**프로그램**은 하드디스크나 USB등 저장 장치에 저장된 실행 가능한 파일, 프로그램이 메모리에 적재되어 실행 중일 때 **프로세스**라 부름. 운영체제는 응용프로그램을 메모리에 적재하고 프로세스라고 부르고, 실행을 관리 CPU는 하드디스크에 저장된 상태에서 코드를 실행하지 않고, 메모리에 적재된 코드들만 실행해서, 프로그램이 실행되기 위해 코드와 데이터가 반드시 메모리에 적재 되어야함. **프로세스 특징** 1)운영체제는 프로그램을 메모리에 적재하고 이를 프로세스로 다룬다 2)운영체제는 프로세스에게 실행에 필요한 메모리를 할당하고 이곳에 코드와 데이터 등을 적재한다 3)프로세스들은 서로 독립적인 메모리 공간을 가지므로, 다른 프로세스의 영역에 접근할 수 없다 4)운영체제는 프로세스마다 고유한 번호를 할당 5)프로세스에 관한 정보는 운영체제 커널에 의해 관리 6)프로세스는 실행-대기-잠자기-실행-대기-잠자기-실행-종료 등의 생명 주기 가짐 7)프로세스를 만들고, 실행하고, 대기시키고, 종료 시키는 모든 관리는 커널에 의해 수행. 하나의 프로그램 여러 번 실행하면 프로그램 실행될 때마다 독립된 프로세스 생성 이 프로세스들을 프로그램의 다중 인스턴스라 부름. CPU주소공간이란 컴퓨터 내에 CPU가 접근 가능한 전체 메모리 공간, 크기는 CPU주소선 개수에 달림. 주소공간은 0번지부터 바이트 단위로 번지 매겨짐. 32비트CPU는 주소선이32개 2의32승개의 번지접근가능. 한번지의 공간크기 1바이트CPU주소공간의 크기는 2의32승바이트=4GB 2의40승은1TB. 프로세스는 4개의 메모리영역(사용자공간)이들을 합친 크기 CPU가 액세스할 수 있는 범위보다 클 수 없음**.1.코드영역**-프로세스코드가 적재되는 영역,텍스트영역으로도 불림.**2.데이터영역**-프로세스의 전역변수들과 정적변수들이 적재되는 영역**3.힙영역**-프로세스가 실행 중에 동적 할당 받는 영역.**4.스택 영역**-함수가 호출될 때,지역변수,매개변수, 함수의 리턴 값 등이 저장되는 영역.구체적으로 코드영역엔 사용자가 작성한 코드와 사용자코드에서 호출하는 라이브러리 함수의코드,데이터영역엔 사용자가 선언한 전역변수와 라이브러리에 선언된 전역변수들.두영역의 크기는 실행 중에 변하지 않음. 프로세스주소공간은 프로세스가 실행 중에 접근할 수 있도록 허용된 주소의 최대범위. 사용자 공간과 커널 공간 전체포함. 프로세스의 코드는CPU가 접근할 수 있는 모든 주소 범위에 접근허용,사실상 프로세스의 주소공간은 CPU의 주소 공간과 같다.프로세스의 주소 공간은 사용자,커널 공간포함.프로세스가 실행 중에 커널 함수를 실행하기 때문. 32비트전체주소공간4GB.사용자 공간3GB,커널공간 1GB.한프로세스는 코드, 데이터, 힙, 스택 합쳐 최대3GB사용가능.스택은 거꾸로 힙은 똑바로. 프로세스의 주소 공간 크기는 프로세스가 액세스할 수 있는, 다른 말로 커져 갈 수 있는 전체영역. 프로세스의 크기는 프로세스가 주소 공간 내에서 현재 사용하고 있는 코드, 데이터, 힙, 스택을 합친 크기.전체코드200MB,데이터크기1MB라고 프로세스의 크기201MB아님. 계속 변함. 프로세스의 주소 공간 가상공간임.가상주소. 프로세스 생성시 운영체제에 의해 코드, 데이터, 힙, 스택의 0번지부터순서대로 주소 매겨지고 할당되는데 가상 주소공간임. 프로세스의 가상주소공간과 물리메모리의 물리주소공간을 연결하는 매핑 테이블을 두고 주소공간 관리.커널 코드도 가상주소로 컴파일 되어있음.매핑 테이블은 프로세스마다 하나씩. 각 프로세스들이 시스템 호출을 통해 접근한 커널 공간을 동일한 물리 메모리 공간임. 결론적으로 프로세스는 자신의 매핑 테이블을 통해 물리 메모리에 접근하며, 긱 프로세스의 영역은 운영체제에 의해 물리 메모리의 서로 다른 공간에 배치되므로 프로세스들 사이의 가상 주소 공간은 충돌하지 않는다. 운영체제 커널은 시스템 전체에 하나의 프로세스 테이블을 두고 모든 프로세스의 정보를 관리.커널은 프로세스를 생성할 때마다 프로세스 제어블록(PCB)을 생성해 프로세스의 정보 저장.PCB와 프테는 커널 공간에 생성 커널만 액세스할 수 있다. PCB에 저장되는 프로세스 정보1)프로세스 번호, PID 커널은 프로세스를 생성할 때 PID를 할당.PID는 프로세스를 식별하는 고유한 번호로 사용자나 응용프로그램,운영체제 모두 PID로 프로세스 식별.(0과 양의 정수만 사용)2)부모 프로세스번호, PPID 부팅할 때 만들어진 최상위 프로세스 제외 모든 프로세스는 부모 프로세스를 가지고 PCB에 PPID저장. 3)프로세스 상태 정보**.New, Running, Ready, Blocked**가 있다. 커널은 프로세스의 상태를 바꿀 때마다 PCB에 상태 정보저장.4)프로세스 컨텍스트 정보. 커널은 현재 CPU가 실행중인 프로세스를 중단시키고 다른 프로세스를 실행시킬 때, 현재 프로세스가 실행중인 상황정보(컨텍스트)를 PCB에 저장.5)스케줄링 정보. PCB에는 커널이 스케줄링 시 참고하는 프로세스의 우선순위, 프로세스가 사용한 CPU시간, 최근에 CPU를 할당 받아 실행한 시간 등이 저장. 6)종료코드. 종료코드는 프로세스 종료 시, 종료이유를 부모 프로세스에게 전달하기 위한 정수 값으로, 종료한 프로세스의 PCB에 저장.exit()시스템 호출의 매개변수 값이나 main()함수의 리턴 값. 7)프로세스의 오픈 테이블 8)메모리 관리를 위한 정보들.9)프로세스 사이의 통신 정보들 10)회계정보 11)프로세스의 소유자 정보 12)프로세스가 현재 사용중인 입출력 장치의 리스트, 준비상태에 있는 다른 PCB에 대한 링크./ 프로세스의 생명 주기와 상태 변이 **New상태)** 프로세스를 생성하는 시스템 호출이 실행되면 새로운 프로세스가 생성. 커널은 새 프로세스의 코드와 데이터를 메모리에 적재, PCB를 만들어 프로세스 테이블의 빈 항목에 등록. 