임베디드 플랫폼

< Oprating System >

1. Operating System의 기본개념

1) 가상머신

▶ 가상머신(Virtual Machine, VM)은 하나의 물리적인 컴퓨터에서 여러 개의 독립적인 가상 환경을 구축하여 실행할 수 있게 해주는 소프트웨어이다. 가상머신은 호스트 컴퓨터에서 동작하는 운영 체제(호스트 운영 체제) 위에 가상화된 하드웨어와 운영 체제를 에뮬레이션하고 실행하는 가상화 소프트웨어 레이어를 제공한다.

(1) 하드웨어 추상화: 가상머신은 호스트 시스템의 하드웨어를 추상화하여 게스트 운영 체제가 실제 하드웨어에 직접 액세스하지 않고도 실행될 수 있도록 한다. 이를 통해 서로 다른 운영 체제(Windows, Linux, macOS 등)를 동시에 실행하거나, 이전 버전의 운영 체제를 호환성 유지 목적으로 지원할 수 있다.

(2) 독립성: 각각의 가상머신은 격리된 환경에서 실행되므로 한 가상머신이 문제가 발생해도 다른 가상머신에는 영향을 주지 않는다. 이는 애플리케이션 개발 및 테스트, 시스템 관리 및 배치 등에서 매우 유용하다.

(3) 리소스 관리: 호스트 시스템 리소스(예: CPU, 메모리, 디스크 공간)를 여러 개의 가상머신 사이에서 분할하여 할당할 수 있다. 이렇게 함으로써 리소스 사용량을 최적화하고 효율성을 높일 수 있다.

(4) 이식성: 가상머신 이미지는 파일 형태로 저장되며, 이 파일을 다른 호스트 시스템으로 복사함으로썬 쉽게 마이그레이션(migration/환경-환경 이동과정)할 수 있다. 따라서 애플리케이션 및 서비스 배치와 같은 작업에서 유연성과 이식성을 제공한다.

2) 자원관리자

▶ 자원 관리자(Resource Manager)는 시스템의 자원(하드웨어 자원/소프트웨어 자원)을 효율적으로 할당하고 모니터링하는 역할을 수행하는 소프트웨어이다. 자원 관리자는 CPU, 메모리, 디스크 공간, 네트워크 대역폭 등과 같은 시스템 리소스를 관리하여 여러 프로세스 또는 작업이 동시에 실행될 수 있도록 한다.

▶ 자원 관리자의 주요 목표

(1) 자원 할당: 자원 관리자는 프로세스나 작업에 필요한 리소스를 할당한다. 이를 통해 각 프로세스가 충분한 CPU 시간, 메모리 공간 등을 얻어 실행할 수 있게 된

다.

(2) 우선순위 조정: 우선순위 기반 스케줄링 알고리즘을 사용하여 다양한 프로세스나 작업 사이에서 리소스 접근 및 실행 우선 순위를 결정한다. 이를 통해 중요한 작업이 우선적으로 처리되거나 실시간 요구 사항이 충족된다.

(3) 데드락 방지: 데드락은 여러 프로세스가 서로의 리소스가 CPU에 할당되기를 기다려서 상호적으로 블록되는 상태이다. 자원 관리자는 데드락 발생 가능성을 감지하고 방지하기 위해 적절한 알고리즘과 기법을 사용한다.

(4) 모니터링 및 성능 분석: 자원 관리자는 시스템의 상태와 성능에 대한 정보를 모니터링하고 수집한다. 이를 통해 시스템 병목 현상, 리소스 사용량 및 부하 등을 파악하여 최적화와 문제 해결에 활용할 수 있다.

3) OS의 분류(실시간 OS, 분산 OS 등)

▶ 실시간 운영 체제 (Real-Time Operating System, RTOS)는 실시간 시스템에서 사용되는 운영 체제이다. 실시간 시스템은 정확한 타이밍 요구 사항을 충족해야 하는 응용 프로그램을 위한 시스템으로, 이벤트 발생 후 정해진 시간 내에 반응하고 작업을 완료해야 한다. 실시간 운영 체제는 이러한 요구 사항을 충족하기 위해 설계되었다.

▶ 실시간 운영체제의 분류

(1) Hard Real-Time OS: 하드 리얼타임 운영 체제는 엄격한 타이밍 요구 사항을 가지고 있다. 특정 이벤트에 대한 응답 시간이 보장되어야 하며, 이를 위해 예약된 작업들 간의 실행 순서와 우선순위가 엄격하게 관리된다. 하드 리얼타임 시스템은 주로 항공우주, 자동차 제어, 의료 장비 등 안전과 생명에 직결된 분야에서 사용된다.

(2) Soft Real-Time OS: 소프트 리얼타임 운영 체제는 상대적으로 더 유연한 타이밍 요구 사항을 가지고 있다. 정확한 응답성은 중요하지만, 일부 작업의 지연이 허용될 수 있다. 소프트 리얼타임 시스템은 멀티미디어 애플리케이션, 게임 개발 등에서 주로 사용된다.

▶ 실시간 운영체제의 특징

(1) 실행 시간 예측이 가능하며 실시간성을 띄고 있다.

(2) 시스템의 응답 속도, 인터럽트 등에서 성능이 우수하다.

(3) 모듈화, 선점형 멀티 태스킹, 스케줄링, 통합 개발 환경 등을 지원한다.

(4) 시스템이 사용자가 원하는 한 가지 목적에 최적화 되어있다.(ex. 자동차 제어, 의료 장비 등)

(5) ROM에 프로세스와 커널을 담아 주소 공간을 공유한다.

▶ 분산 시스템

: 메모리나 클럭을 물리적으로 공유하지 않는 프로세서들의 집합이다. 각 프로세서들이 네트워크로 연결되어 있어 상호 협력 가능하다.

< 분산 시스템의 다이어그램 >

· 분산 시스템의 목적

(1) 자원 공유 : 네트워크로 연결된 다른 사이트(편의상 프로세서의 위치를 표현함)의 자원을 이용할 수 있다.

(2) 연산속도 향상 : 분할이 가능한 작업을 분산 시스템의 여러 사이트에 분산시켜서 동시에 병렬 처리하여 연산속도를 향상시킬 수 있다.

(3) 신뢰성 향상 : 일부 사이트에서 장애가 발생하더라도 전체 시스템의 동작이 멈추지 않는다. 또한 장애 발생 시, 장애 시스템의 기능을 다른 시스템으로 이동한 뒤 문제 해결 후 복귀 시킬 수 있다.

(4) 통신의 용이성 : 통신 네트워크로 연결된 사이트들의 사용자간 정보 교환 가능하다.

▶ 분산 운영 체제 (Distributed Operating System)는 여러 컴퓨터 또는 서버 간의 네트워크를 통해 연결된 컴퓨터 클러스터에서 동작하는 운영 체제이다. 분산 운영 체제는 다음과 같은 목적과 기능을 제공한다.

(1) 자원 공유: 분산 환경에서 여러 컴퓨터 간의 자원(메모리, 디스크 공간 등)을 공유할 수 있도록 한다.

(2) 네트워크 통신: 분산 환경에서 컴퓨터들 간의 효율적인 통신 메커니즘과 프로세스 간 메시지 전달 기능을 제공한다.

(3) 분산 스케줄링: 분산 환경에서 작업 및 자원 할당 스케줄링 알고리즘을 사용하여 여러 컴퓨터 클러스터 내의 작업 로드 밸런싱과 성능 최적화를 달성한다.

(4) 신뢰성 및 내결함성: 분산 환경에서 장애 복구와 오류 처리를 위한 메커니즘이 구현되어 안정성과 신뢰성을 보장한다.

(5) 데이터 이주 : 원격으로 데이터를 필요한 곳으로 전송하여 사용 가능하다.

(6) 계산 이주 : 대량의 데이터가 필요한 경우 원격으로 처리(계산) 후 결과를 받

을 수 있다.

(7) 프로세스 이주 : 프로세스를 이주시킴으로서 부하분산, 계산속도 향상 가능하다.

2. 프로세스 관리

1) 스레드와 프로세스

▶ 스레드(Thread)와 프로세스(Process)는 컴퓨터에서 실행되는 작업의 단위를 나타내는 개념이다. 이 두 용어는 다음과 같은 차이점을 가지고 있다.

(1) 프로세스 (Process)

- 프로세스는 실행 중인 프로그램 인스턴스를 의미한다.(프로그램을 실행하면 운영체제에서 메모리 공간을 할당 받아오며, 그 공간에 프로그램이 실행된다. 이 때 프로그램은 메모리에 정적인 상태로 저장되어 있으며, 프로세스는 실행을 위해 메모리에 올라와 있는 동적인 상태이다.)
- 각각의 프로세스는 독립된 메모리 공간, 자원 및 실행 흐름을 가진다.
- 운영 체제에 의해 관리되며, 프로세스 간에 데이터 공유 및 통신을 위해 별도의 매커니즘이 필요하다.
- 각각의 프로세스는 최소한 하나의 스레드(메인 스레드)를 가지며, 추가적인 스레드를 생성할 수 있다.

(2) 스레드 (Thread)

- 스레드는 하나의 프로세스 내에서 실행되는 독립적인 실행 흐름이다.
- 같은 프로세스 내에서 여러 개의 스레드가 동시에 작업할 수 있으며, 이들은 같은 메모리 공간과 자원을 공유한다.
- 스레드 간에 데이터 공유와 통신은 비교적 쉽고 빠르게 이루어진다. 하지만 적절한 동기화가 필요할 수 있다.
- 한 스레드가 예외 또는 오류 상황으로 인해 종료되면 다른 스레드들도 영향을 받게 된다.

▶ 일반적으로 멀티코어 시스템에서 여러 개의 스레드가 병렬로 실행될 수 있으므로, 멀티쓰레딩(Multithreading)은 다음과 같은 장점을 제공한다.

(1) 성능 향상: 여러 개의 스레드가 병렬적으로 작업하므로 전체 시간 소요를 줄일 수 있다.

(2) 응답성 향상: 사용자 인터페이스와 같이 실시간 응답이 필요한 애플리케이션에서 사용자 경험을 향상시킬 수 있다.

(3) 작업 분할: 큰 작업을 여러 개의 작은 작업으로 분할하여 처리하므로 코드 구

조와 유지보수성이 좋아진다.

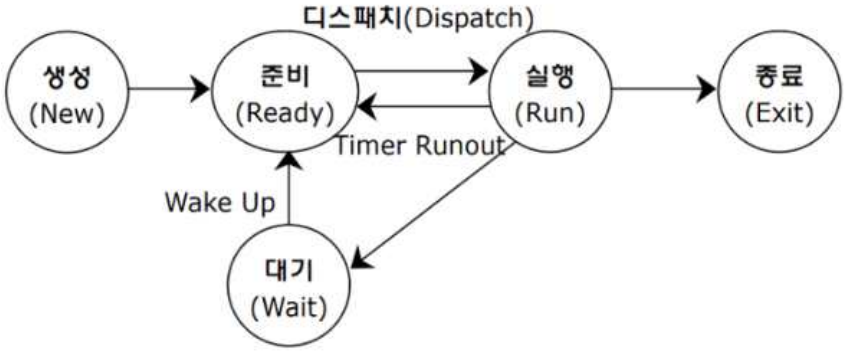
▶ 그러나 멀티쓰레딩에 따르는 문제점도 주목해야 한다.

(1) 동기화 문제: 여러 스레드가 공유 데이터에 접근하면서 발생하는 경합 조건(Race Condition), 교착 상태(Deadlock), 우선 순위 역전(Priority Inversion) 등과 같은 문제가 발생할 수 있다.

(2) 디버깅 및 추론 어려움: 멀티쓰레딩 환경에서 발생하는 버그 추적과 디버깅이 어렵고 복잡해진다.

2) 프로세스 상태

▶ 프로세스 상태(Process State)는 운영 체제에서 프로세스가 존재할 수 있는 다양한 실행 상태를 나타낸다.



< 프로세스 상태 다이어그램 >

(1) 생성 (Created): 프로세스가 생성되었지만 아직 실행되기 전인 초기 상태이다.

(2) 준비 (Ready): 프로세스 실행을 위해 CPU를 할당받기를 기다리는 상태이다. 모든 필요한 리소스(메모리, 파일 등)에 대한 접근 권한을 얻은 후에 준비 상태로 진입한다.

(3) 실행 (Running): CPU를 할당받아 현재 명령어를 실행 중인 상태이다. 한 번에 하나의 프로세서만이 실행 가능하며, 시분할 시스템에서는 일정 시간 후에 다른 프로세스와 교체될 수 있다.

(4) 대기 (Waiting 또는 Blocked): 프로세스가 어떤 이벤트(입출력 완료, 타이머 등) 발생을 기다리며 일시적으로 중단된 상태이다. 이벤트 발생 전까지 해당 프로세스는 CPU 할당을 받지 못한다.

(5) 종료 (Terminated): 프로세스의 실행이 완전히 종료되고 자원이 해제된 후의 최종상태이다.

3) 스케줄링 기초

▶ 스케줄링(Scheduling)은 운영 체제에서 여러 프로세스 또는 스레드 중에서 CPU 시간을 어떻게 할당할지 결정하는 작업이다. 스케줄링은 시스템의 성능, 응답성, 효율성 등을 최적화하기 위해 중요한 역할을 한다.

▶ 디스패쳐(Dispatcher) : CPU스케줄링 기능에 포함된 요소 중에 하나로서 CPU코어의 제어를 스케줄러가 선택한 프로세스에 주는 모듈이다. 디스패쳐는 프로세스간 문맥을 교환해주거나 사용자 모드로 전환, 프로그램 재시작을 위해 적절한 위치로 이동하는 일을 한다. 문맥 교환에 있어서 정지하고 다른 프로세스를 실행하는데 소요되는 시간을 디스패치 지연(Dispatch delay)라고 한다.

▶ 문맥 교환(Context Switching) : 운영 체제의 프로세스 스케줄링에서 중요한 개념이며, CPU가 한 작업(프로세스 또는 스레드)에서 다른 작업으로 전환하는 과정을 의미한다. 문맥 교환은 여러 상황에서 요구된다. 예를 들어, 현재 실행 중인 프로세스가 I/O 작업을 기다리는 대기상태에 있을 때 CPU는 다른 작업을 수행해야 하므로 문맥 교환이 발생해야 한다. 또한 시분할 시스템에서 CPU 사용 시간을 공정하게 분배하기 위해 주기적으로 프로세스 간 문맥 교환이 발생한다.

하지만 문맥 교환이 자주 발생하면 오버헤드(어떤 작업을 수행하기 위해 필요한 추가적인 자원 또는 시간)를 발생시켜서 실제 유용한 작업이 수행되지 않고 시간과 자원이 많이 소비되므로 CPU성능에 부정적인 영향을 줄 수 있다.

▶ 문맥 교환의 주요 단계

(1) 문맥 저장: 현재 실행 중인 프로세스의 상태(즉, 문맥)를 저장한다. 이에는 CPU 레지스터 값, 프로그램 카운터 등이 포함된다.

(2) 상태 변경: 운영 체제는 해당 프로세스의 상태를 '대기'나 '준비' 등으로 변경한다.

(3) 다음 프로세스 선택: 스케줄러가 다음에 실행할 프로세스를 결정한다.

(4) 문맥 복원: 새롭게 선택된 프로세스의 저장된 문맥을 복원하고 실행을 계속한다.

▶ CPU Burst : 프로세스가 CPU에서 실행되는 시간을 의미한다. 이 기간 동안 프로세스는 계산과 같은 CPU 집중적인 작업을 수행한다.

▶ I/O Burst : 프로세스가 I/O작업을 수행하는 시간이다. 이 기간 동안, 프로세스는 대부분의 시간을 I/O장치가 요청을 완료하는 것을 기다리는데 보내므로, CPU는 다른 작업에 사용될 수 있다.

▶ CPU-I/O Burst Cycle

: 프로세스의 실행 패턴을 설명하는데 사용되는 개념으로서 프로세스가 CPU Burst와 I/O Burst 사이를 번갈아 가며 반복하는 패턴을 나타낸다. 프로세스의 생

명 주기 동안, 일반적으로 여러 번의 CPU Burst와 I/O Burst가 번갈아 가며 발생하는데 이를 "CPU-I/O Burst Cycle"이라고 한다. 이 개념은 운영 체제에서 스케줄링 알고리즘 설계에 중요한 역할을 한다. 예를 들어, 짧은 CPU Burst를 가진 프로세스를 먼저 스케줄링하여 전체적인 응답 시간(response time)을 향상시키려는 최소 잔여 시간 우선(SJF: Shortest Job First)알고리즘이나 최단 컴퓨팅 타임 우선(SRTF: Shortest Remaining Time First)알고리즘에 CPU Burst가 필요하다.

