**-OS를 쓰는 이유는??????????????????**

1. 운영체제를 쓰는 이유? 컴퓨터를 사용하기 위해서
2. 커널은 하드웨어를 제어하기 위한 소프트 드라이버 집합
3. 커널 위에 운영체제가 올라간다
4. 운영체제에서 사용자 입력을 받아서 커널 드라이버를 불러 하드웨어를 관리한다
5. 운영체제 모드는 2가지 있다, user mode랑 커널모드
6. 세탁기나 이런건 임베디드 OS ex) RTOS 라는 OS가 들어간다.
7. 기계를 소프트웨어적으로 조종하기 위해서 OS가 필요

하드 디스크와 보조기억장치, 레지스터, 부팅

1. 하드디스크나 SSD와 같은 보조기억장치(ROM)에 OS가 저장되어있다.
2. CPU랑 직접 통신하는 게 주기억장치, 이 메모리로 보조기억장치에서 필요한 것을 가져온다.
3. 가상 메모리라고 해서 하드 디스크나 SSD와 같은 보조기억장치를 일부 가상메모리로 사용된다. 대신 굉장히 느려진다.
4. 레지스터는 CPU보다는 느리지만 메모리에 비해 훨씬 빠르기 때문에 중간 역할을 한다.

Ex) 인덱스 레지스터는 주소 지정, ,.

1. 사용 목적에 따라 레지스터는 8비트 16비트 32비트 등의 크기를 가진다.
2. 운영체제의 일부인 커널이 메모리에 올라와 실행되어 장치들을 준비시키고 각종 레지스트 값을 초기화하고 나서 사용자 입력 받을 준비가 되어있으면 이것을 ‘부팅되었다’라고한다.
3. => 이 때 사용되는 것은? Bootstrap Loader이고 보조기억장지치인 ROM에 저장되어있다.
4. PC 카운터: [마이크로프로세서](https://ko.wikipedia.org/wiki/%EB%A7%88%EC%9D%B4%ED%81%AC%EB%A1%9C%ED%94%84%EB%A1%9C%EC%84%B8%EC%84%9C)([중앙 처리 장치](https://ko.wikipedia.org/wiki/%EC%A4%91%EC%95%99_%EC%B2%98%EB%A6%AC_%EC%9E%A5%EC%B9%98)) 내부에 있는 [레지스터](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EC%84%B8%EC%84%9C_%EB%A0%88%EC%A7%80%EC%8A%A4%ED%84%B0) 중의 하나로서, 다음에 실행될 명령어의 주소를 가지고 있어 실행할 기계어 코드의 위치를 지정한다. 때문에 명령어 포인터 (인텔의 x86계열의 CPU에서는 IP라고 한다.)

IR(Instruction Register): 현재 실행 중인 명령을 기억하는 레지스터

**CPU의 구성 요소**

CPU, 레지스터, CU, Bus, ALU

**명렁어 처리 순서**

1. 명령어 fetch
2. 명령어 decode
3. 유효한 주소 읽기 ( 간접 메모리 명령, 직접 메모리 명령 )

* 간접 메모리만 유효 주소를 메모리로부터 읽어들인다.

1. 명령 실행

**인터럽트**

운영체제가 자원을 효율적으로 관리할 수 있는 방법->컴퓨터 시스템에 존재하는 각 자원들의 현 상황을 파악 수 있는 방법

1. Polling : CPU가 일정한 시간 간격을 두고 각 자원들의 상태를 주기적으로 확인하는 방식.

자신의 상태를 적어놓을 수 있는 곳에 저장시키고 CPU는 폴링 때 그것을 읽어보는 방식

Polling의 한계 : 폴링의 간격을 적절히 정해야 하는 숙제와 동시에 각 자원들은 직전 폴링 이후 변화된 자신의 상태를 다음번 폴링 때가지는 알릴 수 없다는 점, 또한 아무 일이 없었는데도 CPU는 폴링에 일정량의 시간을 들여야한다.