그리고 PCB에서는 프로세스 상태를 New로 기록.실행 준비를 마치면 Ready상태로 만듬. 대부분의 운영체제는 New상태에서 특별한 결정 없이 바로 Ready로 만들지만 실시간 운영체제의 경우 프로세스에 명시된 완료시한 내에 처리 가능하다고 판단될 때Ready로 만듬. **Ready상태)**Ready상태란 프로세스가 스케줄링을 기다리는 준비상태. Read상태의 프로세스들은 커널에 있는 준비 큐(ready queue)에 들어간다. 현재 실행 중인 프로세스가 종료되거나 중단되는 경우, 커널은 준비 큐에서 한 개의 프로세스를 선택(프로세스 스케줄링 또는 CPU스케줄링).스케줄링 정책 잘못되면 준비 큐에 오래 머무르는 기아 프로세스 생김.New상태에서 준비 큐에 삽입될 때, Running상태에서 프로세스에게 할당된 CPU시간이 경과되거나 스스로 다른 프로세스에게 CPU사용을 양보할 때, 혹은 입출력 장치나 저장 장치로부터 요청한 작업이 완료되었을 때Ready상태됨. **Running상태)**프로세스가 CPU에 의해 현재 실행되고 있는 상태. 커널은 CPU스케줄링을 통해 선택된 프로세스의 PCB상태를 Running으로 기록하고 CPU에게 프로세스를 실행하게 한다. 프로세스의 실행이 완료시 커널은 Terminated/Zombie상태로 만듬. 커널은 프로세스를 Running상태에서 다른 상태로 바꿀 때, 스케줄링을 시행하고, Running상태였던 프로세스의 컨텍스트를 PCB에 저장하고, 스케줄링 된 프로세스의 PCB에서 컨텍스트를 CPU에 복귀시킴(컨텍스트 스위칭). **Blocked/Wait상태)** 프로세스가 자원을 요청하거나, 입출력을 요청하고 완료를 기다리는 상태, 입출력이 완료되면 프로세스는 Ready상태로 바뀌고 준비 큐에 삽입. Running에서 시스템호출을 일으켰을 경우 커널은 현재 프로세스를 Blocked/Wait상태로 만듬. **Terminated/Zombie상태)** 프로세스가 불완전 종료된 상태(좀비상태)-프로세스가 차지하고 있던 메모리와 할당 받았던 자원들을 모두 커널에 의해 반환됨. 커널에 의해 열어 놓은 파일도 닫힘. 하지만 프로세스 테이블의 항목과 PCB가 여전히 시스템에서 제거되지 않은 상태. 프로세스가 남긴 종료코드(PCB에 있음)를 부모 프로세스가 읽어가지 않아 완전히 종료되지 않은 상태-좀비 상태라고도 부름. **Terminated/Out 상태)** 프로세스가 종료하면서 남긴 종료 코드를 부모 프로세스가 읽어 가서 완전히 종료된 상태. 프로세스 테이블의 항목과 PCB가 시스템에서 완전히 제거된 상태. 프로세스 스케줄링은 다중프로그래밍 운영체제에서 실행 중인 여러 프로세스 중 CPU를 할당할 프로세스를 결정하는 과정. 지금은 대부분 멀티스레드 운영체제. 스레드는 프로세스보다 작은 크기의 실행 단위, 개발자가 작업을 구현하는 단위이며 운영체제에 의해 스케줄링 되는 단위. 멀티 스레딩 기법의 출현으로 1개의 작업을 1개의 프로세스로 만드는 방식에서, 1개의 작업을 1개의 스레드로 만들고 프로세스 내에 여러 개의 스레드를 두어 여러 작업을 동시에 처리하는 방식으로 바뀜. 오늘날 운영체제는 프로세스 스케줄링 없이 시스템 전체 스레드를 대상으로 스레드를 선택하는 스레드 스케줄링을 실행한다.(프로세스 스케줄링이 없다고봐도됨). 프로세스는 프로세스에 의해 생성되며 생성한 프로세스를 부모 프로세스, 생성된 프로세스를 자식 프로세스라 함.#0프로세스제외 모든 프로세스는 부모 프로세스를 반드시 가지고 부모프로세스는 여러 개의 자식 프로세스를 가질 수 있다.#0프로세스는 시스템 내에 스케줄 될 프로세스가 하나도 없을 때 실행되는 유휴 프로세스이다. #1프로세스는 init프로세스 혹은 systemd프로세스라고 부르는데, 부팅 과정에서 시스템을 초기화하여 사용자가 시스템을 사용할 수 있는 상태까지 부팅을 완료한다. 그리고 시스템 종료시까지 살아있어 시스템종료과정을 책임진다. init프로세스는 외부 네트워크로부터 연결 요청을 대기하는 sshd프로세스를 생성하고, sshd는 외부 네크워크로부터의 접속을 받으면 사용자로부터 로그인 이름과 암호를 확인한 후 쉘 프로세스를 자식으로 생성한다.bash쉘은 스크린에 $를 출력하고 사용자 명령을 받아 자식 프로세스를 생성하고 자식에게 명령을 실행하게 한다. #2프로세스 kthredd는 부팅 때 생성되어 커널 공간에서 커널 모드로 실행되면서 커널의 기능을 돕는 프로세스이다. 응용프로그램과 전혀 관계없이 실행. fork()-자식프로세스를 생성하는 시스템 호출 함수 exit()-현재 프로세스의 종료를 처리하는 시스템 호출 함수 wait()-부모가 자식 프로세스가 종료할 때까지 기다리는 시스템 호출 함수. 운영체제들은 시스템 호출을 통해서만 프로세스르 생성하고, 시스템 호출을 통해서만 프로세스를 종료하도록 하고 있다. 부모 프로세스는 fork()시스템 호출을 이용해 자식을 생성 후, 자신의 작업을 수행하고 나서, 자식 프로세스의 종료를 확인하기 위해 wait()를 호출. Wait()는 자식 프로세스가 종료할 때까지 리턴 하지 않고 커널에서 대기.부모 프로세스는 wait()시스템 호출 내에서 자식 프로세스의 PCB에 남겨진 종료코드를 읽고 자식 프로세스를 완전히 제거하고 wait()에서 리턴 하여 실행을 계속.부모가 wait()를 호출하기 전 자식이 exit()를 먼저 호출해 종료한 경우 exit()는 프로세스가 소유한 모든 자원을 해제하지만, 프로세스 테이블 항목과 PCB는 삭제하지 않고 그냥 둔다. 그리고 PCB에는 프로세스가 남긴 종료코드를 기록함. 부모가 wait()호출 시 wait()은 자식의 PCB에 저장된 종료코드를 읽은 후 자식을 프로세스 테이블에서 제거하고 자식의 PCB도 삭제하여 자식 프로세스의 종료를 마무리한다. 자식이 종료할 때 부모가 반드시 확인하도록 하는 정책은 자식이 종료코드를 통해 종료원인을 부모에게 전달할 수 있도록 하고자 함. 