▶ 프로세스 제어 블록(Process Control Block, PCB)

: PCB는 운영 체제에서 각 프로세스에 대한 중요한 정보를 저장하는 데이터 구조이다. 운영 체제는 프로세스를 관리하고 스케줄링 하기 위해 PCB를 사용한다. 특히 PCB는 문맥 교환이 일어날 때 프로세스의 상태를 저장하기 위해 중요하다. PCB는 일반적으로 다음과 같은 정보를 포함한다.

(1) 프로세스 식별자(Process ID): 각 프로세스는 고유한 ID를 가진다.

(2) 프로세스 상태(Process State): 프로세스의 현재 상태(대기, 실행, 준비 등)을 나타낸다.

(3) 프로그램 카운터(Program Counter): 다음에 실행될 명령어의 주소를 저장한다.

(4) CPU 레지스터들(CPU Registers): 이들은 프로그램 카운터와 함께 프로세서 상태를 나타내며, 명령어 실행 결과나 주소 등을 저장하는 데 사용된다.

(5) CPU 스케줄링 정보(CPU Scheduling Information): 이것은 프로세스의 우선 순위, 스케줄링 큐에 대한 포인터 등을 포함할 수 있다.

(6) 메모리 관리 정보(Memory Management Information): 각각의 프로세스가 차지하는 메모리 영역에 대한 정보이다.

(7) 입출력 상태 정보(I/O Status Information): 열린 파일 목록, I/O 장치들 등과 같은 입출력 관련된 자원들에 대한 정보이다.

▶ 선점 스케줄링(Preemptive Scheduling)

: 선점 스케줄링에서 운영 체제는 현재 실행 중인 프로세스를 중단하고 다른 프로세스에게 CPU를 할당할 수 있다. 이 방식은 시스템의 반응성을 높이며, 공정한 CPU 사용 시간 분배를 가능하게 한다.

선점 스케줄링의 예로는 Round Robin, Priority Scheduling, Shortest Remaining Time First 알고리즘 등이 있다.

하지만 선점 스케줄링 방식은 문맥 교환 오버헤드가 발생한다는 단점이 있다. 즉, 프로세스 간 전환시에 상태 정보를 저장하고 복원하는 데 추가적인 시간과 자원이 소모된다.

▶ 비선점 스케줄링(Non-preemptive Scheduling)

: 비선점 스케줄링에서는 한 번 CPU가 할당되면 프로세스는 작업을 완료하거나 자발적으로 제어를 반환하기 전까지 CPU를 계속 사용한다. 이 방식은 문맥 교환 오버헤드가 발생하지 않지만, 장기 실행 작업 때문에 다른 작업들이 긴 대기 시간을 가질 수 있다는 단점이 있다.

비선점형 알고리즘의 예로는 First Come First Serve(FCFS), Shortest Job Next(SJN) 알고리즘 등이 있다.

▶ 평균 대기 시간(Average Waiting Time)

: 모든 프로세스가 CPU를 기다리는 시간의 평균을 의미한다. 프로세스가 준비 상태에서 실행 상태로 전환되기까지의 시간을 대기시간이라고 하며 이러한 대기 시간들을 모든 프로세스에 대해 합산하고, 프로세스의 총 개수로 나누어 평균 대기 시간을 계산한다.

평균 대기시간은 운영 체제의 스케줄링 알고리즘 성능을 평가하는 데 중요한 지표 중 하나이다. 일반적으로 이 값을 최소화하는 것이 목표이다.

▶ 전체 처리량(Throughput)

: 일정 시간 동안 시스템이 완료할 수 있는 작업의 양을 나타내는 용어이다. 컴퓨터 시스템에서 이는 주로 일정 시간 내에 완료된 프로세스의 수, 처리된 요청의 수, 전송된 데이터의 양 등으로 측정된다.

3. CPU 스케줄링

1) 단일프로세서 스케줄링 기법

▶ 단일 프로세서 스케줄링은 하나의 CPU(프로세서)를 가지고 있는 시스템에서 여러 프로세스 또는 스레드 중에서 CPU 시간을 어떻게 할당할지 결정하는 기법이다. 다양한 스케줄링 알고리즘이 있으며, 각각의 알고리즘은 다른 목적과 특성을 가지고 있다.

(1) 선입선출 (First-Come, First-Served, FCFS): 가장 간단한 스케줄링 알고리즘으로, 도착한 순서대로 프로세스를 실행한다. 선입선출 알고리즘은 비선점형(Non-preemptive) 방식으로 동작하며, 긴 작업이 먼저 도착하면 평균 대기 시간이 길어질 수 있다.

< FCFS 스케줄링 간트 차트 예시 1 >

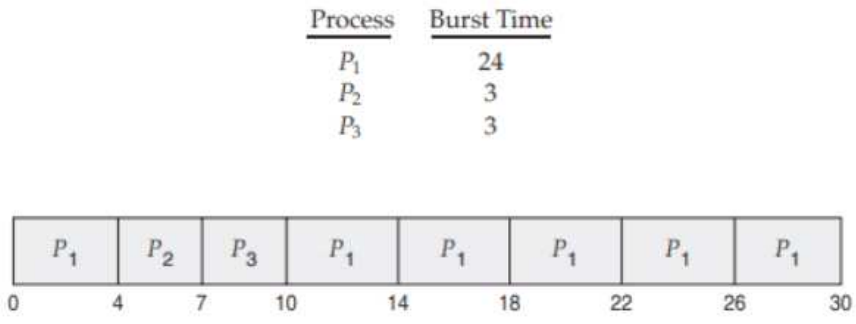
< FCFS 스케줄링 간트 차트 예시 2 >

→ 예시 1의 경우 P1의 Burst Time이 다른 프로세스들에 비해 매우 길어서 P2는 24, P3는 27의 시간을 기다려야 한다. P1~P3의 평균 대기 시간은 (0+24+27)/3=17이 된다.

→ 예시 2의 경우 Burst Time이 짧은 P2, P3를 먼저 실행하였다. 이 때 평균 대기 시간은 (0+3+6)/3=3으로 현저히 줄어들었다.

→ 예시 1,2를 통해 FCFS는 평균 대기시간을 최소화하지 않는다는 것과 프로세스의 Burst Time에 따라 성능이 극명하게 변한다는 것을 알 수 있다.

(2) 라운드 로빈 (Round Robin): 시분할 시스템을 위해 설계된 방법으로서 일정한 시간 할당량(Time Slice or Time Quantum)을 갖고 여러 프로세스를 번갈아 가며 실행하는 선점형 방식이다. 각 프로세스는 주어진 타임 슬라이스만큼 CPU를 사용하고 나머지는 준비 상태로 이동한다. 라운드 로빈 알고리즘은 선입선출보다 공정성과 응답성 측면에서 우수하지만, 타임 슬라이스 크기에 따라 성능 차이가 발생할 수 있다.



< Round Robin을 적용한 간트 차트 예시 >

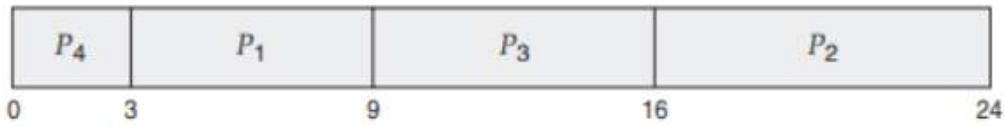
→ 위의 예시에서 볼 수 있듯이 Round Robin은 일정 타임 슬라이스만큼만 프로세스를 실행한다. 프로세스의 Burst Time이 타임 슬라이스보다 작다면 Burst Time만큼만 실행된다.

→ 타임 슬라이스가 매우 크다면 FCFS와 같이 작용하게 되며, 타임 슬라이스가 작다면 문맥 교환이 많이 발생되므로 오버헤드가 증가한다.

(3) 우선순위 (Priority Scheduling): 각 프로세스에 우선순위 값을 부여하고 가장 높은 우선순위를 가진 프로세스부터 실행하는 방식이다. 우선순위가 동일한 경우에는 FCFS를 적용한다. 우선순위는 정적인 방법(사용자 지정) 또는 동적인 방법(작업의 중요도나 실행 상태에 따른 변동)으로 결정된다.

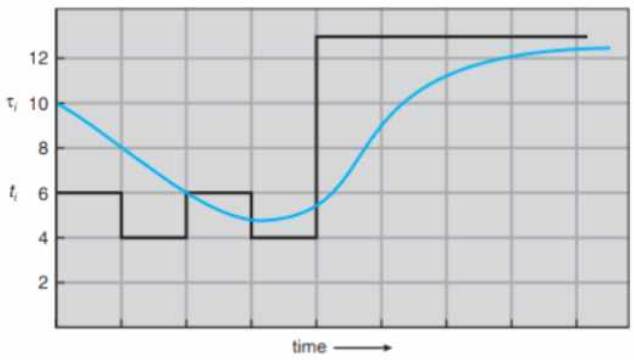
선점형, 비선점형이 모두 가능하며 우선순위 스케줄링의 가장 큰 문제점은 Infinite Blocking(Starvation)이다. 예를 들어, CPU할당을 기다리는 P1이 우선순위가 낮다면, 우선순위가 높은 프로세스들이 계속 들어와서 P1은 CPU에 할당될 수 없게 된다. 이를 해결하기 위한 방법으로는 Aging이 있다. 이는 시간이 지날수록 우선도 값을 변화시켜 우선순위를 증가시기는 방법이다.

(4) SJF (Shortest Job First): 예상 실행 시간이 가장 짧은 작업부터 배치하여 처리하는 방식으로 선점, 비선점 스케줄링이 모두 가능한 알고리즘이다.(짧은 Burst Time을 갖는 프로세스를 우선순위에 두는 우선순위 알고리즘이기도 하다.) 선점형인 경우는 준비 큐(Queue)에서 Burst Time이 짧은 순으로 배치 되어있지 않은 경우에 사용하며 SRTF라고 부른다. 비선점형인 경우 준비 큐에서 Burst Time이 이미 짧은 순으로 배치 되어있는 경우이며 SJF라고 한다. SJF 알고리즘은 최소 평균 대기 시간을 제공하지만 작업의 예상 실행 시간을 정확히 예측해야 하므로 실제 환경에서 적용하기 어렵다.



< SJF 스케줄링 간트 차트 예시 >

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

< SJF의 CPU Burst 길이 예측 >

→ 표의 간트 차트 예시에서 SJF의 평균 대기시간은 (0+3+9+16)/4=7이다. SJF는 최소의 평균 대기시간을 가지는 최적의 알고리즘이다. 하지만 프로세스의 Burst Time 길이를 알고 있어야 하기 때문에 Burst Time을 알 수 없는 경우 이를 예측하는 알고리즘을 실행하는 시간이 필요하다.

→ CPU Burst 길이를 예측하는 방법은 일반적으로 CPU Burst 길이들의 지수평균(Exponential Average)를 통해 예측된다.

(5) SRTF (Shortest Remaining Time First): 현재 남아 있는 작업 중에서 예상 실행 시간이 가장 짧은 작업부터 처리하는 방식이다. SRTF 알고리즘은 SJF의 변형으로 선행된 작업보다 짧게 남아있다면 해당 작업을 선행하여 처리한다.(SJF의 선점형 방식이다.)

(6) HRRN (Highest Response Ratio Next): 대기 중인 모든 작업들에 대해 응답률(Response Ratio)을 계산하여 최대 응답률을 갖는 작업부터 처리하는 방식이다.

2) 멀티프로세서 스케줄링 기법

▶ 멀티프로세서 스케줄링은 여러 개의 CPU(프로세서)를 가지고 있는 시스템에서 여러 프로세스 또는 스레드 중에서 CPU 시간을 어떻게 할당할지 결정하는 기법이

다. 멀티프로세서 시스템은 병렬 처리를 통해 성능을 향상시킬 수 있으나, 복잡도가 상승한다. 이를 위해 다양한 스케줄링 알고리즘이 사용된다.

(1) 비대칭 멀티프로세싱(Asymmetric Multiprocessing/ASMP): 하나의 master프로세서가 다른 프로세서들을 관리/감독하는 형태이다. master프로세서는 모든 스케줄링 결정과 입풀력 처리, 로드 밸런싱을 다 결정한다. 이 때 다른 프로세서들은 User 코드만 수행한다. 이 방식은 한 프로세서가 다 처리하기 때문에 공유자원이 존재하지 않고 프로세서 간의 이동(migration)이 필요없다.

(2) 대칭 멀티프로세싱(Symmetric Multiprocessing/SMP): 대부분의 멀티프로세서가 사용하는 방식이다. 각각의 프로세서가 동등한 권한으로 프로세싱하는 형태이며 각자 스케줄링을 진행한다. 프로세스들은 하나의(공동의) 준비큐를 가지거나 각 프로세서 마다 private 준비큐에 로드되어 있어며, 모든 프로세서에는 각각의 스케줄러가 있어야 한다. 따라서 동시에 여러 프로세서에서 공유자원에 접근할 시 동기화 문제가 발생할 수 있다.

3) 실시간 스케줄링 기법

(1) 하드 리얼타임 (Hard Real-Time): 작업이 정확한 시간 내에 반드시 완료되어야 하는 경우를 다룬다. 이러한 작업들은 장애나 지연으로 인해 데드라인을 놓칠 수 없으며, 놓침으로 인해 심각한 결과가 발생할 수 있다.

(2) 소프트 리얼타임 (Soft Real-Time): 작업이 일반적으로 정확한 시간 내에 완료되어야 하지만 가끔 예외적인 상황에서 데드라인을 초과할 수 있는 경우를 다룬다. 소프트 리얼타임 작업의 경우 일부 지연이 허용될 수 있지만, 너무 많은 지연이 발생하면 성능 저하나 비정상적인 동작을 초래할 수 있다.

4. 병행성 제어

1) 상호배제

▶ 상호배제(Mutual Exclusion: Mutex)는 여러 개의 프로세스나 스레드가 하나의 공유된 자원에 동시에 접근하지 못하도록 하는 동기화 개체이다. 상호배제는 한 번에 하나의 스레드나 프로세스만이 특정 코드 섹션(일반적으로 공유 자원에 대한 접근)을 실행할 수 있도록 한다.

▶ 상호배제 매커니즘

(1) 임계 영역 (Critical Section): 공유 자원에 접근하는 코드 영역을 임계 영역이라고 한다. 임계 영역은 상호배제가 필요한 부분으로, 한 번에 하나의 프로세스 또

는 스레드만이 해당 영역에 진입할 수 있다.

(2) 락 (Lock): 락은 상호배제를 구현하기 위한 동기화 기법 중 하나이다. 락은 임계 영역 진입 전에 획득되어 다른 프로세스나 스레드가 해당 임계 영역에 진입하지 못하도록 막는다. 락을 획득한 후 작업이 완료되면 반드시 락을 해제해야 다른 프로세스나 스레드가 해당 임계 영역에 접근할 수 있게 된다. 공유 자원이 사용되고 있을 경우에 Lock변수의 값은 1이며, 공유 자원이 비어있을 경우에 Lock 변수의 값은 0으로 나타낸다(세마포어와 반대).

(3) 동기화 기법(Synchronization) : 두 개 이상의 프로세스를 한 시점에서 자원 접근을 동시에 처리하게 할 수 없게 하기 위해 각 프로세스에 대한 자원 접근 처리 순서를 결정하는 것으로 상호 배제의 한 형태이다. 대표적인 동기화 기법으로는 세마포어와 모니터가 있다.

2) 세마포어, 모니터

- 세마포어(Semaphore)

: 세마포어는 다중 프로세싱 환경에서 동기화 문제를 해결하는 데 사용되는 중요한 도구이다. 이는 단일 혹은 다수의 공유 자원에 여러 프로세스 또는 스레드에 대한 동시 접근을 제어하고, 임계 영역에 대한 상호 배제를 보장하는 역할을 한다.

▶ 세마포어 변수(S) : 공유된 자원에 접근하려고 하는 두 개 이상의 프로세스에 대해 부여되는 변수이다. 0/1 혹은 0/양수 값을 가지며 P연산, V연산에 의해서만 접근 가능하다. 여기서 1 또는 양수는 공유된 자원의 수를 의미한다. S의 값이 0이면 공유 자원에 접근 할 수 없다는 것을 의미한다.

▶ 세마포어는 정수 값과 함께 작동하는 변수로 볼 수 있으며, 기본적으로 두 가지 주요 연산, 즉 wait (또는 P 연산)과 signal (또는 V 연산)을 지원한다.