1. 인터럽트 : 각 자원들이 능동적으로 자신의 상태변화를 CPU에게 알리는 방식. CPU는 따로 시간을 들이지 않아도 되고, 자원들은 상황이 발생하 면 즉시 알려 처리 받을 수 있으니 폴링에 비해 훌륭한 방식
2. 실행 중인 CPU가 스스로 자신에게 인터럽트를 해야 할 때가 있는데 이것은 실행중인 명령어 때문에 생기는 일이니까 소프트웨어 인터럽트라고 하고 트랩이라고 부른다.

**트랩과 인터럽트의 차이점**

트랩은 프로그램 내에서 발생하는 것이고 내부 인터럽트라고 하며 CPU로부터 발생하는 운영 오류 등 포함됩니다. 인터럽트는 하드웨어적으로 프로그램 외부에서 발생하는 것이며 트랩은 발생하는 시점이 프로그램의 일정함 지점이라는 점에서 동기적. 즉 고정된 영역에서 일어난다.

반면 인터럽트는 프로그램 외부 상황에 따라서 발생 시점이 일정하지 않기 때문에 비동기적입니다.

프로그램 실행중 overflow나 underflow 같은 특이한 사항(어던 조치를 취해야 할) 이 발생한 경우 프로그램은 지정한 특정번지로 점프하여 그곳의 명령을 실행하게 되는데, 이렇게 프로그램이 자동적으로 특정 번지로 점프하여 그곳의 명령을 실행하는 것을 트랩이라고 하며, 특정 번지의 있는 프로그램을 트랩 핸들러라고 합니다.

간단하게 말하자면 인터럽트는 하드웨어적인 흐름의 변화이고, 트랩은 소프트웨어적이라고 할 수 있다. 트랩과 인터럽트의 주된 차이점은 트랩은 어떤 프로그램을 실행시켰을 때, 항상 고정된 위치에서 일어나나 ,외부 인터럽트는 언제 발생할지 모른다는 점입니다.

인터럽트는 장치 폴링 문제 해결을 위해 입출력장치의 완료 신호로 사용할 수 있으며, 트랩은 운영체제 루틴을 호출하거나, 산술 오류를 잡아내는데 사용할 수 있습니다.

**인트럽트 처리되는 시점**

하드웨어 인트럽트 : 현재 진행 중인 명령어 실행을 마친 후 처리

트랩 : 처리 중인 명령어에 의해 발생하므로 오류의 경우는 바로 프로그램의 종료를 가져온다

시스템 콜 : 입출력과 같은 시스템 콜은 입출력이 완료되어야 실행 중인 명령어가 완료되고 이어서 다음 실행문으로 진행

**인트럽트 처리 방법**

1. 장치가 인터럽트 신호를 CPU에게 보내고 이때 CPU는 명령어를 싱행 중이었다면 먼저 이 명령어의 실행을 완료시키고 인터럽트 신호를 확인한다. 인터럽트 처리하는 루틴을 싱행시켜야 하므로 그전에 현재 실행 중이던 프로그램이 인터럽트 처리 후 다시 실행될 때를 위해 현 상태의 정보를 시스템 스택에 저장. 이때 정보란 PSW(Program status word : 시스템 내부의 순간순간의 상태를 기록하고 있는 정보), PC 레지스터의 값 등.
2. 다음으로 인트럽트 처리 루틴의 시작 주소를 PC에 넣어 실행시킴으로써 결과적으로 인터럽트 처리 루틴이 실행되게 된다.
3. 인터럽트 처리 루틴은 먼저 CPU에 있는 레지스터 값들을 저장한 후(처리 루틴이 실행되면서 이 값들이 훼손될 가능성이 있으므로) 필요한 인터럽트 처리를 시작.
4. 인트럽트의 처리가 끝나면 이전에 저장하였던 레지스터 값들을 다시 restore 한 후 psw와 pc 값들을 원래 자리에 넣어주고 실행하게 되면 프로그램 카운터에 들어가 있는 값이 인트럽트 이전에 실행 중이던 프로그램에서 다음에 실행할 명령어 위치이므로 자연스럽게 프로그램의 실행을 이어간다.