종료했지만 부모 프로세스가 wait()를 호출안해 시스템에서 완전히 제거되지 않고 남아 있을 때 좀비 프로세스라고 한다. 리눅스 등 유닉스 계열의 운영체제에서는 프로세스가 종료할 때 커널은 프로세스가 남긴 종료코드를 PCB에 저장하고 프로세스의 상태를 Terminated/Zombie로 표시한다. 그리고PCB는 프로세스 테이블에서 제거하지 않고 남겨둔다. 부모 프로세스는 wait()를 실행해 자식 프로세스가 남긴 종료코드를 읽고, 자식 프로세스의 PCB와 프로세스 테이블 항목이 제거되도록 해야 한다. 자식 프로세스가 exit()를 실행하면,부모 프로세스에게 자식의 죽음을 통보한다(자식의 죽음을 알리는 SIGCHLD). 이때 부모가 wait()를 부르도록 작성되어 있지 않다면 죽은 자식 프로세스는 계속 좀비상태로 남음. windows운영체제는 CloseHandle(자식프로세스핸들) 좀비 프로세스를 제거하려면 부모프로세스에게 SIGCHLD신호를 보내면 되는데 부모 프로세스에 wait()를 호출하는 SIGCHILD신호핸들러가 작성되어 있다는 것을 전제로 한다. 좀비 프로세스의 부모 프로세스가 종료하면, 자동으로 #0init프로세스가 좀비 프로세스의 부모가 되며, init프로세스는 주기적으로 wait()를 실행하기 때문에 자식인 좀비 프로세스가 제거된다. 부모가 먼저 종료하면 자식들은 고아 프로세스됨. 부모 프로세스가 종료할 때 일반적으로 커널은 자식 프로세스가 있는지 확인하고 자식 프로세스를 **init프로세스에게 입양**. 운영체제에 따라, 혹은 쉘의 경우 모든 자식 프로세스 강제 종료 하기도 함. 백그라운드 프로세스는 터미널에서 실행되었지만, 터미널 사용자와의 대화가 없는 채 실행되는 프로세스, 사용자와 대화없이 실행되는 프로세스, 사용자 입력을 필요로 하지않는 프로세스, idle상태로 잠을 자거나 디스크에 스왑된 상태의 프로세스. 포그라운드 프로세스는 실행되는 동안 터미널 사용자의 입력을 독점하는 프로세스. CPU집중 프로세스는 대부분의 시간을 계산 중심의 일을 하느라 보내는 프로세스, 배열 곱, 인공지능 연산, 이미지처리 등의 작업, CPU속도가 성능을 좌우. I/O집중 프로세스는 입출력 작업을 하느라 대부분의 시간을 보내는 프로세스, 네크 워크 전송, 파일 입출력에 집중된 프로세스, 파일 서버, 웹 서버/, 입출력 장치나 입출력 시스템의 속도가 성능 좌우(I/O bound). 운영체제의 스케줄링 우선순위:I/O 집중 프로세스>CPU 집중 프로세스 -I/O작업하는 동안 다른 프로세스에게 CPU할당 가능. **컴퓨터 시스템에서 프로세스가 생성되는 5가지 경우** 시스템 부팅 과정에서 필요한 프로세스 생성, 사용자의 로그인 후 사용자와 대화를 위한 프로세스 생성, 새로운 프로세스를 생성하도록 하는 사용자의 명령, 배치 작업 실행 시, 사용자 응용프로그램이 시스템 호출로 새 프로세스 생성 커널 만이 프로세스를 생성하는 작업가능 **프로세스의 생성 과정** 새로운 PID 번호 할당->PCB구조체 생성->프로세스 테이블에서 새 항목 할당->새로 할당된 프로세스 테이블 항목에 PCB연결->새로운 프로세스를 위한 메모리 공간 할당(프로세스의 코드, 데이터 스택, 힙/할당 받은 메모리공간에 프로세스의 코드와 데이터 적재)->PCB에 프로세스 정보 기록->PCB에 프로세스 상태를 ready상태로 표시하고, 준비 큐에 넣어서 차후 스케줄 되게 함. fork()는 현재 프로세스를 복사하여 자식 프로세스를 생성/ 리턴 값-부모 프로세스에게는 자식 프로세스의 PID 자식 프로세스에게는 0, 실패시-1리턴 **프로세스 오버레이** 현재 실행중인 주소 공간에 새로운 응용프로그램을 적재하여 실행시키는 기법으로 exec()시스템호출이 사용. exec패밀리 시스템 호출 -execlp(), execv(), execvp()시스템 호출들, 실행 파일을 적재하여 현재 프로세스의 메모리 공간에 단순히 덮어쓰고, 새로운 프로세스의 생성 과정을 거치지 않는다./ 프로세스의 PID변경없음. 프로세스의 코드, 데이터, 힙, 스택에 새로운 응용프로그램이 적재됨, fork()에 의해 생성된 자식 프로세스는 생성 후 바로 exec()을 실행하는 경우가 다반사. **프로세스 종료와 프로세스 종료 대기** 정상종료위해 exit()호출 해야함. C프로그램에서 main()리턴 시 exit()시스템 호출이 실행되도록 컴파일. 종료코드-부모 프로세스에게 전달하는 값(main()함수의 리턴 값; return 종료코드;, exit(종료코드)) exit() 시스템 호출로 프로세스 종료 과정 1)프로세스의 모든 자원 반환(코드,데이터,스택,힙 등의 모든 메모리 자원을 반환, 열어 놓은 파일이나 소켓 등 닫음) 2)PCB에 프로세스 상태를 Terminated로 변경, PCB에 종료 코드 저장 3)자식 프로세스들이 있으면 이들을 init프로세스에게 입양 4)부모 프로세스에게 SIGCHLD신호 전송(부모가SIGCHLD신호 핸들러를 작성하고 여기서 wait() 시스템 호출을 이용하여 자식의 종료 코드 읽기 실행, 혹은 언젠가 부모가 자식의 죽음 처리, 그동안 자식은 좀비 상태에 있음) 종료코드는 프로세스가 종료한 상태나 이유를 부모에게 전달하기 위한 것으로POSIX표준에서 0~255사이의 1바이트숫자.(정상종료는0, 1~255의 값은 개발자가 종료이유를 임의로 정해 사용).main 이나 exit()에서 return 300,exit(300)은 44와 같음. -1리턴시(return -1,exit(-1)) -1>0xff>양의 정수로 255. 