(1) Wait (P) 연산: 세마포어의 값이 0보다 크면 값을 감소시키고, 그렇지 않으면 (세마포어의 값이 0이면) 프로세스를 대기 상태로 만든다. 이것은 일반적으로 임계 영역에 진입하기 전에 수행된다.

→ 사용할 수 있는 공유 자원 있으면 세마포어 값을 –1 해준 뒤 사용하고, 사용할 수 있는 공유 자원이 없으면 대기 하도록 하는 함수이다.

(2) Signal (V) 연산: 세마포어의 값을 증가시키고, 만약 어떤 프로세스가 대기 중이라면 하나를 깨운다. 이것은 일반적으로 임계 영역에서 나온 후에 수행된다.

→ 공유 자원을 다 사용한 후에 세마포어 값을 +1 해주고, 대기하고 있는 프로세스나 스레드에게 공유 자원을 사용할 수 있다고 알려주는 함수이다.

▶ 세마포어(Semaphore)와 뮤텍스(Mutex)의 차이점

(1) 사용 방식: 세마포어는 일반적으로 여러 개의 프로세스 또는 스레드가 공유 리

소스에 접근하는 것을 제어하는 데 사용된다. 반면에, 뮤텍스는 오직 하나의 프로세스 또는 스레드만이 특정 코드 섹션(일반적으로 공유 리소스에 대한 접근)을 실행할 수 있도록 한다.
(2) 리소스 개수: 세마포어는 정수 값을 가지며, 이 값은 한 번에 자원에 접근할 수 있는 프로세스 또는 스레드의 수를 나타낸다. 따라서 여러 개의 프로세스/스레드가 동시에 리소스를 사용할 수 있다. 반면에, 뮤텍스는 단일 리소스를 보호하기 위해 사용되므로, 잠긴 상태와 해제된 상태만을 갖는다.
(3) 소유권: Mutex는 스레드가 소유권을 가진다. 즉, Mutex를 잠그고 해제하는 작업은 리소스를 사용중인 스레드에서 이루어져야 한다. 다른 스레드에서 Mutex를 해제하거나 잠그면 예기치 않은 동작이 발생할 수 있다. 반면에 세마포어에서는 서로 다른 프로세서나 스레드가 'wait'와 'signal' 연산을 수행할 수 있으므로 소유권 개념이 없다.
(4) 대기 상태 관리: 세마포어에서 대기(waiting) 상태인 프로세스/스레드들은 시그널(signal) 연산으로 인해 깨워진다.
(5) 복잡성: 일반적으로 세마포어보다 뮤텍스가 구현상 간단하다. 그 이유는 세마포어가 카운팅 기능과 함께 여러 개의 waiters 관리 등 추가 기능을 제공하기 때문이다.
→ 차이점 요약: 뮤텍스는 상호 배제를 위한 도구로, 한 번에 하나의 스레드만이 자원에 접근할 수 있도록 한다. 반면에 세마포어는 동시에 접근할 수 있는 스레드의 최대 수를 제어하는 방식으로 작동한다. 뮤텍스는 스레드가 소유권을 가져서 뮤텍스를 잠그고 해제하고 잠그는 권한을 가지고 있고, 세마포어는 소유권의 개념이 없기 때문에 세마포어를 소유하고 있지 않은 스레드도 세마포어를 해제할 수 있다.

- 모니터(Monitor)
: 세마포어 기법에서 타이밍 문제가 발생하는 것을 보완하기 위해 개발되었다. 세마포어의 P연산과 V연산을 내부에서 동기화를 관리해주어 사용방법이 간단한 기법이다.
▶ 모니터 기법의 구성
· 공유 데이터
· 공유 데이터를 다루는 연산들(procedure)
· 초기화 코드
▶ 모니터 기법의 특징
· 객체 지향 언어(C++, Java)의 클래스와 같은 추상 데이터 타입이다.
· 한 번에 하나의 프로세스만 모니터 개체 안의 프로시저를 사용할 수 있다. 모니

터에 접근하지 못한 프로세스는 Entry Queue에서 대기한다.(상호배제)

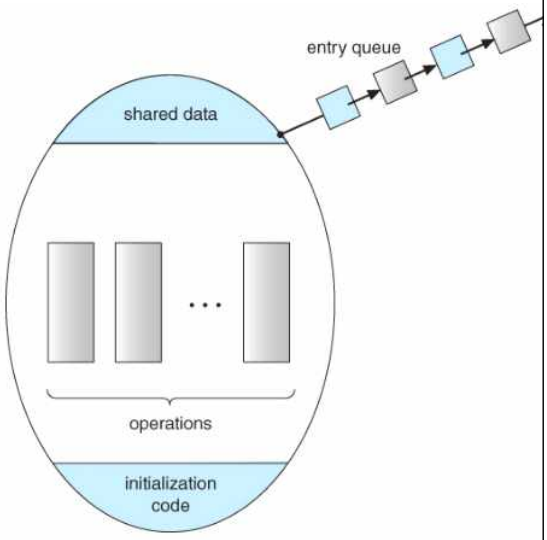
· 모니터 개체 내 공유 자원은 직접 접근할 수 없으며 프로시저를 통해서만 가능하다.(정보 은닉)

· 프로그래머는 동기화에 대한 코딩을 할 필요가 없다.

· 세마포어와 같이 조건 변수로 wait와 signal을 사용한다.

```
monitor monitor_name
{
  // shared variable declarations

  procedure P1 (...) {
    ...
  }

  procedure P2 (...) {
    ...
  }

  ...

  procedure Pn (...) {
    ...
  }

  initialization code (...) {
    ...
  }
}
```

< 모니터 기법 코드 구성 >



< 모니터 기법 기본 구조 >

3) 교착상태(Deadlock)

▶ 교착상태(Deadlock)란, 여러 개의 프로세스나 스레드가 서로가 가지고 있는 자원을 점유하며 대기하고 있어서 아무런 진전이 없는 상태를 말한다. 각 프로세스 또는 스레드는 다른 프로세스나 스레드가 점유한 자원을 기다리면서 자신이 가진 자원을 계속해서 점유하고 있다. 이러한 상태에서는 모든 프로세스나 스레드가 움직일 수 없게 되어 시스템이 정지된다.

▶ 교착상태 발생조건

(1) 상호배제 (Mutual Exclusion): 최소한 하나의 자원은 동시에 한 개의 프로세스만이 사용할 수 있어야 한다.

(2) 점유와 대기 (Hold and Wait): 최소한 하나의 자원을 점유한 상태에서 다른

프로세스에 의해 요구되는 추가적인 자원을 대기한다.

(3) 비선점 (No Preemption): 다른 프로세스가 이미 점유한 자원은 강제적으로 빼앗아올 수 없다.

(4) 순환 대기 (Circular Wait): 각 프로세스들 사이에 순환적으로 연결된 대기 관계가 형성되어야 한다.

4) 교착상태 대처방법

(1) 예방 (Prevention): 교착상태 발생 조건 중 하나 이상을 제거하여 교착상태를 예방하는 방법이다.

- 상호배제 부정: 여러 개의 프로세스가 공유자원에 동시에 접근할 수 있도록 허용한다.
- 비선점 허용: 우선순위가 높은 작업이 우선순위 낮은 작업의 자원을 강제적으로 뺏어올 수 있도록 한다.
- 순환대기 제거: 모든 자원 요청에 번호를 할당하고, 번호 순서대로만 요청 가능하도록 제약한다.

(2) 회피 (Avoidance): 교착상태가 발생할 가능성을 배제하지 않고 교착상태가 발생하면 적절히 피하는 방법이다.

- 은행원 알고리즘(Banker's Algorithm): 교착상태에 빠질 가능성이 있는지 판단하기 위한 알고리즘으로 '안전상태(safe state)'와 '불안전상태(unsafe state)'로 상태를 판단하여 안정상태를 유지할 수 있는 요구만 수락하고 불안전상태를 초래할 수 있는 요구는 운영체제에서 요구를 만족해줄 수 있는 상태가 될 때까지 거절한다.
- 안전상태(Safe State): 시스템이 교착상태를 일으키지 않으면서 각 프로세스가 요구한 최대 요구량만큼 필요한 자원을 할당해 줄 수 있는 상태를 말한다.
- 안전 순서열: 시스템이 프로세스의 요구들을 모두 분석한 뒤, 교착상태가 발생하지 않도록 가능한 모든 안전상태를 나열한 것이다.
- 불안전상태(Unsafe State): 안전순서열이 존재하지 않는 상태를 말한다. 불안전상태는 교착상태이기 위한 필요조건이다. 하지만 불안전상태라고 해서 무조건 교착상태가 발생하는 것이 아니라 불안전상태의 요청이 수락되었을 때 발생하는 것이다.

(3) 탐지 및 회복 (Detection and Recovery): 교착상태가 발생 할 수 있도록 허용한 뒤 교착상태가 발생할 경우 찾아내어 고치는 방법이다.

- 탐지(Detection): 시스템에서 교착 상황을 주기적으로 탐지하는 알고리즘과 메커니즘을 구현하여 교착 상황 타임아웃 등의 기준으로 패턴 분석 등을 통해 탐

지한다. 이 때, 교착상태를 발견하기 위해 알고리즘과 자원 할당 그래프를 사용하는데, 자원을 요청할 때마다 탐지 알고리즘을 사용하게 되므로 오버헤드가 발생한다.

· 회복(Recovery): 교착된 상황일 때 일부 혹은 전체 리소스 해제, 재할당, 프로세스 종료, 자원 선점 등의 방식으로 회복시킨다.

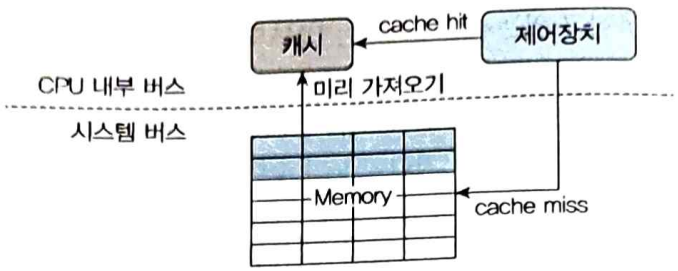
(4) 무시 (Ignorance): 일부 시간 제약 내에서 재부팅 등 시스템 리셋 및 초기화 등 단계에서 모든 리소스 할당과 정리를 실행하여 회복시키는 방식이다.

5. 메모리 관리 방법

1) 캐시메모리

▶ 캐시(cache) : 메모리와 CPU간의 속도 차이를 완화하기 위해 메모리의 데이터를 미리 가져와(prefetch) 저장해두는 임시 장소이다. 캐시는 CPU 안에 있으며, CPU 내부 버스의 속도로 작동하고, 메모리는 시스템 버스의 속도로 작동하기 때문에 캐시가 메모리보다 빠르다.

▶ 캐시의 구조



[그림 2-10] 캐시 메모리의 동작

: CPU는 메모리에 접근하기 전 캐시를 먼저 방문하여 원하는 데이터가 있는지 찾아본다. 캐시에서 원하는 데이터를 찾으면 캐시 히트(cache hit)라고 하며, 그 데이터를 캐시에서 가져와서 바로 사용한다. 그러나 원하는 데이터가 캐시에 없으면 메모리로 가서 데이터를 찾는데, 이를 캐시 미스(cache miss)라고 한다. 캐시 히트가 되는 비율을 캐시 적중률(cache hit ratio)라고 한다.

▶ 캐시의 적중률 증가 방법

(1) 캐시의 용량 증가 : 캐시의 용량을 키우면 더 많은 데이터가 들어와서

캐시 히트 확률이 높아진다.

(2) 앞으로 많이 사용될 데이터 가져오기 : 현재 위치에 가까운 데이터가 멀리 있는 데이터보다 사용될 확률이 더 높다는 지역성 이론에 따라 데이터를 미리 가져온다.

▶ 메모리 기록 방식

(1) 직접 기록 : 한 워드가 메인 메모리와 캐시 메모리 양쪽에 모두 기록되는 방식으로 보통 양쪽의 내용이 일치한다. 기록하는 속도는 느리지만 고장에 대한 데이터 기록 오류가 없으며, 캐시 미스 때문에 메모리를 치환할 때 데이터 복구가 필요없다. 넓은 어드레스 공간을 여기저기 액세스 하는 경우 유리하다.

(2) 기록 복귀 : 캐시 메모리가 변하더라도 메모리는 갱신되지 않고 대시 블록이 캐시 메모리로부터 제거될 때, 메모리를 갱신하는 방법이다. 메인 메모리의 트래픽 측면에서 직접 기록 방식보다 우수한 성능을 가지고 캐시 메모리에만 동작을 하므로 고속으로 행할 수 있다. 하지만 논리적인 복잡성이 따르고, 캐시 메모리와 메인 메모리의 내용이 항상 일치하지 않아 주의가 필요하다. 좁은 어드레스 공간을 빈번하게 액세스 하는 경우에 유리하다.

(3) 기록 사이클의 캐시 무효화 : 워드는 직접 기록시키지만 캐시는 갱신시키지 않는 방법이다. 여러 프로세서가 동일한 메모리 위치에 대한 복사본을 갖고 있는 상황에서, 한 프로세서가 해당 위치를 수정하면 다른 모든 캐시의 복사본이 유효하지 않게된다. 이러한 경우 해당 메모리에 해당하는 모든 캐시 항목을 무효화 한다. 따라서 프로세서는 워드를 메인 메모리로부터 호출하는 것이 된다. 그 캐시 무효화를 플러싱(flushinh) 또는 퍼지(purge)라고 한다.

(4) 기록 버퍼 : 직접 기록 방식의 변형으로 메인 메모리가 기록 버퍼를 통해서 기록되는 방법이다. 버퍼에 데이터를 기록한 후 메인 메모리에 저장하면, 디스크와 같은 슬로우 I/O장치와 같이 액세스가 오래걸리는 경우에 유리하다.

▶ 메모리 사상 방식

: 메인 메모리에서 데이터 참조는 주소로 이루어지고, 캐시 메모리를 사용하기 위해서는 메인 메모리의 주소를 캐시 메모리의 워드에 사상시켜야 한다.

(1) 직접 사상 방식(direct mapping) : 가장 간단한 방법으로 메인 메모리의 블록을 캐시 메모리의 블록 프레임에 직접 사상시키는 방법이다. CPU가

메모리의 참조를 요청하면, 인덱스 필드(태그 필드 or 오프셋 필드 : 데이터 베이스의 특정 테이블의 column 데이터)가 캐시 메모리를 호출하기 위한 번지로 변환할 때 쓰인다. CPU 번지의 태그 필드와 캐시로부터 읽힌 워드의 태그 필드를 비교해서 원하는 데이터가 캐시에 있음을 나타내는 것이다. 즉, 메인 메모리를 블록으로 분할하여 각각의 블록과 캐시 메모리와의 대응 관계가 고정되어 있다. 이 방식은 데이터와 태그에 대한 동시 호출이 허용되고 하드웨어가 간단하다는 장점이 있지만, 매핑의 자유도가 낮아 캐시 메모리 접근에 대한 실패율이 높다.

(2) 완전 연관(연상) 사상 방식(fully associative mapping) : 메인 메모리를 블록으로 분할할 때, 모든 캐시 메모리로부터 어느 블록이나 액세스할 수 있게 되어 있다. 따라서 매핑의 자유도는 높아지지만 시스템의 가격이 올라간다. 메모리 워드의 번지와 데이터를 캐시 메모리의 블록 프레임 내에도 동시에 저장하는 방법이며, 높은 블록 경쟁을 제거할 수 있으나 호출 시간이 길어진다.

(3) 세트 연관(집합-연상) 사상 방식(set-associative mapping) : 직접 사상 방식과 완전 연관 사상 방식을 결합한 방법으로 캐시 메모리의 각 워드를 같은 인덱스 번지 아래서 두 개 이상의 메모리 워드를 저장할 수 있도록 함으로써 직접 사상의 단점을 보완한 방법이다.

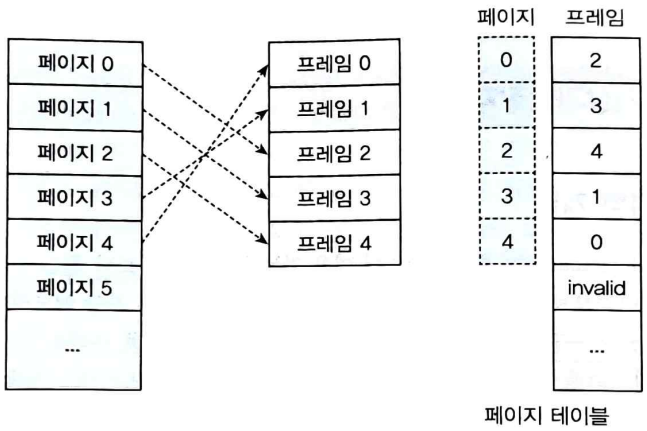
2) 가상메모리

▶ 가상 메모리(virtual memory) : 주기억장치의 이용 가능한 기억 공간보다 훨씬 큰 주소를 지정할 수 있도록 한 개념이다. 가상 메모리를 사용하면 사용자는 실제 주소 공간의 크기에 구애받지 않고 보다 큰 가상 주소 공간에서 프로그래밍을 할 수 있을 뿐만 아니라, 주기억장치보다 크기가 큰 프로세스를 수행시킬 수 있다.