**메모리 구조**

프로그램을 실행시키면 운영체제는 우리가 실행시킨 프로그램을 위해 메모리 공간을 할당해준다. 할당되는 메모리 공간은 크게 스택, 데이터 영역., 힙영역, 코드여역

프로그램이 실행될 때마다 할당되고, 할당 장소는 메인 메모리(RAM) 영역이다.

데이터 영역은 전역 변수, static 변수

힙 영영역은 런 타임에 크기가 결정되고 사용자의 동적 할당 담당, 스택은 지역변수와 매개변수 컴파일 타임에 크기가 결정된다.

스택은 높은 곳에서 낮은 곳으로 주소 할당, 힙 영역은 낮은 주소에서 높은 주소의 방향으로 할당

Context Switching

왜 Context Switching이 필요할까?

1. Computer가 매번 하나의 Task만 처리할 수 있다면? 해당 task가 끝날 때까지 다음 Task는 기다릴 수 밖에 없습니다.(starvation), 또한 반응속도가 매우 느리고 사용하기 불편…

그래서 다양한 사람들이 동시에 사용하는 것처럼 하기 위해선

1. Computer multitasking을 통해 빠른 반응속도로 응답할 수 있다, 빠른 속도로 Task를 바꿔 가며 실행하기 때문에 사람의 눈으론 실시간처럼 보이게 되는 장점이 있다.

CPU가 Task를 바꿔가며 실행하기 위해 Context Switching이 필요하게 되었다.

* Context Switching이란 현재 진행하고 있는 Task(Process, Thread)의 상태를 저장하고 다음 진행할 Task의 상태 값을 읽어 적용하는 과정

1. 어떻게 Context switching은 진행되는 것일까?

Task의 대부분 정보는 레지스터에 저장되고 PCB(Process Control Block)로 관리되고 있다.

현재 실행하고 있는 Task의 PCB 정보를 저장한다.(Process Stack, Ready Queue).

다음에 실행할 task의 PCB 정보를 읽어 레지스터에 적재하고 CPU가 이전에 진행했던 과정을 연속적으로 수행을 할 수 있습니다.

1. PCB의 구조 중 중요한 것

Process State: 프로세스 상태 : Create, Ready, Running, Waiting, Terminated

Process Counter: 다음 실행할 명령어의 주소값

CPU Registers : accumulator, index register, stack pointers, general purpose regitsters

1. .Context Switching 을 하기 위한 Cost
2. Cache의 초기화
3. Memory Mapping의 초기화
4. 메모리 접근을 위해서 커널은 항상 실행되어야 합니다.
5. Process와 Thread의 Context Switiching

Context Switching 비용은 Process가 Thread보다 많이 듭니다. 그 이유는 Thread는 stack 영역을 제외한 모든 메모리를 공유하기 때문에 Context Switching 발생시 Stack 영역만 변경을 진행하면 된다.