종료코드255전달. **프로세스 종료와 좀비 프로세스** 1)C 언어에서 main()함수의 종료나 exit()을 호출한 정상종료 2)다른 프로세스에 의해 강제종료(kill) 프로세스가 종료되면 차지하고 있던 메모리와 자원 모두 반환,PCB는 프로세스 테이블에서 제거되지 않음, 프로세스 상태:Terminated/ 부모프로세스가 wait()시스템 호출을 통해, 죽은 자식이 남긴 종료 코드를 읽게 되면 ,자식 프로세스의 PCB가 완전히 제거/ 좀비 프로세스-종료할 때 리턴한 정보(main()함수에서 리턴 값, 종료코드)를 부모 프로세스가 읽지 않았을 때, 죽었지만 PCB만 남아 완전히 제거되지 못한상태 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////컴퓨터에서 처리하고자 하는 일의 단위를 작업 혹은 태스크라고 부름. 멀티태스킹은 컴퓨터 시스템 내에 여러 태스크를 동시에 실행시키거나 한 응용프로그램 내에서 여러 태스크를 동시에 실행시키는 기법.(사용자나 개발자 입장에선 동시에 여러 작업 처리할 수 있는 장점, 운영체제 입장에선 동시에 실행되는 여러 개의 태스크를 관리해야 하는 문제점) **프로세스를 실행 단위로 하는 멀티태스킹의 문제점 1)프로세스 생성의 큰 오버헤드** 프로세스의 생성에 너무 많은 시간이 걸린다. 프로세스를 위해 메모리 할당, 부모프로세스를 그대로 복사, PCB생성, 매핑 테이블 생성 등 **2)프로세스 컨텍스트 스위칭의 큰 오버헤드** 현재 실행중인프로세스를 중단시키고 다른 프로세스를 실행시키는 컨텍스트 스위칭에 시간적 공간적 오버헤드가 큼. (CPU레지스터들을 프로세스 컨텍스트에 저장, 새프로세스 컨텍스트를 CPU로 옮기는 시간). CPU가 참고할 매핑 테이블 전환에 따른 지연 시간 등, CPU캐시에 새로운 프로세스의 코드와 데이터가 채워지는데 걸리는 시간 등 **3)프로세스 사이의 통신 어려움** 프로세스들이 각각 독립된 주소 공간을 갖고 있어 프로세스가 다른 프로세스의 메모리에 접근 불가. 프로세스 사이의 통신을 위한 제 3의 방법 필요(커널 메모리나 커널에 의해 마련된 메모리 공간을 이용하여 데이터 송수신, 이 방법들은 코딩하기 어렵고, 느린 실행 속도, 운영체제 호환성 부족), **스레드 출현 목적** 프로세스를 사용하는 문제점 해결을 위해 고안 1)프로세스보다 더 작은 실행 단위 필요 2) 프로세스의 생성 및 소멸에 따른 오버헤드 저감 3) 빠른 컨텍스트 스위칭 4) 프로세스의 복잡한 통신 방법, 느린 실행 속도, 코딩의 어려움 해소/ 멀티스레드 운영체제 : 스레드를 실행 단위로 다루는 운영체제(프로세스는 기업, 스레드는 직원에 비유) **스레드 개념 1)**스레드는 실행 단위이며 스케줄링 단위 – 스레드는 응용프로그램 개발자에게는 작업을 만드는 단위, 운영체제에게는 실행 단위, CPU를 할당하는 스케줄링 단위, 코드, 데이터, 힙, 스택을 가진 실체, 스레드마다 정보를 저장하는 구조체 TCB있음 **2) 프로세스는 스레드들의 컨테이너** – 프로세스를 스레드들의 컨테이너 개념으로 수정, 프로세스는 반드시 1개 이상의 스레드로 구성, 프로세스가 생성될 때 운영체제에 의해 자동으로 1개의 스레드 생성 : 메인 스레드라고 부름, PCB와 TCB의 관계(그림 참고, T2 Program Counter와 T3 Program Counter다름 TCB에서 갖고있다 컨텍스트 스위칭 시 넘겨줌) **3)프로세스는 스레드들의 공유 공간 제공** 모든 스레드는 프로세스의 코드, 데이터, 힙을 공유하며, 프로세스의 스택 공간을 나누어 사용, 공유되는 공간을 이용하면 스레드 사이의 통신 용이 **4)스레드는 함수로 작성** 응용프로그램에서 스레드가 실행할 작업을 함수로 작성, 스레드를 만들어줄 것을 운영체제에게 요청해야 비로소 스레드 생성, 운영체제는 함수의 주소를 스레드 실행 시작 주소로 TCB에 등록, 운영체제는 TCB리스트로 스레드 관리, 스레드 단위로 스케줄, TCB에는 스레드의 시작 주소 기록됨. **5)스레드의 생명과 프로세스의 생명** 스레드로 만든 함수가 종료하면 스레드 종료, 스레드가 종료하면 TCB 등 스레드 관련 정보 모두 제거, 프로세스에 속한 모든 스레드가 종료될 때, 프로세스 종료 **맛보기 프로그램을 통해 알 수 있는 개념** 1)프로세스가 생성되면 자동으로main 스레드 생성(main 스레드는 main()함수 실행) 2)스레드 코드는 함수로 만들어진다.(calcThread()함수) 3)스레드 생성(스레드는 pthread\_create() 등 라이브러리 함수나 시스템 호출을 통해 생성) 4)스레드마다 TCB 1개 생성(TCB에는 스레드의 시작주소 저장, 이 주소에서 실행 시작, 컨텍스트 스위칭되어 실행을 재개할 주소 저장, TCB의 존재를 스레드의 존재로 인식하면 됨) 5)스레드는 스케줄링 되고 실행되는 실행 단위 6)프로세스는 스레드들의 컨테이너(프로세스는 더 이상 실행 단위 아님, 맛보기 프로세스 내에 main스레드와 calcThread 있음, 두 스레드가 종료되어야 프로세스 종료) 7)프로세스는 스레드들에게 공유 공간 제공(프로세스의 코드와 전역 변수는 모든 스레드에 의해 공유, 사례에서 main 스레드와 calcThread 스레드는 sum 변수 공유) 일반적으로 함수는 다른 함수에 의해 호출되어 실행되지만 스레드 함수의 코드는 커널에 의해 직접 CPU가 실행하도록 제어됨. **Concurrency(동시성)**1개의 CPU에서 2개 이상의 스레드가 동시에 실행 중인 상태(스레드가 입출력으로 실행이 중단될 때 다른 스레드 실행, 타임 슬라이스 단위로 CPU를 사용하도록 번갈아 스레드 실행) 사례 : 3개의 스레드가 1개 CPU에 의해 동시 실행 **parallelism(병렬성)** 2개 이상의 스레드가 다른 CPU에서 같은 시간에 동시 실행,사례 : 3개의 스레드가 3개의 CPU에 의해 동시 실행 **스레드의 주소 공간**은 스레드가 실행 중에 사용하는 메모리 공간으로 스레드의 코드, 데이터, 힙, 스택 영역이며, 모두 주소 공간에 형성. **스레드 주소 공간의 요소들** 스레드 사적 공간 -스레드 코드, 스레드 로컬 스토리지/스레드 사이의 