60200307 우연정

목차

1. Memory Allocation
    1-1. PCB
2. CPU Context

1. Memory Allocation

프로그램이 실행된다는 것은 메모리에 프로그램이 장착된다는 것이다. 메모리에 올라가서 cpu가 내 코드 세그먼트의 명령어를 실행하고 있다는 것이다. 메모리에 프로세스 하나가 올라가 있다.
프로세스 제일 위에는 프로세스 컨트롤 블록이 있다. PCB 그리고 코드 세그먼트가 있고, 데이터 세그먼트. 힙 세그먼트, 스택 세그먼트가 있다.
힙 메모리는 NEW 명령어를 통해서 만들어지는 메모리이다. 이것은 프로그램이 실행되어야 생기는 동적 할당 메모리이다. 스택은 언제 메모리가 할당되냐면, 함수가 호출될 때 할당된다. cpu는 코드 세그먼트에 있는 cpu가 실행할 수 있는 명령어를 실행한다.
만약 a와 b와 c를 더하라는 명령어가 있다면, 먼저 a와 b를 잠깐 어딘가에 저장해놓아야 한다. 이런 것들을 저장할 공간이 필요해서 메모리가 데이터 세그먼트, 힙 세그먼트, 스택세그먼트로 3가지의 논리적인 세그먼트를 만드는 것이다. cpu가 저장을 하는 공책으로 비유할 수 있다. 코드 세그먼트가 뭔가를 실행할 때 그 중간 중간에 무언가를 저장해놓고 그것을 나중에 쓰는 등의 여러 가지를 하는데, 이렇게 저장하는 용도로 쓰이는 세 종류의 메모리가 있는 것이다.

메모리는 생성되는 형태에 따라 달라진다. 즉 어떤 명령어에서 할당을 하느냐에 따라 달라진다. 데이터 세그먼트는 static 객체를 할당할 때 쓴다. 힙과 스택 세그먼트는 프로그램이 실행되는 순간에 메모리가 생성되고 소멸되면서 동적으로 생성된다.
그러나 데이터 세그먼트는 정적이다. 정적이라는 것은 프로그램이 시작될 때 만들어져서 프로그램이 사라질 때까지 고정되어있다. 데이터 세그먼트는 명령어로 만들어지는 것이 아니라 움직이지 않고 고정적인 메모리이다.
코드 세그먼트도 고정적이다. 프로그램이 실행될 때 메모리에 올라와서 있다가 끝날 때 내려간다. 코드 세그먼트는 함수가 여러 개 호출한다고 해서 여러 개의 코드가 만들어지는 것이 아니라 코드는 한 세트만 있다. 코드는 원래 고정되어있다.

1-1. PCB

프로세스는 메모리에 여러 개가 있고, cpu도 하나만 실행하는 것이 아니다. 타임 쉐어링을 하는데, 시간을 할당해서 계속 도는 것이다. 그 중 에서도 현재 액티브한 프로그램을 많이 실행해준다.
cpu가 5번째 라인을 실행하고 있었는데, OS가 그것을 중단하고 다음 것을 하라고 한 뒤, 돌아왔을 때는 5번째 라인을 하고 있었다는 것(어디까지 실행했는지)과 그 당시 cpu 안에 있는 상태가 어떠한지를 자세하게 기억해야 한다. (Context) 이렇게 하기 위해서는 cpu 안에 있는 메모리를 그대로 복구해줘야 한다. cpu가 다른 곳으로 넘어가려 할 때 미리 cpu의 모든 메모리(Context)를 PCB에 집어넣어 놓는다. 이것을 cpu의 context switching(cpu가 다른 프로세스를 실행하는 것, cpu의 문맥이 바뀌는 것)이라고 한다. 즉 PCB에는 프로세스 기본정보, 상태(cpu context) 이 두 가지를 저장해놓는다.

context는 독립된 의미적 연결성을 가진 단위를 말한다. 프로세스의 문맥은 프로세스 안에

있고 cpu의 문맥은 cpu 내부의 메모리에 있다. 데이터, 힙, 스택 세그먼트를 모두 카피해놓은 것이 프로세스의 상태가 된다. 코드는 상태를 만들어내지 않는다. 상태는 어떤 데이터를 가지고 있느냐에 따라서 어디까지 했느냐가 결정되는 것이다.
cpu에서는 어디에서 문맥이 바뀌냐면, cpu 내부에는 레지스터라는 아주 빠른 메모리가 있다. 우리가 가진 메모리는 cpu가 쓰기엔 너무 느려서 메모리에서 cpu 메모리로 값을 가져온다. 결과를 계산하고 그 값을 메모리에게 다시 준다. 폰노이만 구조는 명령어와 데이터가 복잡한 경우에 쓰는 것이다.