▶ 가상 메모리의 메모리 분할 방식 : 실제 메모리에 있는 물리 주소 0번지는 운영체제 영역으로 일반 프로세스가 사용할 수 없다. 따라서 가상 메모리 시스템에서는 운영체제를 제외한 나머지 메모리 영역을 일정한 크기로 나눠 일반 프로세스에 할당한다. 메모리 분할 방식은 크게 가변 분할 방식(세그멘테이션)과 고정 분할 방식(페이징)으로 나뉜다.

3) 페이징과 세그멘테이션

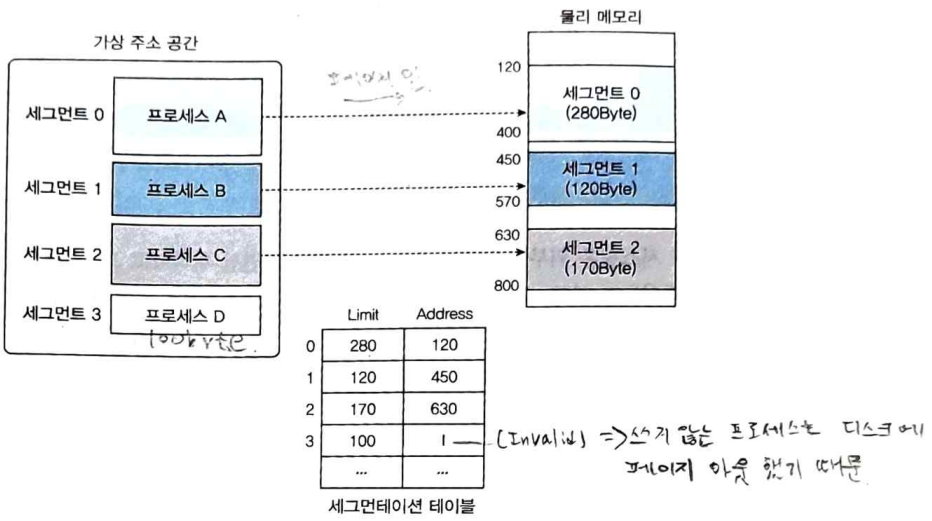
▶ 페이징 기법의 구현



[그림 2-11] 페이징 기법의 구현

: 페이징 기법은 고정 분할 방식을 이용한 가상 메모리 관리 기법으로, 물리 주소 공간을 같은 크기로 나눠 사용한다. 가상 주소의 분할된 각 영역은 페이지라고 부르며 번호를 지정해 관리한다. 물리 메모리의 각 영역은 가상 주소의 페이지와 구분하기 위해 프레임(frame)이라고 부른다. 페이지와 프레임의 크기는 같으며, 물리적 메모리에서 디스크로 페이지를 보내는 것을 페이지 아웃(스왑), 그 반대를 페이지 인이라고 한다.

▶ 세그멘테이션 기법의 구현

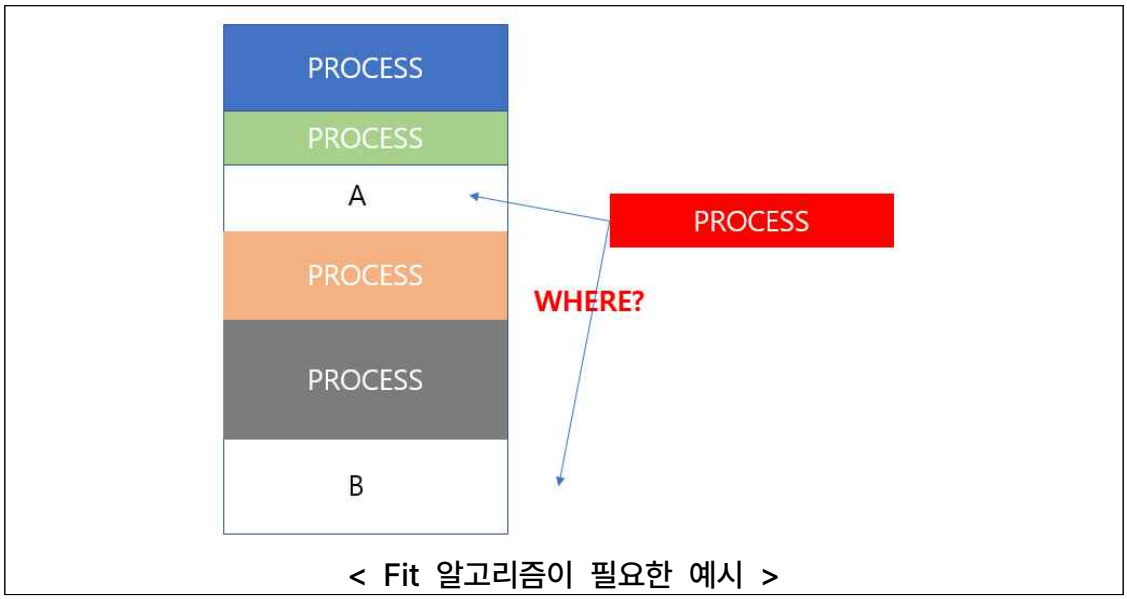


[그림 2-12] 세그먼테이션 기법

: 가변 분할 방식을 사용하며, 페이징 기법과 마찬가지로 세그멘테이션 기법도 매핑 테이블을 사용한다. 이를 세그멘테이션 테이블이라고 한다. 세그멘테이션 테이블에는 세그멘테이션 크기를 나타내는 limit와 물리 메모리의 시작 주소를 나타내는 address가 있다.

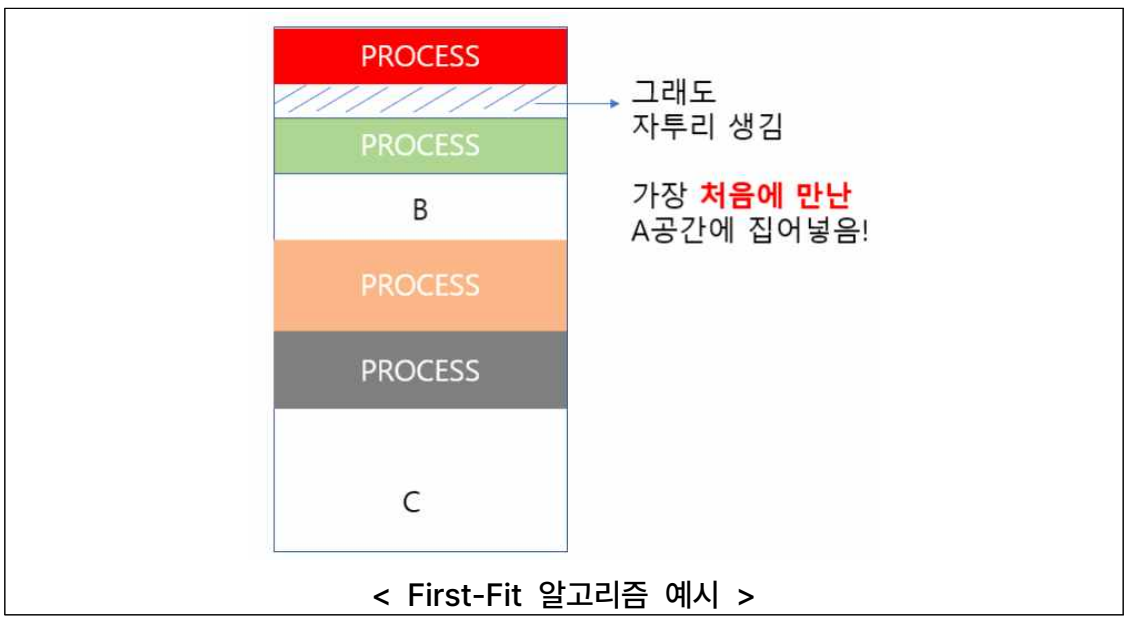
4) 메모리 할당 알고리즘(First-fit, Best-fit, Worst-fit)

▶ Fit 알고리즘은 운영체제에서 메모리의 공간 효율성을 좋게 하기 위해 메모리 할당을 어떻게 할지 결정하는 알고리즘이다.



< Fit 알고리즘이 필요한 예시 >

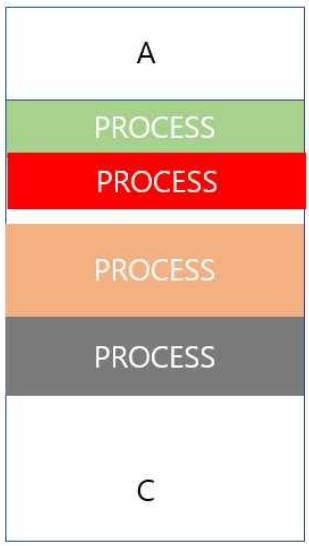
(1) First Fit(최초 적합): 이 알고리즘은 메모리를 처음부터 검색하여 요구 사항을 충족하는 첫 번째 빈 메모리 블록을 찾아 할당하는 방식이다. 이 방식은 검색 속도가 빠르다는 장점이 있지만, 메모리의 후반부에 큰 빈 메모리 블록이 남겨져 있을 수 있다는 단점이 있다.



< First-Fit 알고리즘 예시 >

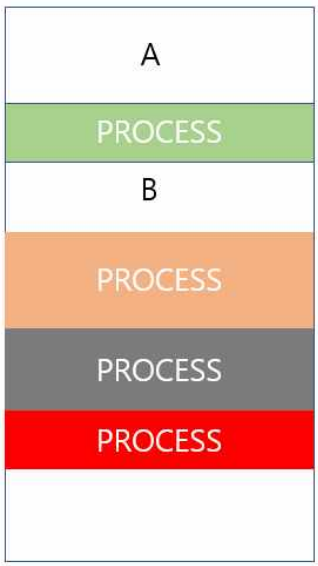
(2) Best Fit: 이 알고리즘은 메모리의 모든 빈 블록을 검사하여 요구 사항을 가

장 잘 충족하는 블록을 찾아 할당한다. 이는 요구 사항보다 크거나 같으면서 가장 작은 빈 블록을 선택하는 것을 의미한다.(이 때, 단편화 문제를 고려한다.) 이 방식은 메모리를 효율적으로 사용할 수 있다는 장점이 있지만, 모든 빈 블록을 검사해야 하므로 시간이 오래 걸릴 수 있다는 단점이 있다.



< Best-Fit 알고리즘 예시 >

(3) Worst Fit: 이 알고리즘은 메모리의 모든 빈 블록을 검사하여 가장 큰 빈 메모리 블록을 찾아 할당한다. 이 방식은 큰 메모리 요구 사항을 처리하는 데 유리하지만, 메모리 내에 사용되지 않는 작은 빈 메모리 블록들을 많이 남기게 될 수 있다는 단점이 있다.



< Worst-Fit 알고리즘 예시 >

6. 장치 관리 방법

1) 디스크 관리

▶ 디스크 관리는 운영체제에서 디스크 장치를 효율적으로 활용하고 데이터를 저장, 접근, 관리하는 기능을 담당한다. 디스크는 주기억장치보다 용량이 크고 비용이 낮은 대용량 저장장치로서 중요한 역할을 수행하므로, 이를 효율적으로 관리하는 것이 중요하다.

(1) 디스크 파티셔닝 (Disk Partitioning): 물리적인 디스크 공간을 여러 개의 파티션으로 분할하는 과정이다. 각 파티션은 독립된 공간으로 간주되어 별도의 파일 시스템을 가질 수 있다.

(2) 디스크 할당 (Disk Allocation): 빈 공간에 파일을 저장하기 위해 어떻게 디스크 블록(데이터 단위)을 어떻게 할당할지 결정하는 방법이다.

- 연속 할당 (Contiguous Allocation): 파일은 연속된 블록에 저장되며, 시작 위치와 길이가 지정된다.
- 연결 리스트 할당 (Linked List Allocation): 각 블록마다 다음 블록의 주소 정보가 저장되어 있어서 연결 리스트 형태로 파일이 구성된다.
- 색인 할당 (Indexed Allocation): 각 파일마다 인덱스 블록이 생성되어 해당 파일의 데이터 블록 주소 정보를 가지고 있다.

(3) 파일 압축 및 압축 해제: 용량 절약 및 I/O 성능 향상을 위해 압축 기술을 사용하여 데이터를 압축하거나 해제한다.

(4) 디스크 스케줄링 (Disk Scheduling): 여러 개의 I/O 요청 사이에서 어떤 순서로 요청들을 처리할지 결정하는 방법이다.

- FCFS(First-Come-First-Served) 스케줄링: 요청된 순서대로 처리한다.
- SSTF(Shortest Seek Time First) 스케줄링: 현재 위치에서 가장 가까운 요청부터 처리한다.
- SCAN 스케줄링: 한 방향으로 이동하면서 요청들을 처리한 후 반대 방향으로 왕복하며 이동하여 처리한다.
- C-SCAN(Circular SCAN) 스케줄링: 한 방향으로 이동하면서 한 쪽 끝까지 이동한 후 반대 방향으로 되돌아오며 요청들을 처리한다.

(5) 디스크 캐싱 (Disk Caching): 메모리 상에 자주 접근되는 데이터나 인덱싱 정보 등 일부 데이터를 보관하여 I/O 성능 향상을 위해 사용된다.

(6) 디스크 오류 복구: 디렉터리와 FAT(FAT32 등) 같은 메타데이터 백업과 체계

적인 오류 검사와 복구 메커니즘 등 오류 복구 기능도 포함될 수 있다.

2) 파일 시스템

▶ 파일 시스템(File System)은 운영 체제가 저장 장치에 데이터를 읽고 쓰기 위해 사용하는 방법을 정의하는 것이다. 이것은 운영 체제의 핵심 부분으로, 파일 및 디렉토리/폴더의 구조와 규칙을 결정하며, 어떻게 저장 공간이 할당되고 회수되는지를 관리한다.

(1) 파일 저장: 파일 시스템은 데이터를 비트로 변환하여 물리적인 디스크에 저장한다. 이는 사용자가 텍스트 문서, 이미지, 오디오 파일 등 다양한 형태의 데이터를 생성하고 수정할 수 있게 해준다.

(2) 공간 관리: 파일 시스템은 디스크 공간을 효율적으로 관리하기 위해 '블록' 또는 '클러스터'라는 단위로 나눈다. 각각의 파일이 여러 개의 블록에 분산되어 저장될 수 있다.

(3) 파일 검색: 사용자가 원하는 파일을 찾을 수 있도록 이름, 위치 정보, 메타데이터(생성 날짜, 수정 날짜 등)와 함께 파일을 인덱싱한다.

(4) 보안: 접근 권한 및 암호화를 통해 특정 사용자만이 특정 파일이나 폴더에 접근할 수 있도록 제한한다.

(5) 오류 복구: 예기치 않은 종료나 하드웨어 오류 후에도 데이터 손실을 최소화하기 위한 방법들(예: 저널링)을 제공한다.

▶ 파일 시스템의 종류

(1) Windows: 가장 일반적으로 사용되는 파일 시스템은 NTFS(New Technology File System)이다. 이는 높은 성능, 안정성, 보안 기능을 제공하며, 대용량 파일과 디스크를 지원한다. 또한 FAT(File Allocation Table)와 exFAT(Extended File Allocation Table)도 지원하며, 이들은 주로 외부 저장 장치에 사용된다.

· NTFS (New Technology File System): NTFS는 마이크로소프트가 개발한 파일 시스템으로, Windows NT 3.1 이후의 버전에서 기본적으로 사용되고 있다. 대용량 파일 및 볼륨을 지원하며, 메타데이터에 대한 저널링 기능, 암호화, 디스크 할당량 관리, 하드 링크 등의 고급 기능을 제공한다.

· FAT (File Allocation Table): FAT는 초기 개인용 컴퓨터에서 널리 사용되던 파일 시스템이다. 그 단순성과 높은 호환성 때문에 여러 장치와 운영 체제에서 지원된다. FAT16과 FAT32 두 가지 주요 버전이 있으며, 이들은 각각 16비트와 32비트 클러스터 주소를 지원한다.