공유 공간-프로세스의 코드, 프로세스의 데이터 공간, 프로세스의 힙 영역 **스레드 코드 영역** 스레드가 실행할 작업의 함수(프로세스의 코드 영역 사용), 스레드는 프로세스의 코드 영역에 있는 다른 모든 함수 호출 가능 **스레드 데이터 영역** 스레드가 사용할 수 있는 데이터 공간(프로세스의 데이터 영역 사용), 2개의 공간으로 구분(개별 스레드 전용 전역 변수 공간(스레드 로컬 스토리지:스레드가 안전하게 지키고자 하는 데이터를 저장하기 위해 사용)-static –thread와 같은 특별한 키워드로 선언, 컴파일러에 의해 결정, 프로세스에 선언된 모든 전역 변수들은 모든 스레드에 의해 공유-스레드 사이의 통신 공간으로 유용하게 사용)**스레드 힙** 모든 스레드가 동적 할당받는 공간, 프로세스의 힙 공간 사용, 스레드에서 malloc()를 호출하면 프로세스의 힙 공간에서 할당받음. **스레드 스택** 스레드가 생성될 때마다 프로세스의 사용자 스택의 일부분을 할당받음, 스레드가 시스템 호출로 커널에 진입할 때, 커널 내에 프로세스에게 할당한 커널 스택에서 스레드를 위한 스택 할당. 아래 사진은 실행되는 동안 main스레드를 포함해 총 3개의 스레드가 실행. Tsum은 각스레드의 TLS영역에 스레드마다 생기고 스레드에 의해 사적으로 사용. Total은 프로세스 전체에 하나만 생기고 프로세스에 속한 모든 스레드에 의해 공유. **스레드 상태** 스레드는 생성, 실행, 중단, 실행, 소멸의 여러 상태를 거치면서 실행된다. 스레드 상태는 TCB에 저장된다. 준비 상태(Ready)스레드가 스케줄 되기를 기다리는 상태, 실행 상태(Running)-스레드가 CPU에 의해 실행 중인 상태, 대기 상태(Blocked)스레드가 입출력을 요청하거나 sleep()과 같은 시스템 호출로 인해 중단된 상태, 종료 상태(Terminated)-스레드가 종료한 상태 **스레드 생성** 스레드는 스레드를 생성하는 시스템 호출이나 라이브러리 함수를 호출하여 다른 스레드 생성 가능, 프로세스가 생성되면 자동으로 main스레드 생성 main스레드 제외 스레드는 스레드에 의해 생성되는데 이들을 부모-자식 스레드라 부른다. 부모스레드 가 종료한다고 자식이 종료하지 않고 자식 스레드 종료한다고 부모 스레드에 통보되는 것 아님. 단 부모스레드는 자식스레드를 생성했기 때문에 자식스레드에 대한 스레드ID를 갖고있어 제어가능. **스레드가 생성되는 과정** - TCB구조체를 만든 후 스레드에게 ID를 부여한다. 그리고 스레드가 실행을 시작할 코드(함수)의 주소를 TCB의 PC에 기록하고, 스레드 사용자 스택을 할당하고 그 주소를 TCB의 SP에 저장한 후, 스레드 상태를 Ready로 하고 스레드를 준비 리스트에 넣는다. TCB를 프로세스의 PCB와 다른 TCB에 연결하면 끝. 스레드 생성과정은 프로세스에 비해 매우 단순. 이미 적재된 프로세스의 영역 속에서 스레드가 생성되므로 메모리 관련된 초기작업이 매우 적기때문. **스레드 종료** 프로세스 종료와 스레드 종료의 구분 필요 /프로세스 종료) 프로세스에 속한 아무 스레드가 exit()시스템 호출을 부르면 프로세스 종료(모든 스레드 종료), 메인 스레드의 종료(C프로그램에서 main()함수 종료)-모든 스레드도 함께 종료, 모든 스레드가 종료하면 프로세스 종료 /스레드 종료) pthread exit()와 같이 스레드만 종료하는 함수 호출 시 해당 스레드만 종료(,프로세스에 속한 마지막 스레드가 종료되면 프로세스가 종료), main()함수에서 pthread\_exit()을 부르면 main스레드만 종료(스레드의 생성과 종료가 빠른 것이 바로 프로세스 대신 스레드를 사용하는 이유) **스레드 조인** 스레드가 다른 스레드가 종료할 때까지 대기(주로 부모 스레드가 자식 스레드의 종료 대기) 스레드 번호(tid)만 알면 아무 스레드나 다른 스레드를 조인할 수 있지만, 스레드 조인은 부모 스레드가 자식스레드를 생성하여 작업을 시키고 자식 스레드가 작업을 완료하기를 기다릴 때 주로 사용된다. **스레드 양보** 스레드가 자발적으로 yield()와 같은 함수 호출을 통해 스스로 실행을 중단하고 다른 스레드를 스케줄하도록 요청.양보한 스레드는 Ready상태로 준비큐에 들어가고, 준비큐에 있는 스레드 중 하나가 스케줄링되어 실행. 준비 큐에 아무 스레드도 없으면 양보한 스레드가 다시 실행. **스레드 컨텍스트** (스레드가 현재 실행중인 일체의 상황(상태 정보)) 메모리엔 코드와 데이터가 있고 스택에는 현재 실행 중인 함수의 매개변수와 지역변수 등이 저장되어 있다. CPU의 **PC레지스터**는 현재 실행 중인 스레드의 코드 주소가, **SP레지스터**는 스레드 스택의 톱 주소가, **여러 데이터 레지스터**에는 실행 결과나 실행에 사용될 데이터들이, 상태 레지스터에서는 CPU의 상태 정보와 프로그램에 의해 설정된 제어 정보들이 들어있다. 스레드 코드, 데이터, 스택 등 메모리에 저장된 정보는 프로세스의 공간에 그대로 있기 때문에, CPU레지스터들만 저장하면 현재 실행 중인 스레드의 컨텍스트 저장가능. 따라서 스레드 컨텍스트는 CPU가 스레드를 실행되고 있을 때 CPU의 레지스터 값들이다. **스레드 제어 블록(TCB)** 스레드를 실행 단위로 다루기 위해 스레드에 관한 정보를 담은 구조체(스레드 엔터티, 스케줄링 엔터티라고도 불림), 커널 영역에 만들어지고, 커널에 의해 관리(스레드가 생성될 때 커널에 의해 만들어짐, 스레드가 소멸되면 함께 사라짐)스레드의 주소 공간이 프로세스 내에 있기 때문에, PCB와 프로세스 내에 속한 모든 스레드의 TCB들을 링크드 리스트로 연결하여 관리.TCB도 자신이 속한 프로세스의 PCB에 대한 링크를 가지고 있음. 프로세스들은 스레드들이 생기고 활동하는 자원의 컨테이너로서, 운영체제는 프로세스를 스케줄링 단위가 아니라 스레드들을 위한 자원의 할당 단위로 다룬다. 