프로세스 상태와 변화

1. 생성 상태(Created), 준비 상태(Ready), 실행 상태(Running), 대기 상태(Blocked), 보류 준비 상태(Suspended Ready), 보류 대기 상태(Suspended Blocked), 종료(Terminated) 상태가 있다.
2. 생성 상태는 사용자가 요청한 작업이 커널에 등록되고 PCB가 만들어져 프로세스가 만들어진 다음 준비나 보류 준비 상태로 되기 위해 잠시 거치는 상태이다.
3. 운영체제는 프로세스를 생성한 후 메모리 공간을 검사하여 충분한 공간이 있으면 메모리를 할당하면서 준비 상태로 바꾸어주고, 그렇지 못할 경우 보류 준비 상태로 만든다.
4. 준비 상태는 CPU를 할당받기 위해 기다리고 있는 상태 즉, CPU만 주어지면 바로 실행할 준비가 되어 있는 상태이다. 다중 프로그래밍 시스템의 경우 준비 상태의 여러 프로세스들은 메모리에 적대되어 있으며 CPU를 할당 받기를 기다리고 있는데 이들을 위해 큐(혹은 리스트)가 사용된다. 준비 상태의 프로세스들은 순서에 따라 CPU를 할당 받으면 실행 상태가 되는데 이때 순서를 정하는 것을 CPU 스케줄링이라고 부른다.
5. 실행 상태는 CPU를 할당 받아 실행 중인 상태를 말하는데 이때 CPU를 할당하는 것을 디스패치(Dispatch)라고 부른다. 단일 CPU 시스템에서는 오직 한 개의 프로세스만이 CPU를 사용할 수 있기 때문에 한 프로세스만이 실행 상태에 있을 수 있다. 만일 여러 개의 CPU가 있는 시스템이라면 동시에 여러 개의 프로세스가 실행 상태에 있을 수 있고 이때를 다중처리라고 부른다.
6. 실행 상태의 프로세스는 CPU 스케줄링 정책에 의해 CPU를 뺏길 수 있으며 이 경우 준비 상태로 바뀌게 된다. 특히, 시간 할당량이 소진되어 뺏길 때를 시간 종료(Timeout)라고 하고 이 경우도 인터럽트가 동원되어 처리된다. 또 다른 경우로, 실행 상태의 프로세스가 입출력이 필요하게 되어 시스템 호출을 하면 입출력 처리의 종료를 기다리면서 대기 상태로 바뀌게 되고 CPU는 바로 준비 상태의 프로세스들 중에서 하나를 선택에 실행해 줄 것이다.
7. 대기 상태는 프로세스가 실행되다가 입출력 처리를 요청하거나, 바로 확보될 수 없는 자원을 요청하면 CPU를 양도하고 요청한 일이 완료되기를 기다리면서 대기하는 상태이다. 대기 상태의 프로세스들 역시 이들의 관리를 위해 큐 혹은 리스트가 사용된다. 요청한 일이 완료되면 다시 실행 차례를 기다리기 위해 준비 상태로 바뀌면서 큐에 들어간다.
8. 종료 상태는 프로세스가 종료될 때 아주 잠시 거치는 상태이며 이 상태의 프로세스는 할당되었던 모든 자원들이 회수되고 PCB만 커널에 남아있는 상태이다. 운영체제가 시스템에 남겨져 있는 이 프로세스의 흔적들을 최종 정리 후 PCB를 삭제하면 프로세스가 완전히 사라진다.
9. 준비, 실행, 대기 상태들을 활성 상태라고 부른다. 활성 상태라는 것은 실행될 프로그램과 데이터 등을 위해 메모리 공간의 일정량을 부여 받는다. 다중 프로그래밍의 정도에 포함되는 프로세스들에 따라 다르다. 메모리가 부족하거나 다른 이유에 의해 시스템은 활성 상태의 프로세스로부터 메모리를 회수할 경우가 있는데 이를 보류라고 한다.
10. 한정된 메모리 공간의 여유가 없어지게 되면 시스템은 당분간 메모리를 회수해도 문제되지 않을 프로세스들을 골라 보류시켜 메모리 공간을 확보하게 된다. 메모리의 여유는 없는데 지금 생성된 프로세스가 더 중요한 것이라서 반드시 메모리를 줘야 될 때.
11. 입출력에 비해 턱없이 바른 CPU의 속도를 인정한다면, 메모리를 꽉 채우고 있는 프로세스들이 전부 입출력 중이라 CPU는 쉬고 있을 경우도 가능하다. 이때 새로운 프로세스를 받아들여 CPU를 가동시키고 싶다면 먼저 메모리를 확보해야 하고 입출력 때문에 대기 중인 프로세스들 중 누군가를 보류시켜야 한다.
12. 보류 상태는 프로세스가 메모리 공간을 뺏기고 디스크로 나가야 하는 걸 말한다고 했는데 이것을 스왑되어 나간다라고 하고, 나중에 다시 메모리로 들어오면 스왑되어 들어온 들어온다 라고 부르며 두 경을 통틀어 swapping이라 부른다.
13. 보류 준비 상태는 생성된 프로세스가 바로 메모리를 받지 못할 때나, 준비 또는 실행 상태에서 메모리를 잃게 될 때를 위해 필요하다. 실행 상태의 프로세스가 CPU를 반납하면서 준비 상태로 바뀔 때 메모리 공간까지 잃어야 하는 경우라면 보류 준비 상태로 바뀌게 된다. 충분한 메모리 공간의 확보를 위해 준비 상태의 프로세스를 보류시킬 수밖에 없을 경우나, 높은 우선순위의 보류 대기 상태 프로세스가 준비 상태가 되면서 실행 상태의 프로세스로부터 CPU를 뺏게 되는 경우가 여기에 해당.
14. 보류 준비 상태의 프로세스는 메모리의 여유가 생기거나 또는, 준비 상태의 프로세스가 전혀 없을 때 대기 상태의 프로세스를 보류 대기로 만들고 메모리 공간이 확보되면 준비 상태로 바뀌게 된다, 이때 위에서 말한 스와핑이 있게 되는 것이다.
15. 보류 대기 상태는 대기 상태일 때 메모리 공간을 잃은 상태이다. 바로 위에서 준비 상태의 프로세스가 아예 없어 대기를 보류 대기로 만드는 경우를 설명하였는데, 그 당시 준비 상태의 프로세스가 있었다고 하더라도 메모리의 여유 공간을 더 확보하기 위해서 보류 대기 상태가 되기도 한다. 보류 대기의 프로세스는 특별한 경우가 아니면 입출력이나 기다리던 사건의 종료 시 보류 준비 상태가 된다.
16. 보류 상태의 필요는 일차적으로 메모리 공간의 확보이지만 다른 이유 때문에 필요하기도 하다. 실행되는 프로세스의 현재 결과가 바라던 것이 아닌 오류가 보일 때, 시스템에 위해를 가할 수 있는 수상한 행동을 보일 때, 주기적인 일이라서 다음 주기의 실행 때까지의 메모리를 회수해도 문제되지 않을 때 등이 예가 되겠다.
17. 수행 중이라는 말은 실행 중이라는 것을 포함하는 뜻이므로 활성 상태는 물론 보류 상태에 있는 것까지 모두 프로세스이다.