· FAT32: FAT32는 FAT의 확장판으로서, 더 큰 디스크 크기와 파일 크기를 지

원하게 해준다(최대 볼륨 크기는 약 8TB이며 최대 파일 크기는 4GB). 그러나 NTFS나 exFAT 같은 후속 파일 시스템에 비해 성능과 효율성 면에서 한계가 있다.

· exFAT (Extended File Allocation Table): exFAT도 마이크로소프트가 개발한 파일 시스템으로서, 특히 대용량 메모리 카드 및 USB 플래시 드라이브 등 외부 저장 장치에 적합하다. exFAT는 거의 제한 없는 볼륨 및 파일 크기를 지원하며(최대 64ZB), FAT32보다 훨씬 더 큰 파일을 저장할 수 있다.

(2) macOS: macOS에서는 기본적으로 APFS(Apple File System)를 사용한다. APFS는 SSD 최적화, 암호화, 공간 효율성 등을 위해 설계되었다. 이전에는 HFS+(Hierarchical File System Plus)가 사용되었으며, macOS는 여전히 이를 지원한다.

· HFS+ (Hierarchical File System Plus): HFS+는 Apple이 개발한 파일 시스템으로, 1998년에 도입된 Mac OS 8.1부터 macOS High Sierra까지의 기본 파일 시스템으로 사용되었다. HFS의 후속 버전으로, 대용량 하드 드라이브를 지원하며 유니코드를 사용한 파일 이름과 향상된 데이터 무결성 기능을 제공한다.

· APFS (Apple File System): APFS는 Apple이 2017년에 도입한 macOS, iOS, tvOS, 및 watchOS를 위한 파일 시스템이다. HFS+의 후속으로 개발되었으며 SSD 최적화, 암호화, 공간 효율성 등을 위해 설계되었다. 또한 "코피-온-라이트" 방식을 사용하여 데이터를 보호하고 스냅샷 및 클론 기능을 지원한다.

(3) Linux: Linux에서 가장 널리 쓰이는 파일 시스템은 ext4(fourth extended filesystem)이다. 그 외에도 XFS(eXtended Filesystem), Btrfs(B-tree file system), ZFS(Zettabyte file system) 등 다양한 파일 시스템을 지원한다.

· ext4 (fourth extended filesystem): ext4는 Linux에서 가장 널리 사용되는 파일 시스템 중 하나이다. ext3의 후속 버전으로서 대용량 파일과 디스크를 지원하며, 저널링 기능을 제공하여 전원 손실이나 시스템 충돌 등에 대비한다. 또한 파일시스템 수준에서 데이터 압축과 암호화를 지원한다.

· XFS (eXtended Filesystem): XFS는 실시간 서브시스템을 포함하는 고성능 저널링 파일 시스템으로 SGI가 1994년에 처음 개발하였다. 이 후 Linux 커뮤니티에 의해 오픈소스로 개방되었고 현재까지 확장성과 성능 면에서 뛰어나다. XFS는 특별히 대용량 데이터 처리와 병렬 입출력 연산에 적합하게 설계되었다.

· Btrfs (B-tree file system): Btrfs는 Linux 커널용으로 Oracle이 개발한 파일 시스템이다. 'Butter FS' 또는 'Better FS'로 발음되며, ZFS와 유사한 기능을 제공하는 것을 목표로 한다. 스냅샷 생성, 데이터 압축 및 암호화, 오류 검출 및 복구 등의 고급 기능을 제공한다.

· ZFS (Zettabyte file system): ZFS는 Sun Microsystems가 개발하고 현재는 Oracle이 관리하는 고급 파일 시스템 및 볼륨 관리자이다. 대용량 저장 장치를 지원하며 데이터 무결성에 초점을 맞추고 있다. 스냅샷 및 클론 생성, 데이터 압축 및 암호화, 자동 오류 검출 및 복구 등의 기능을 포함한다.

▶ FHS(Filesystem Hierarchy Standard)

: 리눅스와 유닉스 계열 시스템에서 디렉터리와 파일의 배치를 표준화하기 위해 만들어진 규약이다.

▶ FHS 디렉토리 구조

경로	설명
/	루트 디렉터리로, 파일 시스템 계층의 최상위에 위치한다. 모든 다른 디렉터리는 이 루트 디렉터리 아래에 위치한다.
/bin	이 디렉터리는 'Binaries'의 약자로, 시스템의 모든 사용자가 사용할 수 있는 기본적인 명령어들이 들어 있다. 이 명령어들은 파일 조작, 프로세스 관리, 텍스트 처리 등의 일반적인 작업에 필요한 것들이다. 예를 들어, bash, cat, cp, ls, rm 등의 명령어가 /bin에 위치해 있다.
/sbin	이 디렉터리는 'System Binaries'의 약자로, 시스템 관리에 필요한 명령어들이 들어 있다. 이 명령어들은 주로 시스템 root 사용자)가 사용하며, 시스템 부팅, 복구, 복원 등의 작업에 필요한 것들이다. 예를 들어, fdisk, ifconfig, mkfs, reboot 등의 명령어가 /sbin에 위치해 있다.
/etc	이 디렉터리는 시스템의 전반적인 설정 파일들을 저장하는 곳이다. /etc 안의 파일들은 주로 텍스트 형식으로 되어 있어, 관리자가 직접 편집하여 시스템의 동작을 설정할 수 있다.
/dev	이 디렉터리는 장치 파일을 저장하는 곳이다. 리눅스와 유닉스 계열 시스템에서는 하드웨어 장치를 파일처럼 다루는데, 이러한 장치 파일들이 /dev 디렉터리에 위치하게 된다. 각 파일은 특정 하드웨어 장치를 나타내며, 파일에 데이터를 쓰거나 읽어서 장치와 상호작용할 수 있다.
/home	이 디렉터리는 각 사용자의 홈 디렉터리를 저장하는 곳이다. /home 디렉터리 아래에는 각 사용자 이름으로 디렉터리가 만들어지며, 사용자는 해당 디렉터리에서 개인 파일을 생성하고 관리할 수 있다.
/lib	이 디렉터리는 시스템 라이브러리를 저장하는 곳이다. /lib 디렉터리에는 시스템이 동작하는 데 필요한 기본적인 라이브러리 파일들이 포함되어 있다. 이러한 라이브러리들은 /bin과 /sbin 디렉터리에 있는 실행 파일들이나 시스템이 동작하는 데 필요한 다른 프로그램들이 공통적으로 사용하는 코드를 포함하고 있다.
/opt	이 디렉터리는 'optional'의 약어로, 추가적인 소프트웨어 패키지들이 설치되는 곳이다. 일반적으로, /opt 디렉터리는 어떤 패키지 관리 시스템에 의해 관리되지 않는, 독립적으로 설치되는 큰 소프트웨어 패키지를 위해 사용된다. 각 소프트웨어 패키지는 일반적으로 자신의 이름으로 /opt 아래에 하위 디렉터리를 만들고, 그 안에 모든 필요한 파일들을 두게 된다.
/tmp	이 디렉터리는 임시 파일들을 저장하는 곳이다. '/tmp' 디렉터리는 시스템이나 사용자가 잠시 사용하고 버릴 파일들을 위한 공간을 제공한다. 이 디렉터리에 저장된 파일은 일정 시간이 지나면 시스템에 의해 자동으로 삭제될 수 있다.
/usr	이 디렉터리는 시스템의 대부분의 사용자 응용 프로그램과 관련 파일들이 위치하는 곳이다. /usr 디렉터리는 주로 읽기 전용 파일을 포함하며, 여러 하위 디렉터리들을 포함하고 있다.
/var	이 디렉터리는 시스템 운영 중에 생성되는 가변 데이터를 저장하는 곳이다.

< 리눅스 커널 프로그래밍 >

1. 리눅스 개요

1) 리눅스 설치 및 관리

▶ 리눅스 설치와 관리 작업

(1) 리눅스 배포판 선택: 먼저 사용할 리눅스 배포판을 선택한다. Ubuntu, CentOS, Fedora 등 다양한 배포판 중에서 목적과 요구사항에 맞는 것을 선택한다.

(2) 설치 매체 준비: 선택한 리눅스 배포판의 ISO 이미지를 다운로드하고, 설치 USB 또는 DVD를 생성한다.

(3) 시스템 부팅 및 설치: 부팅 가능한 설치 매체를 이용하여 시스템을 부팅하고, 설치 과정을 진행한다. 일반적으로 언어 설정, 디스크 파티셔닝, 사용자 계정 생성 등의 단계가 포함된다.

(4) 기본 시스템 설정: 설치 후 기본 시스템 설정 작업을 수행한다.

- 네트워크 설정: IP 주소 할당, DNS 구성 등 네트워크 설정을 수행한다.
- 패키지 업데이트: 패키지 관리자를 사용하여 시스템 패키지들을 최신 버전으로 업데이트한다.
- 방화벽 설정: 필요에 따라 방화벽(Firewall) 규칙을 구성하여 보안 설정을 수립한다.

(5) 사용자 및 권한 관리: 필요한 경우 새로운 사용자 계정과 그룹을 생성하고 권한 관련 작업을 수행한다.

(6) 서비스 관리: 서버나 워크 스테이션에서 실행되는 서비스(예: 웹 서버, 데이터베이스)들의 시작/중지/재시작 등의 작업을 수행한다.

- systemd: 대부분의 최신 리눅스 시장은 systemd라는 초기화 및 서비스관리 용도로 개발된 표준 시장인 systemd를 사용한다.
- service / systemctl 명령어 : 해당하는 목록보기(service list), 시작(start), 중지(stop), 재시작(restart)등 의 기능 제공

(7) 로그 파일 분석: 로그 파일은 시장상태와 성능 문제 해결에 유용한 정보가 포함되어 있으므로 주기적으로 분석해야 한다.

2) 커널 구조

▶ 리눅스 커널은 주로 C 언어로 작성된 운영체제의 핵심 부분이다. 이는 하드웨어와 소프트웨어 간의 인터페이스 역할을 하며, 시스템 자원과 장치를 관리하고 프로세스와 메모리 관리 등 다양한 기능을 수행한다.

리눅스 커널은 모노리식(Monolithic) 구조를 가지며, 모듈화가 가능한 형태로 설계되었다. 즉, 전체적인 기능이 하나의 큰 프로그램으로 구성되어 있지만 필요에 따라서는 특정 기능들을 모듈 형태로 추가하거나 제거할 수 있다.

▶ 모노리식 커널(Monolithic Kernel)과 마이크로 커널(Microkernel)

(1) 모노리식 커널(Monolithic Kernel): 모노리식 커널은 하나의 큰 프로세스로 실행되며, 모든 기본 시스템 서비스를 하나의 덩어리로 합쳐서 처리한다(모듈화 가능한 형태로 설계되어 모듈을 추가하거나 제거 가능). 이로 인해 모노리식 커널은 효율적인 성능을 발휘할 수 있다. 그러나 이런 구조는 시스템의 한 부분에 문제가 발생하면 전체 시스템에 영향을 미칠 수 있어 유지보수가 어렵다는 단점이 있다. 또한, 모든 서비스가 하나의 주소 공간에서 실행되므로 안정성과 보안 측면에서 취약할 수 있다.

(2) 마이크로 커널(Microkernel): 커널의 기능을 최소한으로 줄인 후, 나머지 기능들을 사용자 모드에서 실행하는 별도의 서버로 이동시킨다. 이렇게 하면 커널 모드에서 실행되는 코드의 양이 줄어들어 시스템의 안정성이 향상된다. 또한, 개별 서비스가 분리되어 있으므로 유지보수가 쉽고, 한 서비스에 문제가 발생해도 전체 시스템에 영향을 미치지 않는다. 그러나 커널과 사용자 공간 사이에서의 상호 작용이 늘어나면서 오버헤드가 증가하고, 이로 인해 성능이 저하될 수 있다.

▶ 리눅스 커널(2.5 버전부터)에서는 커널의 동작을 디버깅하기 위한 다양한 옵션들을 제공하고 있다.

(1) CONFIG_DEBUG_PREEMPT: 이 옵션은 커널의 선점 가능성에 대한 디버깅을 활성화한다. 커널에서 선점이 발생하지 않아야 하는 상황에서 선점이 발생하면, 버그를 찾아내기 위해 경고 메시지를 출력한다.

- 파라미터: Debug preemptible kernel

(2) CONFIG_DEBUG_KERNEL: 이 옵션은 커널의 일반적인 디버깅을 위한 기본 옵션이다. 이 옵션을 활성화하면, 커널에서 발생하는 여러 가지 문제를 추적하고 디버깅하는데 도움이 되는 다른 디버깅 옵션들을 사용할 수 있게 된다.

- 파라미터: Kernel debugging

(3) CONFIG_DEBUG_KALLSYMS: 이 옵션은 커널의 심볼 테이블을 사용하여 디버깅 정보를 제공한다. 이 옵션을 활성화하면, 커널 패닉이 발생했을 때 함수의 주소 대신에 심볼 이름을 출력하는 등의 기능을 사용할 수 있다.

- 파라미터: Load all symbols for debugging
(4) CONFIG_DEBUG_SPINLOCK_SLEEP: 이 옵션은 스핀락을 사용하는 코드에 대한 디버깅을 활성화한다. 스핀락은 원자적인 연산을 보장하기 위한 동기화 기법 중 하나이다. 이 옵션을 활성화하면, 스핀락을 잡은 상태에서 잠자기 상태로 들어가는 등의 잘못된 동작을 디버깅할 수 있다.
- 파라미터: Sleep-inside-spinlock checking
▶ 커널의 주요 역할
(1) 자원 관리: 커널은 CPU 스케줄링, 메모리 관리, 디스크 관리 등을 통해 시스템의 자원을 효율적으로 분배하고 관리한다.
(2) 프로세스 관리: 커널은 프로세스의 생성, 종료, 중단, 재개 등의 상태를 관리한다. 또한, 프로세스간의 통신(IPC, Inter-Process Communication)도 제어한다.
(3) 장치 드라이버 관리: 커널은 컴퓨터의 하드웨어를 제어하기 위해 장치 드라이버를 관리한다. 장치 드라이버는 특정 하드웨어 장치와 운영 체제 간의 인터페이스를 제공한다.
(4) 시스템 호출 인터페이스 제공: 커널은 사용자 공간 프로그램이 시스템 자원에 접근할 수 있도록 시스템 호출 인터페이스를 제공한다. 시스템 호출을 통해 프로그램은 파일 시스템 접근, 네트워크 통신, 프로세스 관리 등의 서비스를 요청할 수 있다.
(5) 보안: 커널은 시스템의 보안을 담당한다. 이는 사용자의 권한 관리, 메모리 보호, 하드웨어 접근 제어 등을 포함한다.
▶ 커널의 추상화 자원 관리
(1) 프로세스 추상화: 커널은 실행 중인 프로그램에 대한 '프로세스'라는 추상적 개념을 제공한다. 프로세스는 자신만의 가상 메모리 공간과 상태 정보(예: 레지스터 값, 우선 순위 등)를 가지며, 이런 방식으로 여러 프로그램이 동시에 실행되는 것처럼 보일 수 있다.
(2) 메모리 추상화: 커널은 물리적 메모리에 대한 '가상 메모리'라는 개념을 제공하여 각 프로세스가 전체 메모리를 독점하고 있는 것처럼 보이게 한다. 이렇게 함으로써 각 프로세스는 다른 프로세스의 메모리 영역을 침범하지 않고 안전하게 작동할 수 있다.
(3) 파일 시스템 추상화: 커널은 디스크나 다른 저장 장치에 저장된 데이터에 대해 '파일'과 '디렉터리'라는 개념을 제공한다. 사용자나 응용 프로그램은 복잡한 물리적 구조나 저장 방식 등을 몰라도 되며 파일 이름과 경로만으로 원하는 데이터에 접근할 수 있다.
(4) 장치 드라이버 추상화: 커널 내부의 장치 드라이버들은 다양한 하드웨어 장치

들(예: 키보드, 마우스, 디스크 드라이브 등)과 상호작용하는 방법을 정의한다. 이렇게 함으로써 응용 프로그램은 특정 하드웨어의 세부 사항을 몰라도 일관된 방식으로 장치를 사용할 수 있다.

▶ 커널의 주요 구성 요소

(1) 태스크 관리자(Task Manager) : 물리적 자원인 CPU를 추상적 자원인 태스크로 제공하며, 태스크(프로세스 또는 스레드)의 생성 및 종료, 스케줄링, 동기화, 통신, 문맥 교환을 담당한다.

(2) 메모리 관리자 (Memory Manager): 물리적 자원인 메모리를 추상적 자원인 페이지나 세그먼트로 제공하며, 메모리 할당 및 해제, 가상 메모리를 관리하고 메모리를 보호한다.