커널이 실행 단위로 인식하는 스레드 실체는 TCB이다. **준비리스트** 준비 상태에 있는 스레드들의 TCB를 연결하여 관리하는 링크드 리스트, 스레드 스케줄링은 준비 리스트의 TCB들 중 하나 선택 **블록 리스트** 블록 상태에 있는 스레드들의 TCB를 연결하여 관리하는 링크드 리스트 **스레드 컨텍스트 스위칭** 현재 실행중인 스레드를 중단시키고, 다른 스레드에게 CPU할당(스레드 스위칭이라고도함.)현재 CPU컨텍스트를 TCB에 저장하고, 다른 TCB에 저장된 컨텍스트를 CPU에 적재. **스레드 스위칭이 발생하는 경우** 1)스레드가 자발적으로 다른 스레드에게 양보 – yield()등의 시스템 호출(혹은 라이브러리 호출)을 통해. Sleep()나 wait()등의 시스템 호출을 불러 다른 스레드로 컨텍스트 스위칭되는 경우 2)스레드가 I/O 작업을 요청하는 시스템 호출 시 블록되는 경우(시스템 호출 내) read(), sleep(), wait() 등 I/O가 발생하거나 대기할 수 밖에 없는 경우 3)스레드의 타임 슬라이스(시간 할당량)를 소진한 경우 – 타이머 인터럽트에 의해 스레드의 실행 시간을 체크하고 스레드가 CPU타임 슬라이스를 소진한 경우 강제로 중단시켜 준비 리스트에 넣음. 그리고 준비 리스트에서 다른 스레드를 선택한 후 컨텍스트 스위칭 4)I/O장치로부터 인터럽트가 발생한 경우 – 현재 실행중인 스레드보다 더 높은 우선순위의 스레드가 블록 상태로 I/O완료를 기다리고 있는 상황에서, I/O작업이 완료되어 I/O장치로부터 인터럽트가 발생하면, 인터럽트 서비스 루틴은 현재 스레드를 강제로 중단시켜 준비 리스트로 옮기고, I/O작업 완료를 대기중인 높은 순위의 스레드로 컨스. **스레드 스위칭이 이루어지는 위치** 1. 프로세스가 시스템 호출을 하여, 커널이 시스템 호출을 처리하는 과정에서 2. 인터럽트가 발생하여 인터럽트 서비스 루틴이 실행되는 도중 커널 코드에서 **스레드 스위칭 과정** 1)CPU레지스터 저장 및 복귀 현재 실행중인 스레드 A의 컨텍스트를 TCB-A에 저장. TCB-B에 저장된 스레드B의 컨텍스트를 CPU에 적재(CPU는 스레드 B가 이전에 중단된 위치에서 실행 재개 가능, SP레지스터를 복귀함으로서 자신의 이전 스택을 되찾게 됨(스택에는 이전 중단될 때 실행하던 함수의 매개변수나 지역변수들이 그대로 저장되어 있음.)). 2)커널 정보 수정 TCB-A와 TCB-B에 스레드 상태 정보와 CPU 사용 시간 등 수정, TCB-A를 준비 리스트나 블록 리스트로 옮김, TCB-B를 준비 리스트에서 분리. **컨텍스트 스위칭 오버헤드** 컨텍스트 스위칭은 모두 CPU작업이라 CPU시간 소모. 컨텍스트 스위칭의 시간이 길거나, 잦은 경우 컴퓨터 처리율 저하. **동일한 프로세스 내 다른 스레드로 스위칭되는 경우** 1)컨텍스트 저장 및 복귀 시간 -현재 CPU의 컨텍스트(PC,PSP,레지스터) TCB에 저장, TCB로부터 실행할 스레드의 스레드 컨텍스트를 CPU에 복귀 2)TCB리스트 조작 -스레드의 TCB를 준비 리스트나 블록 리스트 등으로 옮기는 시간 3)캐시 플러시와 채우기 시간 현재 CPU캐시에 담겨 있는 스레드 코드와 데이터를 지우거나 변경된 데이터를 메모리로 복사하는 시간이 소요된다. 새로 실행될 스레드의 코드와 데이터가 CPU캐시에 채워지는 것은 컨텍스트 스위칭 이후 새 스레드가 실행하는 동안 이루어지므로 컨텍스트 스위칭 시간에 포함하지 않음. **다른 프로세스의 스레드로 스위칭하는 경우** 다른 프로세스로 교체되면, CPU가 실행하는 주소 공간이 바뀌는 큰 변화로 인한 추가적인 오버헤드 발생 1)추가적인 메모리 오버헤드(시스템내에 현재 실행 중인 프로세스의 매핑 테이블을 새로운 프로세스의 매핑 테이블로 교체) 2)추가적인 캐시 오버헤드(프로세스가 바뀌기 때문에, 현재 CPU캐시에 담긴 코드와 데이터를 무력화시킴, 새 프로세스의 스레드가 실행을 시작하면 CPU캐시 미스 발생, 캐시가 채워지는데 상당한 시간 소요) / 컨스는 CPU시간이 소모되는 CPU집중 작업으로, 컨스 시간이 클수록, 스위칭 횟수가 많을수록 컴퓨터의 처리율이 떨어져 컨스 시간을 최소화할 필요있음. 멀티 코어 CPU를 가진 현대의 컴 시스템에서는 프로세스를 특정 CPU코어에 배치하여, CPU코어가 여러 프로세스에 걸쳐 스레드를 실행하지 않도록 하는 방법을 사용하기도 하고, CPU와 TCB사이에 컨텍스트를 이동하는 작업을 없애기 위해 CPU에 스레드 별로 레지스터 셋을 따로 두어, 컨텍스트 스위칭 시 현재 실행 중인 스레드의 CPU레지스터들을 TCB에 저장하지 않는 방법 쓰기도함. 스레드 스케줄링 주체에 따라 2종류로 구분 1)커널 레벨 스레드: 커널에 의해 스케줄링되는 스레드 2)사용자 레벨 스레드: 스레드 라이브러리에 의해 스케줄링되는 스레드/ **커널 레벨 스레드** 응용프로그램이 시스템 호출을 통해 커널 레벨 스레드 생성, 스레드에 대한 정보(TCB)는 커널 공간에 생성되며 커널에 의해소유. 커널에 의해 스케줄, 스레드 주소 공간(스레드 코드와 데이터):사용자 공간에 존재, main 스레드는 커널 스레드(응용프로그램을 적재하고 프로세스를 생성할 때 커널은 자동으로 main스레드 생성 **사용자 레벨 스레드** 응용프로그램이 라이브러리 함수를 호출하여 사용자 레벨 스레드 생성, 스레드 라이브러리가 스레드 정보(U-TCB)를 사용자 공간에 생성하고 소유(스레드 라이브러리는 사용자 공간에 존재, 커널은 사용자 레벨 스레드의 존재에 대해 알 수 없음), 스레드 라이브러리에 의해 스케줄, 스레드 주소공간(스레드 코드와 데이터):사용자 공간에 존재. **순수 커널 레벨 스레드** 부팅 때부터 커널의 기능을 돕기 위해 만들어진 스레드, 커널 코드를 실행하는 커널 스레드, 스레드의 주소 공간은 모두 커널 공간에 형성, 커널 모드에서 작동, 사용자 모드에서 실행되는 일은 없음. **커널 레벨 스레드와 사용자 레벨 스레드의 장단점** 1)컨텍스트 스위칭과 스레드 관리-사용자레벨스레드가 더 효율적.