**번외 context Switching**

인터럽트 이전에 실행되던 프로세스가 인터럽트를 처리한 후에 계속 실행이 될 경우도 있지만 다른 프로세스로 CPU가 할당되어 버릴 수도 있어요. 대표적인 예로 시간 종료나 우선순위가 이유일 경우도 있고, 입출력을 발생시킨 경우 역시 CPU는 다른 프로세스에게 넘겨지게 되는 거죠.

반면에 실행 중 다른 프로세스의 입출력 완료를 알리기 위한 인터럽트가 도착할 경우는 인터럽트의 처리 후 좀 전에 실행되던 프로세스를 그대로 계속 실행시켜주는 경우가 대부분이다. 문맥교환을 위해 필요한 일의 양은 인터럽트 처리 전후의 프로세스가 같으냐, 다르냐에 따라 차이가 나게 되는데 당연히 같을 때가 해줄 일이 적다. 같을 경우는 인터럽트 처리를 하는 동안 변경되어 버릴 가능성이 있는 최소한의 정보 – PCB의 일부분 – 만 보관되면 되지만, 다른 경우는 프로세스가 바뀜으로 인해 해주어야 할일이 더 많게 된다. 즉, 문잭의 저장은 물론이고, PCB에서 관련 데이터들의 변경과 함께 PCB를 적절한 상태의 큐로 옮겨야 하며, 실행될 프로세스를 고른 다음 –스케줄링 동원된다. – 이 프로세스의 PCB 변경과 함께 문맥의 복원과 같은 일이 요구되는 것이다. 그래서 인터럽트 처리 전후의 프로세스가 같은 경우 사용자 모드에서 인터럽트 처리를 위해 커널 모드로 바뀌고 처리가 끝나면 다시 사용자 모드로 바뀌면 된다는 의미에서 모드 스위칭이라고 부른다. 인터럽트 처리 전후의 프로세스가 달라지는 후자의 경우 프로세스 스위칭이라고 부른다. 두 경우 모두 문맥교환이 있지만 후자일 때 해주어야 할 일이 기술적으로 더 요구되기 때문에 프로세스 스위칭을 문맥교환이라고 하고, 모드 스위칭은 단순히 모두 스위칭이라고 부르기도 한답니다.