(3) 파일 시스템 관리자 (File System Manager): 물리적 자원인 디스크를 추상적 자원인 파일로 제공하며, 파일의 생성 및 삭제, 읽기 및 쓰기 기능을 제공하고, 디렉토리, 파일 권한, 메타데이터를 관리한다.

(4) 디바이스 드라이버 관리자 (Device Drivers Manager): 각종 장치를 디바이스 드라이버를 통해 일관되게 접근하게 하고, 디바이스 제어 및 인터럽트 처리, 디바이스 드라이버 간의 통신 기능을 제공한다.

(5) 네트워크 관리자 (Network Manager): 물리적 자원인 네트워크 장치를 추상적 자원인 소켓으로 제공하며, 데이터 송수신, 네트워크 연결 관리, 네트워크 보안 등의 기능을 제공한다.

▶ 커널 관련 명령어

명령어	설명
printk	리눅스 커널에서 로깅을 위해 사용되는 함수이다. 사용자 공간의 printf 함수와 유사하지만, 커널 공간에서 동작하며, 커널 로그 버퍼에 메시지를 기록하는 역할을 한다.
dmesg	이 명령어는 커널 메시지 버퍼의 내용을 출력한다. 시스템 부팅 이후에 발생한 커널 메시지를 모두 볼 수 있다.
kmod	커널 모듈을 관리하는 명령어로, lsmod, insmod, rmmod, modprobe 등의 기능을 통합한 형태이다.
lsmod	현재 로드된 커널 모듈의 목록을 보여준다. 각 모듈의 이름, 크기, 사용 횟수 등을 볼 수 있다.
insmod	지정된 커널 모듈을 로드한다. 모듈 파일의 경로를 인자로 받는다.
rmmod	지정된 커널 모듈을 언로드한다. 모듈의 이름을 인자로 받는다.
modprobe	커널 모듈을 로드하거나 언로드한다. 필요한 경우 종속 모듈도 함께 로드하거나 언로드한다.
uname	시스템 정보를 출력한다. -a 옵션을 사용하면 모든 정보를 한 번에 볼 수 있다.
sysctl	커널 파라미터를 읽고 쓰는 데 사용된다. /proc/sys 디렉터리에 있는 파일을 조작하여 커널의 동작을 변경할 수 있다.
lspci	PCI 버스에 연결된 장치들의 정보를 출력한다.
lsusb	USB 버스에 연결된 장치들의 정보를 출력한다.

▶ 리눅스 파일 관련 명령어

명령어	설명
cat	파일 내용 출력
more	페이지 단위로 파일 내용 출력
head	파일의 앞부분(10줄) 출력
tail	파일의 뒷부분(10줄) 출력
wc	줄/단어/문자 수 세기
cp	파일1을 파일2로 복사
mv	파일1을 파일2로 이름 변경
rm	파일 삭제
rmdir	디렉터리 삭제
ln	링크 만들기

2. 커널 서비스

1) 시스템 콜

▶ 시스템 콜(system call)은 사용자 모드에서 실행되는 프로그램이 운영체제의 커널 서비스를 요청하기 위한 인터페이스이다. 이는 하드웨어 자원에 접근하거나,

입출력, 프로세스 생성과 종료, 네트워크 통신 등의 기능을 사용하기 위해 필요하다.

시스템 콜은 보안을 유지하기 위한 중요한 메커니즘이기도 하다. 일반적으로, 직접적인 하드웨어 접근은 커널 모드에서만 가능하며 사용자 모드에서 실행되는 프로그램은 이를 직접 수행할 수 없다. 따라서 사용자 프로그램이 하드웨어 자원에 접근하려면 시스템 콜을 통해 커널에 서비스를 요청해야 한다.

▶ 사용자 모드(User Mode)와 커널 모드(Kernel Mode)

· 사용자 모드(User Mode): 사용자 모드에서 실행되는 프로그램(일반적으로 사용자 응용 프로그램)은 제한된 권한만 갖는다. 이들은 직접적으로 하드웨어나 중요 시스템 자원에 접근할 수 없으며, 필요할 경우 운영 체제가 제공하는 API(Application Programming Interface)나 시스템 호출(System Call)을 통해 해당 작업을 요청해야 한다.

· 커널 모드(Kernel Mode): 커널 모드에서 실행되는 코드(일반적으로 운영 체제의 일부)는 하드웨어와 시스템 자원에 대한 완전한 접근 권한을 가진다. 이러한 코드는 장치 드라이버를 제어하거나 메모리 관리와 같은 저수준의 작업을 수행하는 등, 시스템의 핵심 기능을 담당한다. 커널 모드에서 실행되는 코드에 오류가 발생하면 전체 시스템에 심각한 문제를 일으킬 수 있기 때문에, 이러한 코드를 작성하고 수정하는 것은 매우 주의가 필요하다.

· 사용자 모드와 커널 모드 사이의 전환은 보안과 안전상의 이유로 매우 중요하다. 예를 들어, 임의의 응용 프로그램이 저레벨 하드웨어 자원에 직접 접근할 수 있다면, 심각한 보안 위협이 될 수 있다(운영체제의 중요 데이터 파괴 등). 따라서 사용자 응용 프로그램들은 일반적으로 제한된 권한인 사용자 모드에서 동작하며 필요시 커널 모드에서 동작하는 OS 서비스를 요청하여 특별한 작업들을 처리하게 된다.

▶ 시스템 콜의 동작 과정

(1) 사용자 프로그램이 시스템 콜을 호출한다.

(2) 이 호출로 인해 CPU가 사용자 모드에서 커널 모드로 전환된다.

(3) 지정된 시스템 콜 번호에 따라 해당하는 커널 함수가 실행된다.

(4) 함수 실행 결과가 사용자 프로그램으로 반환되고 CPU는 다시 사용자 모드로 전환된다.

▶ 시스템 콜의 유형

(1) 프로세스 제어: 프로세스 생성/종료/스케줄링 등과 관련된 작업을 담당한다. 예시로 fork(), exit(), wait()등이 있다.

(2) 파일 조작: 파일 생성/삭제/읽기/쓰기 등과 관련된 작업을 처리하는 시스템 콜이다. 예시로 open(), read(), write(), close() 등이 있다.

(3) 장치 관리: 장치와 관련된 작업을 처리하는 시스템 콜이다. 장치 정보 요청 및 변경, 요청한 작업의 완료 대기 등이 포함된다.
(4) 정보 유지: 운영체제나 사용자 데이터에 대한 정보를 얻거나 설정하는 데 사용되는 시스템 콜이다. 예를 들어, 현재 날짜와 시간을 얻거나 설정하거나, 시스템 상태 정보를 조회하는 것들이 해당한다.
(5) 통신(Communication): 프로세스 간 통신(IPC) 또는 네트워크 통신을 위한 시스템 콜이다. 메시지 전송과 수신, 신호 보내기와 받기 등 다양한 형태의 통신 메커니즘을 지원한다.

▶ 시스템 콜 주요 레퍼런스(Reference)

번호	함수이름	동작	번호	함수이름	동작
1	read()	파일을 버퍼에 읽어옴	11	mmap()	파일을 메모리에 매핑
2	write()	버퍼내용을 파일에 씀	12	munmap()	매핑된 메모리 영역 해제
3	open()	파일을 열음	13	pipe()	두 개의 파일 디스크립터 생성 후 파이프로 연결
4	close()	파일을 닫음	14	dup()	열려 있는 파일 디스크립터 복제
5	fork()	자식 프로세스 생성	15	chdir()	현재 작업 디렉터리로 경로 변경
6	wait()	자식 프로세스 종료까지 기다림	16	stat()	파일 디스크립터에 대한 정보를 가져옴
7	execv()	프로그램의 실행	17	ioctl()	장치 특화 명령 수행 또는 특수한 입/출력 작업 수행
8	exit()	프로세스 정상 종료	18	getuid()	현재 프로세스의 사용자 ID 반환
9	getpid()	프로세스 PID 반환	19	setuid()	현재 프로세스의 사용자 ID 설정
10	kill()	프로세스 강제 종료 및 특정 시그널을 특정 프로세스에게 보냄	20	socket()	소켓 생성 및 소켓 타입과 프로토콜 지정

2) 시그널과 인터럽트

▶ 리눅스 시스템에서의 시그널(Signal)과 인터럽트(Interrupt)는 비동기적인 이벤트를 처리하는 데 사용되는 중요한 메커니즘이다.

▶ 시그널 (Signal): 시그널은 일종의 소프트웨어 인터럽트로, 프로세스에게 특정 사건이 발생했음을 알리는 방법이다. 예를 들어, 프로세스가 정의할 수 없는 연산을 수행하려고 하면 운영체제는 해당 프로세스에게 SIGFPE (부동소수점 오류) 시

그널을 보낸다.

▶ 리눅스에서는 kill(), raise() 명령어를 사용해 특정 프로세스에게 원하는 시그널을 보낼 수 있다

ex) $ kill –signal pid

▶ 시그널을 수신한 프로세스의 반응

(1) 시그널에 대해 기본적인 방법으로 대응한다. 대부분의 시그널에 대해 프로세스는 종료하게 된다.

(2) 시그널을 무시한다. 단, SIGKILL과 SIGSTOP은 무시될 수 없다.

(3) 프로그래머가 지정한 함수(시그널 핸들러)를 호출한다.

▶ 시그널 핸들러(Signal Handler) : 프로세스가 특정 시그널을 포착했을 때 수행해야 할 별도의 함수이다. 프로세스는 시그널을 포착하면 현재 작업을 일시 중단하고 시그널 핸들러를 실행하고, 실행이 끝나면 중단된 작업을 재개한다.

▶ 시그널 종류

번호	이름	설명
1	SIGHUP	터미널 접속이 끊겼을 때 보내지는 시그널로서 데몬 프로세스를 재시작할 때 사용함
2	SIGINT	키보드로부터 오는 인터럽트 시그널로 실행을 중지시킴(CTRL+C)
3	SIGQUIT	키보드로부터 오는 실행 중지 시그널(CTRL+\). 기본적으로 프로세스를 종료시킨 뒤 코어를 덤프함
4	SIGILL	비정상적인 명령어 실행 시 발생함
5	SIGABRT	오류 발생시 프로그램이 비정상 종료되었을 때 발생함
6	SIGFPE	부동소수점 연산 오류 시 발생함
7	SIGKILL	프로세스를 강제로 종료하는데 사용됨
8	SIGSEGV	세그멘테이션 폴트에 의해 발생함
9	SIGPIPE	파이프가 닫혀있는 상태에서 쓰기 작업 수행시 발생함
10	SIGTERM	프로세스에게 종료하도록 요청할 때 주로 사용됨

* 데몬 프로세스 : 백그라운드에서 동작하는 프로세스를 말한다.

* 세그멘테이션 폴트 : 프로그램이 할당되지 않은 메모리 영역에 접근할 때 발생하는 오류이다.

* 코어 덤프 : 프로그램이 비정상적으로 종료되었을 때 메모리 상태 등의 정보를 담은 파일이다.

▶ 인터럽트 (Interrupt): 인터럽트는 하드웨어 장치가 CPU에게 어떠한 사건이 발생했음을 알리기 위해 사용하는 메커니즘이다. 예를 들어, 키보드에서 키 입력이 발생하면 키보드 컨트롤러는 CPU에 인터럽트를 보내서 이 사실을 알린다.

▶ 인터럽트의 종류

(1) 입출력(I/O) 인터럽트 : 해당 입출력 하드웨어가 주어진 입출력 동작을 완료하였거나 입출력의 오류 등이 발생했을 때 CPU에 요청하는 인터럽트이다.

(2) 외부 인터럽트 : 시스템 타이머에서 일정한 시간이 만료된 경우나 오퍼레이터

가 콘솔 상의 인터럽트 키를 입력한 경우 또는 다중 처리 시스템에서 다른 처리기로부터 신호가 언 경우 등에 발생한다.
(3) SVC(SuperVisor Call) 인터럽트 : 사용자 모드에서 실행 중인 프로그램이 커널 모드의 서비스나 기능을 요청할 때 발생한다.
(4) 기계 검사 인터럽트 : 컴퓨터 자체의 기계작인 장애나 오류로 인한 인터럽트이다.
(5) 프로그램 오류 인터럽트 : 주로 프로그램 실행 오류로 발생한다.
(6) 재시작 인터럽트 : 오퍼레이터가 콘솔의 재시작 키를 누를 때 발생한다.
▶ 인터럽트 핸들러(Interrupt Handler) or 인터럽트 서비스 루틴(Interrupt Service Routine, ISR)
: 시스템에서 특정 인터럽트가 발생할 때 실행되는 하나의 루틴을 말한다.
인터럽트 핸들러의 역할은 다음과 같다.
(1) 인터럽트 식별: 인터럽트 핸들러는 먼저 어떤 종류의 인터럽트가 발생한 것인지를 확인한다. 이 정보는 보통 인터럽트 요청 번호(IRQ)에 의해 제공된다.
(2) 필요한 처리 수행: 그 후, 해당하는 서비스 루틴은 필요한 모든 처리를 수행한다. 예를 들어 I/O 장치에서 데이터를 읽거나 쓰거나, 에러 조건을 체크하거나 해결하는 등의 작업이 있다.
(3) 상태 복원 및 복귀: 마지막으로, 원래의 프로세스 상태를 복원하고 프로세서 제어권을 원래의 프로세스에 반환함으로써 작업을 재개한다.
▶ 인터럽트 컨트롤러(Interrupt Controller)
: 인터럽트 컨트롤러는 하드웨어 장치로서, 다양한 소스에서 발생하는 인터럽트를 CPU에 전달하는 역할을 한다. 이 장치는 여러 개의 인터럽트 요청을 동시에 처리하고, 그 중 어느 것이 우선순위가 높은지 결정하며, 적절한 순서로 CPU에 전달한다.
인터럽트 컨트롤러의 주요 기능 및 작동 원리는 다음과 같다.
(1) 인터럽트 수집: 인터럽트 컨트롤러는 시스템 내 다양한 하드웨어 장치들로부터 발생하는 인터럽트 신호를 수집한다. 각각의 장치는 고유한 인터럽트 요청 번호(IRQ)를 가지며, 이것으로 해당 장치와 관련된 특정 인터럽트를 구별할 수 있다.
(2) 우선순위 결정: 여러 개의 인터럽트 요청이 동시에 들어올 경우, 각각의 우선순위를 결정하여 어떤 요청을 먼저 처리할지 정한다. 이 우선순위는 보통 하드웨어 설정이나 프로그램 코드에 의해 정해진다.
(3) CPU 알림: 선택된(즉, 가장 우선 순위가 높은) 인터럽트 요청을 CPU에 알림으로써 CPU가 현재 작업을 중단하고 해당 IRQ와 연관된 서비스 루틴(인터럽트 핸들러)을 실행하도록 한다.

(4) 인애킹(In Acknowledgment): CPU가 해당 IRQ를 받아들였음을 확인하기 위해 사용되는 신호이다. 이 신호를 받으면 컨트롤러는 그 다음 우선 순위의 IRQ를 준비한다.

(5) End of Interrupt (EOI): 서비스 루틴이 모든 처리 작업을 완료하면 EOI 신호를 보내서 그 사실을 컨트롤러에 알림으로써 새로운 인터럽트 처리 준비 상태로 돌아간다.

▶ 인터럽트의 동작 과정

(1) 인터럽트 발생: 하드웨어 장치, 타이머, 소프트웨어 등에서 어떤 이벤트가 발생하여 시스템에 인터럽트를 요청한다.

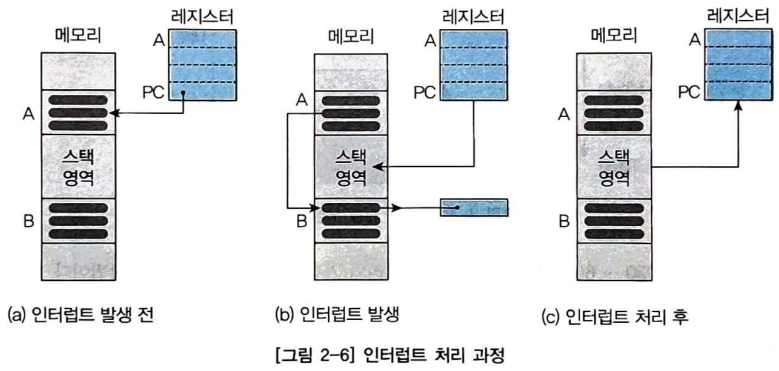
(2) 인터럽트 감지: 프로세서는 매 명령어 실행 후에 인터럽트를 확인한다. 만약 인터럽트가 감지되면 현재 실행 중인 작업을 일시 중단한다.

