호출한 프로세스의 PID가 곧 새로운 프로세스의 PID가 되고, 기존 프로세스의 메모리 영역을 그대로 새로운 프로세스가 이어받아 사용하게 된다. 이러한 특징을 이용해 fork 함수를 통해 만들어진 자식 프로세스에서 부모 프로세스와 다른 프로세스를 실행할 때 사용하는 용도로 사용되기도 한다.
- pipe() : 서로 독립된 프로세스들이 데이터를 주고받을 때 사용하는 함수이다. 하나의 프로세스의 출력 결과가 다음 프로세스의 입력 인자로 사용되는 등 하나의 파이프를 통해 여러 프로세스 간의 데이터 이동 및 상호작용이 필요할 때 사용한다.

3.

(a) JAVA는 Exception Class에 시스템에서 정의한 예외들이 존재한다. 이러한 예외들을 처리하는 방법은 두 가지가 있는데, Java Virtual Machine (JVM)에 정의된 기본 예외 처리 방법을 사용하는 방법이 있고, 사용자가 직접 예외를 정의하고 어떻게 처리할 것인지 정의하여 처리하는 방법이 있다.

프로그램 작성 시 try ... catch 문을 이용해 예외가 발생하는지 여부를 판단하고 처리하는데, try 부분에서는 예외가 발생할 것으로 예상되는 코드를 작성하고, catch 문에서 검출된 예외를 처리한다.

또한, throw 문을 사용하여 JVM에서 정의되지 않은 사용자 지정 예외를 정의할 수 있다.

(b) Unix나 Linux는 JAVA와 달리 사용자 정의 예외처리를 수행할 수 없다. 즉, 시스템에 정의된 예외만 처리할 수 있다.

OS에서 실행하고 있는 프로세스에서 Interrupt(또는 예외)가 발생한 경우 그 프로세스에 Signal을 보내고 Kernel의 역할은 종료된다. Signal이 수신된 경우 그 프로세스 PCB에 해당 Signal 값이 1로 변하게 된다. 그 다음, OS에서 Interrupt(또는 예외)가 발생한 프로세스의 PCB 중 Signal Table을 참조하여 1값이 있는 경우, 그 1값에 해당하는 예외에 대한 처리를 시스템에 정의된 방식으로 처리한다.

위 단락에서 설명한 내용이 가장 일반적인 Unix/Linux 시스템에서의 예외 처리 방식이다. 만약 시스템에 정의된 기본적인 처리 방식(대부분은 '덤프' 또는 '종료'다.)이 아닌 다른 방법으로 처리하고자 할 때는 그 예외에 해당하는 Signal에 대한 처리 지침을 Sigaction 함수를 통해 바꾸면 된다. 이 함수의 첫 번째 인자로 Signal의 번호를 주고, 두 번째 인자로 0, 1, 또는 임의의 메모리 주소값을 준다. 두 번째 인자에서 0 일 경우 시스템에서 정의한 Default action으로 처리하고, 1인 경우는 Ignore(무시), 그리고 임의의 메모리 주소값인 경우 그 메모리 주소에 있는 시그널 처리 함수를 실행시킨다.