커널 레벨 스레드 사이의 컨텍스트 스위칭은 커널 공간에서 이루어지며, 커널 내 여러 자료 구조를 수정하는 작업이 일어남, 사용자 레벨 스레드사이의 컨텍스트 스위칭은 스레드 라이브러리에 의해 사용자 공간에서 진행되서 커널모드로 바꾸는 등 커널로 진입하는 부담도 없고 실제 CPU에 컨텍스트를 쓰거나 읽어오는 작업이 없다**. 2)이식성 및 활용성** 사용자 레벨 스레드가 우세. 스레드 라이브러리에 의해 관리되어, 사레스로 작성된 응프는 운체에 종속적이지 않음. 또 멀티스레드 미지원 운영체제에서도 스레드 라브를 이용해 멀티스레드 응프 작성가능. **3)실행 블록킹** 사레스로 작성된 멀스 응프의 경우, 임의의 사레스가 read()나 write()시스템 호출을 실행해 입출력 완료를 기다리는 블록 상태가 될 때, 응프 내 다른 모든 사용자 레벨 스레드가 실행될 수 없게 된다. 하지만 커레스로 작성된 멀스 응프는 임의의 커레스가 중단(블록)되도, 응프내 다른 커레스가 스케줄링되어 실행가능해 응프전체가 중단되진 않음. **4)병렬성** 멀티 코어 CPU를 가진 컴퓨터에서 커널은 CPU코어마다 하나의 커널 레벨 스레드 실행. 4개 코어 가진 CPU있다면 4개의 커레스 동시실행가능. 사레스는 하나의 CPU코어에서 스레드라이브러리에 의해 번갈아 실행. **멀티스레드 구현** 응용 프로그램에서 작성한 스레드가 시스템에서 실행되도록 구현하는 방법(사용자가 만든 스레드가 시스템에서 스케줄되고 실행되도록 구현하는 방법, 스레드 라이브러리와 커널의 시스템 호출의 상호 협력 필요**) 1)N:1매핑**(N개의 사레스를 1개의 커레스로 매핑) 운영체제는 모든 프로세스를 단일 스레드 프로세스로 다룸. 프로세스 당 1개의 커널 레벨 스레드(TCB) 생성(스케줄 가능한 엔터티라고 부름, 프로세스의 모든 사용자 레벨 스레드가 1개의 커널 레벨 스레드에 매핑), 사용자 레벨 스레드는 스레드 라이브러리에 의해 스위칭됨. 매핑이란 사레스는 해당 커레스가 스케줄되어야 실행 가능하도록 묶여있는 것. **장점**-단일 코어CPU에서 멀티스레드 응프의 실행 속도가 전반적으로 빠름(스레드 생성, 스케줄, 동기화 등에 있어 커널로 진입없이 사용자 공간에서 이루어져서) **단점**-멀티코어 CPU가 보편화된 현대 컴에서 비효율, 하나의 사레스가 블록되면 프로세스 전체 블록 **2)1:1매핑**(1개의 사레스 1개의 커레스로 매핑) 사레스는 매핑된 커레스가 스케줄되면 실행 **장점**-개념이 단순하여 구현이 용이, 멀티코어 CPU에서 멀티스레드 응프에게 높은 병렬성 제공, 하나의 사레스가 블록되도 응프 전체가 블록안됨. **단점**-커널에겐 부담스러움, 사레스가 많아지면 모두 커널의 부담. **3)N:M매핑**(N개의 사레스 M개의 커레스) **장점**-1:1매핑에 비해 커널 엔터티 개수가 작아 커널의 부담이 적다. **단점**-구현하기 복잡하여 현대의 운영체제에서는 거의 사용되지 않음. **프로세스와 스레드 정리** 프로세스는 스레드들의 코드와 데이터, 힙, 스택이 생성되고 실행되는 주소 공간이며 공유 공간이다(모든 스레드의 주소 공간이 프로세스 주소 공간 내에 형성되고 공유) 2)프로세스는 운영체제가 응프를 적재하는 단위이며, 실행 단위는 스레드이다. 3) PCB에 저장된 정보를 환경 컨텍스트, TCB에 저장된 정보를 실행 컨텍스트라 하는데 PCB에 저장되는 내용은 모든 스레드가 공유하는 프로세스의 정보가 저장, TCB에는 현재 실행중인 실행 단위의 정보만이 저장 4)다른 프로세스에 속한 스레드로의 스위칭보다 동일한 프로세스에 속한 스레드 스위칭 속도가 빠름. 5)프로세스에 속한 모든 스레드가 종료할 때 프로세스 종료. **멀티스레딩으로 응프 작성시 장점** 1)높은 실행 성능(병렬실행) 2)사용자에 대한 우수한 응답성(한 스레드가 블록되어도 다른 스레드를 통해 사용자 인터페이스 가능) 3)서버 프로그램의 우수한 응답성(웹서버나 파일 서버 등은 동시에 많은 사용자들의 접근, 이들을 병렬적으로 서비스하는데 우수) 4)시스템 자원 사용의 효율성(스레드는 프로세스에 비해 생성, 유지 시 메모리나 자원 적게 사용) 5)응용프로그램 구조의 단순화(작업 기준으로 응프를 여러 함수로 분할하고, 각 함수 별로 스레드를 만들어 동시 실행, 새로운 기능 추가용이, 프로그램의 높은 확장성) 6)작성이 쉽고 효율적인 통신 **멀티스레딩에 있어 주의할 점 1)**여러 개의 스레드 중 한 스레드가 fork()를 호출한 경우(새로 생성된 프로세스는 fork()를 호출한 스레드로만 구성, 심각한 문제) 2)한 스레드가 exec()를 호출한 경우(현재 프로세스의 모든 스레드가 사라지는 문제) 3)스레드 사이의 동기화 문제(다수 스레드의 공유 데이터에 대한 엑세스 시 공유 데이터 훼손 가능성->동기화 기법으로 해결 **부록!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!** 신호-운영체제가 응요프로그램(프로세스)에게, 프로세스가 프로세스에게 비동기적 사건을 알리는 방법으로 통신 -1970년 Bell Lab의 Unix서 처음 사용, 지금도 사용 -2개의 시스템 호출 함수 사용(signal()-신호 핸들러를 등록하는 시스템 호출, kill()-신호를 보내는 시스템 호출) **신호를 수신하게 되는 3가지 경우** 1)프로세스 자신으로부터 신호 수신 2)다른 프로세스로부터 신호 수신(kill() 시스템 호출을 이용하여 다른 프로세스가 신호 받기) 3)커널로부터 신호 수신(프로세스의 명령 실행 중 불법 명령이나 0으로 나누기 발생, 오버플로우 발생, 허용되지 않거나 할당 받지 않은 메모리 영역을 접근하는 경우, 자식 프로세슷 종료, 시스템 알람) **신호 종류** 운영체제마다 다르지만 대력 30개, 신호 번호와 이름(모든 신호는 고유한 번호로 구분됨, 각 신호는 SIG로 시작하는 고유한 이름 있음, 신호의 번호는 시스템마다 다를 수 있기 때문에 응용프로그램 개발자는 신호 이름으로 프로그램을 작성할 것) **신호 처리에 있어 디폴트 행동**신호핸들러가 등록되어 있지 않을 때, 커널이 취하는 4가지 행동 1)종료(terminate)신호를 수신할 프로세스 강제 종료 2)코어덤프(core sump)신호를 수신할 프로세스를 강제 종료시키고 코어파일 생성 3)중지(stop)신호를 수신할 프로세스의 실행을 일시 중지. 