(3) 현재 상태 저장: 프로세서는 현재 실행 중인 작업의 상태를 저장한다. 이 정보는 나중에 원래의 작업으로 돌아갈 때 사용된다. 이 정보는 보통 프로세스 제어 블록(Process Control Block, PCB)에 저장되며, 프로그램 카운터(Program Counter), 레지스터 값 등을 포함한다.

(4) 인터럽트 처리: CPU는 인터럽트 벡터 테이블(Interrupt Vector Table)에서 해당하는 서비스 루틴의 주소를 찾는다. 그리고 그 주소로 점프하여 해당 서비스 루틴(즉, 인터럽트 핸들러)를 시작한다.

(5) 서비스 수행: 서비스 루틴은 필요한 모든 처리를 수행한다. 예를 들어 I/O 장치에서 데이터를 읽거나 쓰거나, 에러 조건을 체크하거나 해결하는 등의 작업이 있다.

(6) 상태 복원 및 복귀: 서비스 루틴이 완료되면 원래의 상태 정보(PCB 내용)가 복원된다. 그리고 CPU 제어권이 원래의 프로세스(또는 다음으로 스케줄링된 프로세스)에게 반환된다. 이로써 원래의 작업이나 새로운 작업이 계속 실행된다.



[그림 2-6] 인터럽트 처리 과정

< 인터럽트 처리 과정 >

(a) 인터럽트가 도달하기 전에 프로그램 A를 실행한다고 가정하자. 프로그램

카운터는 현재 명령어를 가리킨다.

(b) 프로세서에 인터럽트 신호가 도달하여 현재 명령어를 종료하고 레지스터의 모든 내용을 스택 영역에 보낸다. 그리고 프로그램 카운터에서는 인터럽트 처리 프로그램(프로그램B)의 시작 위치를 저장하고 제어를 넘긴 프로그램 B를 실행한다.

(c) 인터럽트 처리 프로그램을 완료하면 스택 영역에 있던 내용을 레지스터에 다시 저장하며, 프로그램 A가 다시 시작하는 위치를 저장하고 중단했던 프로그램 A를 재실행한다.

▶ 인터럽트 요청 번호(Interrupt Request Number, IRQ)

: 특정 인터럽트를 구별하기 위해 사용되는 번호이다. 컴퓨터 시스템에서 여러 하드웨어 장치가 동시에 인터럽트를 발생시킬 수 있기 때문에, 각각의 인터럽트가 어떤 장치에서 발생한 것인지 알아내기 위해 사용된다.

인터럽트 요청 과정에서의 IRQ의 역할은 다음과 같다.

(1) 인터럽트 발생: 하드웨어 장치는 특정 이벤트(ex.데이터 준비 완료, 오류 발생 등)가 일어나면 프로세서에 인터럽트를 요청한다.

(2) IRQ 확인: 프로세서는 인터럽트 컨트롤러를 통해 IRQ번호를 받아온다. 이 IRQ 번호는 후속 처리 단계에서 해당하는 인터럽트 핸들러(또는 서비스 루틴)을 찾는 데 사용된다.

(3) 인터럽트 벡터 테이블 조회: 프로세서는 받아온 IRQ 번호에 대응하는 핸들러의 주소를 찾기 위해 "인터럽트 벡터 테이블"이라고 불리우는 테이블을 조회한다.

(4) 핸들러 실행: 조회된 주소로 점프하여 해당하는 인터럽트 핸들러 코드를 실행한다. 이 코드 내부에서 필요한 작업(예: 데이터 읽기/쓰기, 에러 해결 등)을 수행하게 된다.

▶ 인터럽트와 시그널의 차이점

: 인터럽트는 하드웨어 레벨의 신호로서, CPU에게 중요한 사건이 발생했음을 알리는 역할을 하는 반면에 시그널은 소프트웨어 레벨의 메커니즘으로서 프로세스 간 통신(IPC) 혹은 동일한 프로세스 내부에서 예외 처리 등을 위해 사용된다.

3) 벡터 인터럽트(Vector Interrupt)

▶ 벡터 인터럽트는 컴퓨터 시스템에서 발생하는 이벤트를 처리하는 방법 중 하나이다. 이는 특히 중앙 처리 장치(CPU)가 외부 이벤트에 대응할 수 있게 하는데 사용되며, 각 이벤트는 고유한 인터럽트 벡터를 가진다.

인터럽트 벡터는 일반적으로 메모리 내의 특정 위치를 가리키며, 해당 위치에는 인터럽트를 처리하는 데 필요한 코드인 인터럽트 서비스 루틴(ISR)의 주소가 저장되

어 있다. 인터럽트가 발생하면 CPU는 현재 실행 중인 작업을 중단하고, 대신 인터럽트 벡터를 통해 지정된 ISR을 실행한다. ISR 실행이 완료되면 CPU는 중단된 작업을 계속 진행한다. 벡터 인터럽트는 하드웨어적으로 인터럽트 우선순위를 판별하며, 별도의 프로그램 루틴이 없기 때문에 응담 속도가 빠르다. 하지만 회로가 복잡하고 융통성이 없으며, 추가 하드웨어가 필요하므로 비경제적이다.

4) 폴링(Polling)과 데이지 체인(Daisy Chain)

▶ CPU 외부의 하드웨어적인 요구에 의해서 정상적인 프로그램의 실행순서를 변경하여 보다 시급한 작업을 먼저 수행한 후에 다시 원래 프로그램 복귀하는 것으로 소프트웨어 적인 폴링방식과 하드웨어적인 데이지 체인이 있다

▶ 폴링(Polling): 소프트웨어적으로 인터럽트 우선순위를 판별하는 방법으로서 CPU가 주기적으로 각 장치의 상태를 확인(폴링)하여 요청이 있는 장치를 찾아 그 요청을 처리하는 방식이다. 폴링 방식은 우선순위 변경이 용이하고 구조가 단순하고 프로그래밍이 쉽지만, CPU가 불필요한 시간을 소비하게 되어 반응시간이 느리고 비효율적이라는 단점이 있다.

▶ 데이지 체인(Daisy Chain): 하드웨어적으로 인터럽트 우선순위를 판별하는 방법으로서 여러 장치를 일렬(직렬)로 연결하여 하나의 신호선을 통해 인터럽트를 전달하는 방식이다. 인터럽트가 발생하면 이 신호가 체인을 따라 전달되며, 각 장치는 신호를 받았을 때 자신의 상태를 확인하고 요청이 있으면 인터럽트를 처리한다. 데이지 체인 방식은 장치의 추가나 제거가 용이하다는 장점이 있지만, 체인에 문제가 발생하면 전체 시스템에 영향을 미칠 수 있다는 단점이 있다.

5) /proc, /sys 파일 시스템, kobject

▶ 리눅스에서 /proc과 /sys는 가상 파일 시스템으로, 실행 중인 커널에 대한 정보를 제공하거나 커널 설정을 변경하는 인터페이스를 제공한다.

(1) /proc 파일 시스템: /proc은 프로세스와 시스템 정보를 덤프 형식으로 제공하는 가상 파일 시스템이다. 이 디렉터리에는 현재 실행 중인 각 프로세스에 대한 디렉터리가 있으며, 각 디렉터리 이름은 해당 프로세스의 PID(Process ID)이다. 또한, /proc/meminfo, /proc/cpuinfo 등의 파일들을 통해 시스템의 전반적인 상태 정보를 얻을 수 있다.

(2) /sys 파일 시스템: /sys는 sysfs라고도 불리며, 주로 하드웨어 장치와 드라이버들에 대한 정보를 제공한다. sysfs는 커널영역에 디바이스들을 객체화하여 유저영역에 정보를 제공하는 가상 파일시스템이다. 이는 udev 같은 도구가 하드웨어 장치를 관리하는 데 사용된다.

▶ kobject: kobject는 리눅스 커널의 객체 모델로서, 커널 내부에서 사용되는 데이터 구조체를 추상화하고 이들 사이의 관계를 정의한다. kobject는 리눅스 커널 내부에서 여러 개체와 그들 간의 관계를 추상화하고 조작하는 중요한 도구이다.
kobject의 주요 기능과 특징은 다음과 같다.
(1) 객체 수명 관리: kobject에는 참조 카운트가 포함되어 있어 해당 객체가 얼마나 많은 곳에서 참조되고 있는지 추적한다. 이를 통해 메모리 누수를 방지하고, 객체가 더 이상 필요하지 않을 때 안전하게 해제할 수 있다.
(2) 시스템 계층 구조 표현: kobject들은 부모-자식 관계를 가질 수 있으므로 시스템 내부의 계층적인 구조를 나타낼 수 있다.
(3) sysfs와 연동: kobject는 sysfs와 밀접하게 연동되어 사용자 공간에서 커널 객체에 접근하는 인터페이스를 제공한다. kobject가 생성되거나 변경될 때 sysfs에 해당 정보가 반영되며, 이것을 통해 사용자는 /sys 디렉토리 아래에서 해당 정보를 조회하거나 수정할 수 있다.
(4) 속성(attribute) 관리: 각 kobject에는 속성(attribute)이라는 메타데이터 집합이 연결될 수 있다. 속성은 일반적으로 kobj_attribute 구조체로 정의되며, 각각 이름과 mode(읽기/쓰기 권한), 그리고 show() 및 store() 함수 포인터 등을 가진다.
▶ ktype: 리눅스 커널에서 kobject의 타입을 정의하는 구조체이다. 이는 특정 종류의 kobject(예: 디바이스, 모듈 등)가 어떻게 동작해야 하는지, 어떤 속성을 가지고 있는지를 명시한다. 각각 다른 종류의 kobjects(예: 장치, 모듈 등)는 각각 다른 ktype을 가진다. 그리고 각각의 ktype은 그 종류에 맞게 동작하도록 필요한 메소드 및 속성들을 제공한다.
ktype 구조체에는 다음과 같은 필드들이 포함될 수 있다.
(1) name: 해당 ktype의 이름이다. 이 이름은 /sysfs/ 파일 시스템 내에 해당 객체가 위치할 경로를 결정하는 데 사용된다.
(2) sysfs_ops: 이는 sysfs 파일 시스템에 대한 연산을 정의한다. 일반적으로 show와 store 연산을 포함하며, 이들 함수는 사용자 공간에서 해당 객체의 속성 값을 읽거나 쓸 때 호출된다.
(3) release: kobject_put() 함수가 호출되어 kobject의 참조 카운트가 0이 되면 호출되는 함수이다. 이 함수 내에서 kobject와 관련된 메모리 해제 등 필요한 정리 작업을 수행한다.
(4) default_attrs: 해당 타입의 kobjects가 기본적으로 가질 sysfs 속성(attribute)들을 목록화한 배열이다.
▶ kset: 리눅스 커널에서 관련된 kobject들을 그룹화하는 데 사용되는 구조체이

다. kset은 하나 이상의 kobject를 포함하며, 각각의 kset는 공통의 특성이나 목적을 공유하는 kobject들로 구성된다. 각각의 kset은 관련된 여러 개체(kobject)들을 그룹화하여 관리하고, sysfs 파일 시스템 내에서 일관된 계층 구조를 형성하는 데 중요한 역할을 한다.

예를 들어, 모든 블록 디바이스는 /sys/block/(sysfs 파일 시스템 내)에 위치한 동일한 kset에 속하게 된다. 이와 유사하게, 모든 네트워크 인터페이스는 /sys/class/net/에 위치한 동일한 kset에 속한다.

각각의 kset은 다음과 같은 필드를 가진다.

(1) list: 해당 kset에 속하는 kobjects를 추적하는 연결 리스트이다.

(2) list_lock: 위에서 언급된 list를 보호하기 위한 스핀락이다.

(3) kobj: kset 자체도 하나의 kobject이며, 이것을 통해 sysfs 계층 구조 내에서 위치할 수 있다.

(4) uevent_ops: 해당 kset에서 발생하는 uevent(사용자 공간으로 전달되는 커널 이벤트) 처리 방식을 정의한다.

3. 메모리 관리

1) 주소 공간 및 구조

▶ 리눅스에서의 주소 공간은 프로세스가 메모리를 사용하는 방식을 정의한다. 각 프로세스는 자신만의 독립적인 가상 주소 공간을 가지며, 이를 통해 실제 물리 메모리에 접근한다.

▶ 일반적으로 리눅스 시스템에서 프로세스의 가상 주소 공간은 다음 네 부분으로 나뉜다.

(1) 텍스트 (Text) 세그먼트: 이 영역에는 실행 가능한 코드가 저장된다. 보통 읽기 전용으로 설정되어 있어, 프로그램이 자신의 명령어를 변경하지 못하게 한다.

(2) 데이터 (Data) 세그먼트: 이 영역에는 초기화된 정적 변수와 전역 변수가 저장된다. 데이터 세그먼트는 두 부분으로 나뉘는데, 초기화된 데이터 영역과 초기화되지 않은 데이터(BSS, Block Started by Symbol) 영역이다.

(3) 스택 (Stack) 세그먼트: 이 영역에는 함수 호출 정보와 지역 변수 등이 저장된다. 스택은 LIFO(Last In First Out) 방식으로 동작하며, 함수 호출 시 새로운 스택 프레임이 생성되고 함수 종료 시 해당 스택 프레임이 제거된다.

(4) 힙 (Heap) 영역 : 동적 메모리 할당을 위한 공간이다. malloc, free 같은 함수를 사용하여 메모리를 동적으로 할당하거나 해제할 수 있다.

▶ 리눅스 커널 주소 공간

: 리눅스 커널 또한 자체적인 주소 공간을 가지고 있다. 이 공간은 커널 코드, 커널 데이터, 페이지 테이블, 시스템용 버퍼 등을 저장하며 사용자 공간과는 별도로 관리된다.

(1) 코드 영역: 커널 코드가 위치하는 영역이다. 이 영역은 읽기와 실행 권한이 있지만, 쓰기 권한은 없다.

(2) 데이터 영역: 커널의 전역 변수와 정적 변수가 위치하는 영역이다. 이 영역은 읽기와 쓰기 권한이 있다.

(3) BSS 영역: 초기화되지 않은 데이터를 위한 영역이다. 프로그램 시작 시에 커널에 의해 자동으로 0으로 초기화된다.

(4) 힙 영역: 동적 메모리 할당을 위한 영역이다. kmalloc, vmalloc 등의 함수를 통해 메모리를 동적으로 할당하고 해제할 수 있다.

(5) 스택 영역: 함수 호출과 로컬 변수 저장을 위한 영역이다. 각 스레드는 자신만의 커널 스택을 가지며, 이는 스레드 생성 시에 할당된다.

(6) 고정 매핑 영역: 특정 물리 메모리 주소를 고정된 가상 주소에 매핑하는 영역이다. 이를 통해 커널은 특정 메모리 주소를 바로 접근할 수 있다.

(7) VMALLOC 영역: 커널의 가상 메모리 할당을 위한 영역이다. 이 영역에서 할당된 메모리는 페이지 테이블을 통해 물리 메모리에 매핑된다.

▶ 리눅스 커널 안의 다양한 종류의 메모리 주소

(1) 사용자 가상 주소(User Virtual Address): 사용자 공간에서 접근하는 주소이다. 사용자 프로세스에 접근하는 주소는 모두 가상 주소이다.

(2) 물리 주소(Physical Address): 프로세서와 메모리 사이에서 사용하는 실제로 물리적으로 존재하는 주소이다.

(3) 버스 주소(Bus Address): 주변장치와 메모리가 서로 데이터를 주고 받기 위해 사용하는 주소이다. 실제로 PCI(Peripgeral Component Interconnect, 주변장치와의 상호 데이터 전송)와 같은 경우 Memory-mapped IO를 수행하여 데이터를 읽거나 쓴다.

(4) 커널 논리 주소(Kernel Logical Address): 이 주소 공간은 물리 주소를 직접 참조하는 방식이다. 커널 논리 주소는 Low Memory영역에 있는 페이지들을 참조하며, 이 주소들은 물리 주소와 직접 일대일로 매핑된다. 즉, 물리 메로리 주소에 일정 값을 더함으로써 커널 논리 주소를 얻을 수 있다. 이 주소 공간은 커널이 Low Memory에 있는 데이터에 빠르게 접근할 수 있게 해주므로, 커널의 성능에 중요한 역할을 한다.

(5) 커널 가상 주소(Kernel Virtual Address): 이 주소 공간은 커널 논리 주소

공간을 포함하며, 추가적으로 High Memory영역에 위치한 페이지들을 참조할 수 있다. 커널 가상 주소는 페이지테이블을 통해 물리 메모리 주소와 매핑되며, 이 주소 공간은 커널이 전체 물리 메모리를 효과적으로 관리할 수 있게 해주며, 필요에 따라 메모리를 동적으로 할당하거나 해제하는데 사용된다.

