11\_11\_시스템아키텍처

141019 김연우

1. 어셈블리 코드의 개념
   1. 어셈블리란?
      1. ISA(Instruction set achitechure)를 사람이 알기 쉬운 형태로 전달하는 것
      2. 기계어 & 어셈블리
         1. 기계어: 컴퓨터가 수행할 작업을 인코딩한 연속된 바이트 파일
         2. 어셈블리: 기계어 코드를 사람이 읽기 쉬운 형태로 표시한 것
            1. 고급언어보다 하드웨어의 동작을 구체적으로 유도할 수 있다.
            2. 하지만 짜기 어렵다. 요즘 컴파일러 >>>>>> 나
            3. 근데 왜 공부?

하드웨어에 최적화된 프로그래밍

버그의 근본적 원인을 찾기 위해. esp) 동기화

* 1. 인텔 프로세서
     1. x86 (ISA 명령어 셋 아키텍처의 이름, IA32랑 거의 같다고 생각하자)
     2. 8086(1978년 프로세서)부터 호환성을 유지하면서 확장했음
     3. CISC 구조(복잡한 명령어 셋, (cf) RISC: 단순한 명령어 셋))의 대표주자.
     4. 클럭 구조를 사용하여 최적화
     5. 컴퓨터 아키텍처의 발전과정은 명령어 셋 아키텍처와 밀접한 연관이 있다.
        1. 아키텍처는 설계도 : ISA
        2. 마이크로 아키텍처가 구체적인 하드웨어의 구성을 말함.
  2. C vs 어셈블리
     1. CPU란 결국 상태기계
        1. 처리한 결과 레지스터에 저장
        2. 명령어 하나를 처리하고 PC값에 따라 다음 명령 수행
        3. 현재 상태에 따라 조건 레지스터가 다음 상태를 결정(?)
     2. c코드를 컴파일 하면 어셈블리 코드로 변경된다. (gcc -01 –S cosde.c)
     3. 어셈블리 코드가 어셈블러를 거쳐 바이너리 코드(기계어)로 변경된다.
        1. CISC는 기계어의 명령 단위가 서로 다르다. (RISC는 똑같음)
        2. 다른 코드와의 연결정보(LINK)는 누락된 상태.
        3. 기계어 표현은 사용이 제한이 있다. (2항연산만 가능…)
     4. 기계어로부터 역 어셈블이 가능하다.(objdump –d prog)
        1. 목적 코드를 분석가능 (reverse engineering)
        2. 링크하기 전까지 함수나, 변수의 주소는 결정되지 않는다.
        3. 디버거를 사용해서 역어셈블도 가능
           1. 리눅스 디버거 : gdb dissamsemble….
           2. 비주얼 스튜디오에서 가능 : break걸고 디버그 창\디스어셈블리

어셈블리 코드 형식이 다름 AT&T(리눅스) vs 인텔(윈도우)

* + 1. 링커는 다른 목적파일들을 연결하고 변수 ,함수등의 데이터들의 실제 주소를 결정한다.
  1. 어셈블리 기본
     1. 정수 1, 2, 4, 바이트, float 2, 4, 8(?)
     2. WORD 2바이트 ( 16비트 호환성), DWORD…QWORD
     3. ISA에는 메모리 단위별로 명령어세트를 따로 가지고 있다.
        1. Ex) add(l) : 32 , add(b) : 8 , add(w) : 16, add(q) : 64
     4. 명령어 종류
        1. 데이터 이동 : 메모리 로드 세이브 카피
        2. 산술 논리 연산 : 메모리(레지스터)에 있는 데이터 조작
        3. 제어 이동 : 조건부 분기, 함수 호출

1. 어셈블리 코드의 형식
   1. IA32 정수 레지스터
      1. 레지스터 사이즈는 컴퓨터 비트 사이즈 32-64 bits
      2. 16bit짜리 가상 레지스터를 쓸 수 있다. (high low로 나눠서 사용하기도)
      3. 쓸 수 있는 저장용 레지스터 8개
      4. 범용 레지스터 6개 변수 담는 용도
         1. %eax (accumulate), %ecx (counter), %edx(data), %ebx(base : 변수에 해당하는 주소를 저장), %esi (source index) , %edi (dst index) 는 훼이크고 그냥 막 쓴다. 하지만 가능하면 용도에 맞게 쓸려고 한다.
      5. ??? : %esp 스택 포인터(?), %ebp 베이스 포인터(?)
   2. 명령어 형식
      1. Operator + Operand
      2. Operand 형식
         1. 레지스터 : %eax
         2. 상수 정수 데이터 : $0x400 (상수는 앞에 $를 쓴다.)
         3. 메모리 : (%eax) 괄호 붙이거나 그냥 숫자 ($이 없다\*)
      3. 문법
         1. OP + Src + Dst : movl %eax ,%edx
         2. 문법이 정해져 있음. (RISC는 간결(일반 연산은 모두 레지스터, 메모리 이동할 때에만 메모리 사용), SISC는 더러움)
            1. Ex) Move 는 src + dst 이때 src가 메모리이면 dst는 레지스터만 가능
   3. 메모리 주소지정 모드
      1. 일반 지정 : Movl (%ecx), %eax;
      2. 변위 지정 : Movl 8(%ebp), %edx 상수 D는 참조할 변수의 변위를 표시
      3. Generic 변위지정 : 배열이나 구조체 클래스의 단위로 접근하기 위한 메모리 표현방법
         1. D(Rb, Ri, S) => movl 0x2048(%ebp, %edi , 4)
         2. Rb(head의 위치) + (Ri \* S(한 단위의 크기: 1,2,4,8 only)) + D(상수 변위)
         3. D에 큰 단위의 위치 값을 넣고 Rb에 그 내부의 단위에서 Base를 지정해주는 방식을 사용할 수 있다.
2. 데이터 이동 명령어
   1. MOV 일반 이동, MOVS 부호비트 고려 확장 MOVZ 그냥 0으로 채우는 확장
   2. PUSH ,POP ??? 스택!
      1. 스택 : 함수의 호출을 지원하고 지역변수를 저장하는 영역, CPU가 지원함
         1. %esp에 스택의 top주소를 유지함.
            1. 함부로 %esp에 값을 넣으면 Stack영역 터짐
         2. PUSH %eax == SUB $4 %esp MOV %eax, (%esp)
         3. 자주 쓰이므로 한번의 연산으로 해결
   3. %ebp ?? Base Pointer , Stack Frame
      1. 함수가 사용하는 영역을 표시하는 것
      2. Esp는 전체 프로그램의 스택의 top
      3. Ebp는 현재 진행되는 함수의 진행 위치 등을 표현
3. 산술 연산 명령어
   1. 함수가 호출되면 스택의 top에 인자, 리턴 주소가 자동으로 쌓인다.
   2. 일반적으로 시작부분에 이전 진행상황을 저장해서 이번 호출된 함수가 종료된 이후에 팝해서 다시 원래의 진행상황으로 돌아가게 한다.