중지된 프로세스는 이후 계속 실행 가능 4)무시(ignore)신호를 무시하고 아무 처리도 하지 않음. **신호등록** signal()시스템 호출, 신호를 수신할 프로세스는 신호 핸들러 등록 **신호 발생** 목적 프로세스에게 신호 전송, kill(PID,sig)시스템 호출/ 처리과정 – 수신프로세스의 PCB에 sig의 신호 발생 표시하고 리턴, 이미 sig의 신호가 미처리 상태로 있으면 표시하지 않고 리턴/ **신호와 관련된 2개의 비트 벡트(프로세스마다 있음)** -미처리 비트 벡터(pending bit vector)신호별로 1비트, 신호가 도착하여 처리를 기다리고 있음을 나타내는 벡터, 동일 타입의 신호가 2개 도착한다고 해도 1번만 도착한 것으로 처리 -거부 비트 벡터(blocked bit vector)신호별로 1비트, 신호에 대해 핸들러 실행을 일시 거부하기 위해 사용하는 벡터, 비트 값이 1인 경우라도, 신호가 도착하면 미처리 벡터에 표시. 미처리 벡터에 표시된 신호는 거부가 해제되면 실행 **신호 핸들러가 실행되는 두 시점** 1. 신호를 수신한 프로세스가 스케줄링되어 실행되기 바로 직접, 사용자 모드에서 2. 신호를 수신한 프로세스가 시스템 호출 중에 신호가 도착한 경우, 시스템 호출이 어떤 것이든 시스템 호출을 끝내고 사용자 모드로 돌아간 바로 직후 신호를 수신한 프로세스가 사용자 모드에서 실행 중일 때 수신한 신호의 핸들러는 실행되지 못함. **신호전파** 도착한 신호가 있는 경우, 해당 프로세스의 신호 핸들러가 실행되도록 하는 과정 1)커널은 스케줄된 프로세스의 실행을 재개시키기 전이나 프로세스가 시스템 호출을 끝내고 사용자 모드로 돌아가기 전 2)프로세스의 미처리 비트 벡터(pending bit vector)를 검사 3)도착한 신호(pending signal)가 있는지 확인 4)도착한 신호가 있다면, 커널은 프로세스의 스택을 조작하여 프로세스가 사용자 모드에서 실행을 시작할 때 신호 핸들러의 시작 주소에서부터 실행하도록 하고, 신호 핸들러에서 리턴하면 원래 예정되었던 사용자 주소로 돌아가도록 조작 5)커널이 프로세스에게 도착한 신호가 있는지 확인할 때, SIGKILL 신호가 있다면 커널은 수신 프로세스를 사용자 모드로 돌려 실행시키지 않고 바로 종료 6)도착한 신호가 SIGSTOP이라면, 프로세스를 중단시켜 잠든 상태로 만듦 7)신호 핸들러가 등록되어 있지 않다면, 커널은 디폴트 행동 실행 **신호 핸들러가 실행되는 모드 : 사용자 모드, 신호 핸들러가 할당되는 메모리 영역 : 사용자 영역(커널 영역 아님)/ 신호에 대한 이슈** 신호 발생 시점과 신호 핸들러 실행 시점에 시간 차(수신 프로세스가 스케줄될 때만 신호 핸들러가 실행될 수 있기 때문) 신호 핸들러는 사용자 모드에서 실행(신호 핸들러는 사용자 영역에 존재, 신호핸들러가 커널 모드에서 실행되면, 사용자 코드에서 커널에 접근하는 위험 때문) 신호를 전달하고 신호 핸들러가 실행되게 하는 것은 커널에 의해 이루어짐. 신호가 수신 프로세스에 의해 거부된 경우(신호는 커널의 미처리 비트 벡터에 기록되어 미처리 상태로 존재하다가 프로세스가 신호 거부를 해제할 때처리) **파이프????????????????** 프로세스들이 메시지를 전송하기 위해 커널에 생성되는 메모리, 원형 큐와 유사하게 작동/ **익명의 파이프(anonymous pipe)**파이프를 생성한 부모와 자식 간에 메시지를 주고 받기 위한 목적, 한 컴퓨터 시스템에서만 유효 **이름을 가진 파이프(named pipe)**파이프의 이름을 아는 프로세스들끼리 메시지를 주고 받기 위한 목적, 네트워크로 연결된 여러 컴퓨터 시스템에서도 사용가능. 선입 선출방식으로 메시지 전송, 여러 응프에서 사용, Nagios, 쉘 **익명의 파이프 특징)** 파이프에 대한 이름없음. 파이프 양 끝 단에 송신 프로세스 1개와 수신 프로세스1개연결(송신 프로세스는 송신만, 수신 프로세스는 수신만 할 수 있는 단방향, 한 프로세스가 하나의 파이프에 대해 한 끝단에만 연결 가능) 선입선출 방식으로 작동, 하나의 컴퓨터 내에 있는 프로세스끼리만 통신 가능 **익명의 파이프에 대한 조작** 파이프 생성, pipe() 시스템 호출(int pipe(int fd[2]);, 파이프를 생성하고 fd[0]에는 파이프에서 수신용 파일 디스크립터를, fd[1]에는 파이프에 송신용 파일 디스크립터가 저장됨) 파이프 닫기, close()시스템 호출,파이프에서 읽고 쓰기, read()/write() 시스템 호출 **이름을 가진 파이프** 여러 프로세스들이 읽고 쓸 수 있는 특수 파일, FIFO라고 불려짐, 응용프로그램에서 입출력 방식은 파일 입출력 방식과 동일, FIFO는 생성된 후 삭제할 때까지 컴퓨터가 꺼진 상태에서도존재, 네트워크에 연결된 컴퓨터들사이에 통신가능 **이름을 가진 파이프 다루기** 생성 - 쉘에서 파일을 만들듯이 생성($ mknod 파이프이름, $ mkfifo 파이프이름,mkfifo() 시스템호출) 열기, open() 시스템 호출(FIFO의 이름을 알고, 이 이름으로 열기, 파일 열기와 동일), 닫기, close() 시스템 호출, 읽기 쓰기, read()/write() 시스템 호출