**Thread란**

1. 프로세스는 부여된 자원의 소유자로서, 스레드는 스케줄링의 단위로서 존재하게 된다. 한 프로세스에 속한 각각의 스레스들은 프로세스각 가지는 자원을 – 주소 공간 역시 자원이다. – 공유하면서 각자는 자신의 실행 환경 즉, 프로그램 카운터로 표현되는 현재의 실행 위치와 스택, 레지스터 값들을 따로 가지게 되는 것이다.

2. 다중 스레딩이란 하나의 프로세스를 다수의 스레드로 만들어 실행하는 것을 말하는데, 다중 스레딩을 하게 되면 하나의 프로세스 내에 다수의 실행 단위들이 존재하여 작업의 수행에 필요한 자원들 공유하기 때문에 자원의 생성과 관리가 중복되는 것을 줄일 수 있다

3. 한 프로세스 내의 다수개의 스레드 각각은 스레드의 수행 상태 예를 들어, 실행, 준비 등과 실행 상태가 아닐 경우를 위한 스레드 문맥, 각자의 실행 스택, 자신이 속한 프로세스가 가지는 메모리와 자원에 대한 접근 권한을 가진다.

4. 각 스레드는 자신의 제어 블록 – 실행 중의 레지스터 값, 우선순위, 또는 스레드와 관련한 상태 정보 등을 위해 필요한 자료구조이며 프로세의 PCB와 같은 개념으로 이해하면 된다. – 과 스택을 가진다. 동시에 프로세스의 정보인 PCB와 사용자 주소 공간은 공유함으로써 결과적으로 자신이 속한 프로세스의 상태와 자원들은 자연스럽게 공유하게 된다.

5. 따라서 한 스레드에 의해 메모리의 데이터가 변경될 경우 다른 스레드들은 변경된 데이터를 사용하게 되며, 열린 파일은 다른 스레드들에게도 열린 상태로 사용 된다.

6. 스레드의 가장 큰 장점은 성능 측면에서 발견할 수 있다. 다시 말해 스레드를 만들고, 없애며, 이들 간의 스위칭에 소요되는 시간과 비용이 프로세스 단위로 이루어질 때보다 빠르고 저렴하다는 것인데 실제로 맨 처음 예를 든 것처럼 관련된 작은 일 각각을 프로세스들로 만드는 것보다 스레드들로 만드는 것이 더 저렴하다는 것이다.

7. 프로세스 간의 통신 역시 커널의 개입을 필요로 하지만, 한 프로세스 내의 스레드 간 통신은 메모리와 파일을 공유하기 때문에 커널의 개입이 필요 없어 이 또한 이점이다.

8. 참고로 CPU의 할당 단위가 스레드라고 했으므로 CPU 스위칭을 위한 스레드 단위의 자료는 유지외어야 하며, 여전히 프로세스 단위로 행해지는 보류, 종료 등은 해당 프로세스에 속하는 전체 스레드에 동일안 영향을 미친다.

.

**스레드의 상태와 동기화**

1. 프로세스와 마찬가지로 스레드 역시 실행, 준비, 대기와 같은 상태를 가지며, 다만 보류는 프로세스 레벨의 개념이므로 스레드에서는 필요 없는 상태이다. 이미 말했듯이 대기는 레지스터 값, 프로그램 카운터, 스택 포인터 등의 보관이 요구되며, 스레드의 종료는 해당 스레드의 레지스터 값들과 스택을 없애게 된다.
2. Tcp의 대칭키 / 비대칭키
3. ㅁㅁ