만약 y=a+b라는 계산을 하기 위해서는 cpu가 a와 b를 읽어와서 결과를 계산한 후 그 결과를 y에다가 집어넣어야 한다.

static 함수 안에 있는 변수는 static이다. static 함수는 호출해도 스택에 메모리가 생기지 않는다. main에 있는 것은 전부 static 함수이다.
view는 0번지, x는 4번지로 계산한다. (컴파일러는 항상 상대주소로 계산한다.)
함수 run이 호출되면 실제로 스택 세그먼트에 8바이트가 만들어진다. 함수가 끝나면 스택에서 없어진다.

new를 하면 힙에 메모리가 CMain만큼 (8바이트)이 만들어진다.
첫 번째 단계에서 계산할 때 데이터에 관련된 내용은 필요없어졌다.
main의 a는 0번지, b는 4번지라고 기억을 해놓는다. (심볼 테이블, 소스코드를 한 번 읽어서)

컴파일을 하면 심볼 테이블을 만든다. 심볼 테이블은 함수의 메모리 사이즈와 위치, 주소를 담고 있다.

두 번째 단계에서는 진짜 실행할 코드만 남겨져 있다.
x는 run 함수 안에 있다. 이는 스택 세그먼트의 0번지(함수의 시작점)+4번지에, a는 힙 세그먼트의 메인 객체의 0번지, b는 힙 세그먼트의 메인 객체의 4번지라고 주소를 만든다.
main은 main 함수의 4번지이다. 스택 세그먼트의 main함수의 시작점의 4번지였으나 main은 static 함수이므로 데이터 세그먼트의 4번지이다. main = new CMain();은 '데이터 세그먼트의 4번지에 new 8바이트 해라'라는 기계어로 바뀐다. 'new 8바이트'라는 명령어는 힙 세그먼트한테 8바이트를 할당해서 주소를 알려달라는 명령어인 것이다. 그리고 그 주소가 데이터 세그먼트의 4번지로 들어간 것이다. 이것이 가능한 이유는 심볼 테이블을 만들어 놓았기 때문이다.

main.run()을 하면 코드 세그먼트의 0번지로 점프한다. 우리가 정의한 것은 컴파일러가 변환하지 않고 참조할 정보일 뿐이다. 참조 정보를 참고해서 이름과 정보가 담긴 심볼 테이블을 미리 만들어 놓고 그 테이블을 가지고 명령어들을 기계어로 바꾸는 것을 컴파일이라고 한다.

public class CMain {
  static private int s;
  private a, b;
  private void run() {
  View view;
  int x;
  x = a+ b;
  view = new View();
  view.showUserInfo();
  }
 	
public static void main(String[] args) {
  int x;
  CMain main;
  main = new CMain();
  main.run();
  }
}
	첫 번째 단계
	-> CMain
	4,4 a,b 4바이트
	
	-> run함수
	4 포인터 4바이트
	4 함수 4바이트
	
	->main 함수
	4 integer x 4바이트
	4 main 4바이트
		두 번째 단계
		public class CMain {
		  private void run() {
		  x = a+ b;
		  view = new View();
		  view.showUserInfo();
		  }
		
		public static void
		  main(String[] args) {
		  main = new CMain();
		  main.run();
		  }
		}

2. CPU Context

PC는 프로그램 카운터이다. 프로그램 카운터는 cpu의 레지스터 안에 있고, 코드 세그먼트(CS)가 몇 번째 줄을 실행하고 있었느냐를 알고 있다. 세그먼트 레지스터에는 코드, 데이터, 힙, 스택 세그먼트 레지스터가 있다. 각각의 레지스터는 프로세스의 코드, 데이터, 힙, 스택 세그먼트를 가리키고 있다.

세그먼트 레지스터들은 PCB에 있다. 여기에는 고정되어있는 데이터, 코드 세그먼트가 저장되어있다.

프로그램 카운터는 현재 코드에서 cpu가 어디를 실행하고 있는 지를 가지고 있다. 세그먼트 레지스터에는 현재의 주소들이 들어가 있고, General Purpose 레지스터에는 계산의 중간 상태가 들어가 있다. 세그먼트 레지스터는 주소를 계산할 때 모든 것을 상대적으로 계산하기 위한 일반적인 기준점일 뿐이다. 내 context 내에서 모든 것을 0번지로 생각하고 계산하기 위함이다.

cpu가 다른 프로세스를 실행할 때 cpu의 context를 다른 프로세스의 context로 바꾼다. 여러 개의 프로세스가 동시에 타임 쉐어링을 한다. 여러 개의 프로세스를 계속 오가며 계속 실행한다.