▶ High Memory와 Low Memory

(1) Low Memory: 물리 메모리의 처음부터 약 896MB까지의 주소 범위를 나타내는 영역이며, 커널에 의해 직접 접근될 수 있는 메모리 영역으로 고정 매핑된 가상 주소를 통해 접근된다. 이 영역은 주로 코드, 데이터, BSS 영역과 같은 일반적인 커널 구성 요소들이 위치하는 공간이다.(커널 논리 주소가 존재하는 공간이다.)

(2) High Memory: Low Memory영역을 넘어서는 나머지 물리 메모리 주소 공간을 나타내며, 커널에 의해 직접 접근할 수 없고 페이지 테이블을 통해 간접적으로 접근된다. 커널 논리 주소가 존재하지 않고 커널 가상 주소 공간으로 할당되며, 사용자의 가상 주소 공간과는 구분되는 영역이다.

▶ 커널 동적 메모리 할당

(1) kmalloc

```
void *kmalloc(size_t size, gfp_t flags);
```

· 할당하려는 메모리 블록의 크기인 size, 메모리 할당에 대한 특정 옵션을 설정하는데 사용되는 flag 두 인수를 받아 동작하는 함수이다. 성공적으로 메모리를 할당하면 메모리 블록의 시작 주소를 반환하며, 실패하면 NULL을 반환한다.

· 할당된 메모리는 물리적으로 연속되며, 커널의 가상 주소 공간에서 접근할 수 있다. 이 메모리는 디바이스 드라이버와 같이 연속적인 메모리를 필요로 하는 커널의 서브시스템에서 주로 사용된다.

· 크기에 맞는 연속적인 메모리를 찾지 못하면 동적 할당에 실패하기 때문에 이런 경우에는 vmalloc함수를 이용한다.

```
void kfree(const void *ptr);
```

· 할당받은 메모리는 kfree함수를 사용하여 해제한다.

(2) vmalloc

```
void *vmalloc(unsigned long size);
```

· 할당하려는 메모리 블록의 크기를 바이트 단위로 입력받아 해당하는 크기의 메모리 블록을 가상 메모리 공간에서 할당하고, 할당된 메모리 블록의 시작 주소를 반환한다. 만약 메모리 할당에 실패하면 NULL을 반환한다.

· vmalloc으로 할당된 메모리는 커널의 가상 주소 공간에서 연속적이지만, 실제 물리 메모리에서는 연속적이지 않을 수 있다. 이는 페이지 테이블을 통해 각각의

가상 페이지를 물리 메모리의 다른 위치에 있는 페이지에 매핑했기 때문이다.

```
void vfree(const void *addr);
```

· 할당받은 메모리는 vfree함수를 사용하여 해제한다.

2) 가상 메모리, 메모리 매핑

▶ 리눅스와 같은 현대 운영체제에서 가상 메모리는 프로세스가 물리적 메모리를 직접 다루지 않고도 독립된 주소 공간을 갖게 해주는 중요한 메커니즘이다. 이를 통해 프로세스는 자신만의 고유한 주소 공간을 사용하는 것처럼 동작하며, 실제 물리적 메모리에 대한 접근은 운영체제가 관리한다.

▶ 가상 메모리의 주요 개념과 기능

(1) 프로세스 격리: 각 프로세스는 자신만의 가상 주소 공간을 갖게 되어, 다른 프로세스의 메모리 영역에 접근할 수 없다. 이를 통해 시스템 안정성이 유지된다.

(2) 메모리 보호: 운영체제는 페이지 테이블을 사용하여 각 페이지의 접근 권한(read, write, execute 등)을 제어한다. 이를 통해 잘못된 메모리 접근으로부터 시스템을 보호할 수 있다.

(3) 페이징과 스왑: 가상 메모리는 물리적인 RAM 크기보다 큰 용량으로 작동할 수 있다. 이때 필요하지 않은 페이지들은 스왑 영역(보조기억장치 영역)으로 내보내질 수 있으며(이를 스왑 아웃이라고 한다), 필요할 때 다시 RAM으로 가져올 수 있다(이를 스왑 인이라고 한다).

(4) 메모리 매핑(Memory Mapping): 파일이나 장치 등의 I/O 리소스를 가상 주소 공간에 매핑함으로써, 해당 리소스에 대한 읽기/쓰기 연산을 일반적인 메모리 접근처럼 처리할 수 있게 해준다.

- 예를 들어 mmap 시스템 콜은 파일을 가상 주소 공간에 매핑하는데 사용된다.
- 이렇게 매핑된 파일은 일반적인 read나 write 함수 대신에 그냥 포인터 연산으로 읽거나 쓸 수 있게 된다.

3) 페이징, 스위칭, 캐싱

(1) 페이징 (Paging): 페이징은 가상 메모리를 고정된 크기의 블록인 '페이지'로 나누는 메모리 관리 기법이다. 이를 통해 가상 주소 공간과 물리 주소 공간 사이의 매핑을 관리한다. 페이지는 일반적으로 4KB 크기를 갖지만, 대형 페이지(2MB나 1GB 등)를 지원하는 시스템도 있다.

(2) 스와핑 (Swapping): 메모리 관리 기법 중 하나로, 시스템의 물리적 메모리가 부족해지는 상황을 해결하기 위해 사용된다. 스위칭은 가상 메모리 관리의 핵심 기능 중 하나로, 물리 메모리와 디스크 간에 데이터를 교환(swap)하는 과정을 말한

다. 필요하지 않은 페이지들은 스왑 영역(보조기억장치 영역)으로 내보내질 수 있으며(이를 스왑 아웃이라고 한다), 필요할 때 다시 RAM으로 가져올 수 있다(이를 스왑 인이라고 한다).

(3) 캐싱 (Caching): 캐싱은 자주 사용되는 데이터나 연산 결과를 빠르게 접근 가능한 저장소에 보관하는 기법이다.

- 하드웨어 캐시: CPU 내부에 있는 L1, L2, L3 캐시 등이 이에 해당하며, 이들은 RAM보다 더 빠른 속도로 데이터에 접근할 수 있게 해준다.
- 소프트웨어 캐시: 운영체제나 애플리케이션 수준에서 구현되며, 파일 시스템 캐시(page cache), DNS lookup 결과 캐싱 등 다양한 형태가 있다.
- 페이지 캐시(Page Cache): 페이지 캐시는 파일 시스템의 데이터를 캐싱하는 데 사용된다. 파일로부터 읽은 데이터는 페이지 캐시에 저장되며, 이후 같은 데이터를 다시 읽을 필요가 있을 때는 페이지 캐시에서 빠르게 가져올 수 있다. 이로 인해 디스크 I/O를 크게 줄일 수 있다. 또한, 페이지 캐시는 파일 쓰기 연산에서도 사용된다. 즉시 디스크에 쓰지 않고, 먼저 페이지 캐시에 쓴 후 나중에 디스크에 반영하는 방식으로, 쓰기 연산의 효율성을 높인다.
- 버퍼 캐시(Buffer Cache): 버퍼 캐시는 파일 시스템의 메타데이터를 캐싱하는 데 사용된다. 메타데이터는 파일의 이름, 크기, 생성 시간, 수정 시간 등의 정보를 포함하며, 파일 시스템의 동작에 필수적이다. 버퍼 캐시를 통해 이러한 메타데이터의 접근 속도를 향상시킬 수 있다.

4) 프로세스 관리 및 스케줄링

▶ 프로세스 생성

· 기본 프로세스는 자체 주소 공간(fork)을 복제해서 새로운 구조를 생성한다.
· 모든 프로세스는 하위 프로세스를 생성 가능하다.
· 자식(하위) 프로세스는 보안 ID, 파일 설명자, 우선 순위, 환경 변수 등을 상속받는다.
· 부모 프로세스는 고유의 PID를 가진다. PPID는 생성된 자식 프로세스에서 부모 프로세스의 PID값을 나타낸다.
· 자식 프로세스는 fork()를 이용하여 생성하고, 이전의 프로세스를 마친 뒤 새로운 프로세스를 생성하려면 exec()를 사용한다.
· 부모 프로세스는 fork() 사용 후 자식 프로세스의 동작 완료까지 대기한다. 자식 프로세스는 작업 완료 후 부모 프로세스에 신호 전달 후 종료된다.

▶ 프로세스의 상태

· Running: CPU가 현재 실행하고 있는 상태

· Waiting (Interruptible): 특정 조건이 충족될 때까지 대기하고 있는 상태 (예: I/O 작업 완료)
· Waiting (Uninterruptible): 특정 조건이 충족될 때까지 대기하고 있으며, 시그널에 의해 깨어나지 않음
· Stopped: 일시 중단된 상태 (예: 사용자가 Ctrl+Z를 누름)
· 좀비 프로세스(Zombie): 프로세스가 종료될 때, 커널은 프로세스의 상태와 종료 코드를 저장하고, 프로세스를 '좀비'상태로 만든다. 이 정보는 부모 프로세스가 종료 상태를 조회할 수 있도록 남겨둔다. 부모 프로세스는 wait시스템 호출을 통해 자식 프로세스의 종료 상태를 확인하고, 그 정보를 회수할 수 있다. 하지만 부모 프로세스가 자식 프로세스의 종료 상태를 회수하지 않으면, 그 자식 프로세스는 계속해서 '좀비'상태에 머무르게 된다. 이 경우, 프로세스 테이블의 항목이 사용되고 있기 때문에 시스템 자원이 낭비될 수 있다. 해결방안으로서 만약 부모 프로세스가 먼저 종료되는 경우, init 프로세스(리눅스에서의 PID 1 프로세스)가 그 자식 프로세스를 '입양'하게 된다. init 프로세스는 주기적으로 wait시스템 호출을 실행하여 종료된 자식 프로세스를 회수한다. 이와 같이 '좀비' 상태를 해결할 수 있다.
· 고아 프로세스(Orphan): 부모 프로세스가 종료되고 자식 프로세스가 아직 실행중인 상태를 가리키는 용어이다. 이런 상황에서 자식 프로세스는 '고아'가 되며, 이후에는 init 프로세스(리눅스에서의 PID 1 프로세스)가 고아 프로세스의 '새로운 부모'가 되어 프로세스의 종료 상태를 처리한다. 이렇게 함으로써 시스템에서 '좀비 프로세스(Zombie Process)'가 남아있는 상황을 방지한다.
· 데몬 프로세스(Daemon): 데몬 프로세스는 백그라운드에서 실행되는 특별한 유형의 프로세스를 의미한다. 데몬은 사용자의 직접적인 조작 없이 특정 서비스를 제공하거나 주기적인 작업을 수행한다. 예를 들어, 웹 서버, FTP 서버, 크론 등이 데몬 프로세스의 예이다. 데몬 프로세스는 보통 시스템이 부팅될 때 시작되며, 종료될 때까지 계속 실행된다.

▶ 프로세스 종료

: 프로세스는 exit 시스템 호출을 통해 자신을 종료시킬 수 있다. 또한 부모 프로세스는 wait시스템 호출을 통해 자식 프로세스의 종료 상태를 회수할 수 있다.

▶ 프로세스 스케줄링

: 리눅스의 스케줄러는 어느 프로세서가 CPU의 실행 시간을 얻을 것인지 우선 순위를 정하기 위해 각 프로세스 마다 중요도에 따른 가중치를 부여한다. 여기에 사용되는 알고리즘은 'Completely Fair Scheduler(CFS)'라고 부르며, 각각의 태스크에게 가능한 한 공정하게 CPU 시간을 분배하는 것이 목표이다. 스케줄러는 다

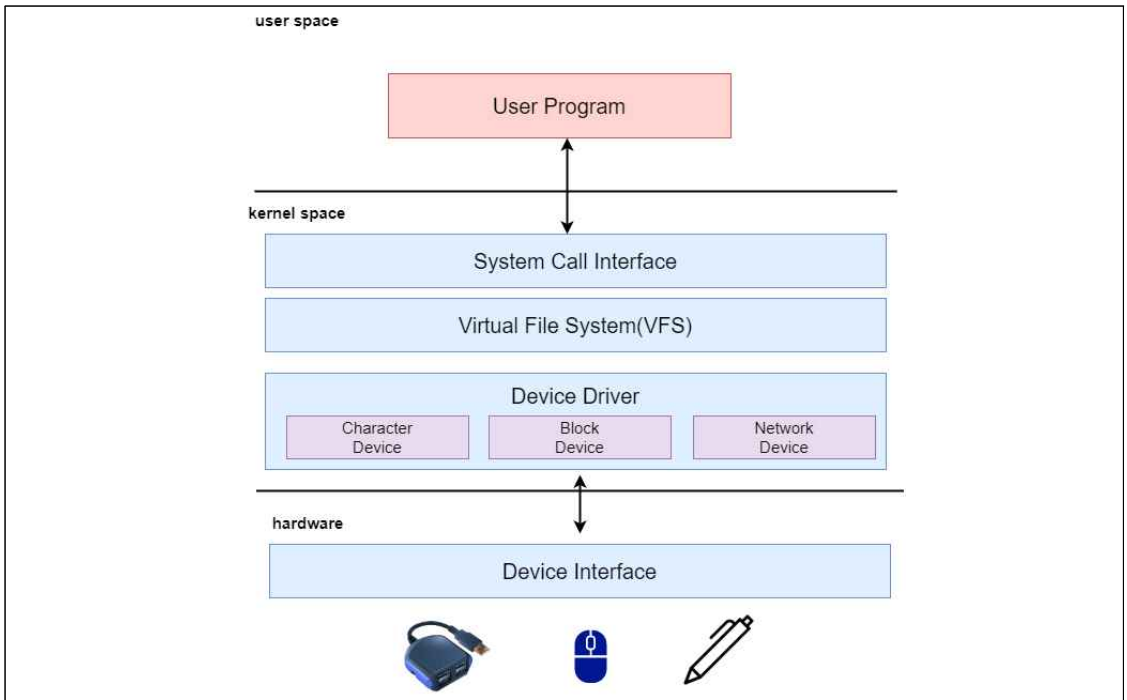
양한 요소들을 고려하여 결정을 내린다.

- Nice value: 사용자에 의해 설정되며, 낮은 값일수록 높은 우선 순위를 가진다.
- Priority: 커널이 설정하는 값으로서 실시간 태스크 등 특별한 경우에 사용된다.
- I/O-bound vs CPU-bound: I/O 작업이 많은 태스크와 계산 작업이 많은 태스크 사이에서 균형을 유지하기 위해 고려된다.
- CFS는 각 프로세스에 대해 '가상 런타임(Nice value)'이라는 값을 유지한다. 가상 런타임은 프로세스가 CPU를 얼마나 사용했는지를 나타내며, 작은 값을 가진 프로세스가 CPU를 더 많이 사용할 수 있도록 한다.
- 가상 런타임은 실제 런타임을 '프로세스의 가중치'로 나눈 값이다. 여기서 프로세스의 가중치는 프로세스의 우선순위(Priority)를 나타내며, 높은 우선순위를 가진 프로세스는 더 낮은 가상 런타임을 가진다. 이로써 높은 우선순위를 가진 프로세스가 더 많은 CPU시간을 할당받는다.
- CFS는 레드-블랙 트리라는 자료 구조를 사용하여 프로세스를 관리한다. 레드-블랙 트리는 노드(프로세스)를 삽입하거나 삭제하는데 O(log n)의 시간이 걸리는 이진 탐색 트리의 일종이다.(** O에 대해서는 알고리즘 전공서적 참조) CFS는 가장 낮은 가상 런타임을 가진 프로세스를 트리에서 찾아 CPU를 할당하고, 해당 프로세스의 가상 런타임을 업데이트하며, 이 프로세스를 적절한 위치로 다시 삽입한다.

4. 디바이스 관리

1) 디바이스 드라이버 구조

▶ 리눅스에서 디바이스 드라이버는 하드웨어 장치를 제어하고, 커널과 사용자 공간의 애플리케이션 사이에서 데이터를 주고받는 인터페이스 역할을 한다.



< 커널 시스템 구조 >

▶ 디바이스 드라이버의 주요 구성 요소

(1) 모듈: 대부분의 리눅스 디바이스 드라이버는 모듈 형태로 제공된다. 모듈은 커널에 동적으로 적재되거나 제거될 수 있는 코드 블록이다. 이렇게 함으로써 시스템은 필요한 기능만 메모리에 적재하고, 필요 없어진 기능은 메모리에서 해제하여 효율적인 리소스 관리가 가능하다.

· 커널에 모듈 적재 및 제거

명령어	설명
insmod	모듈을 커널에 적재한다.
rmmod	커널에서 모듈을 제거한다.
lsmod	커널에 적재된 모듈 목록을 출력한다.
depmod	모듈 간 의존성 정보를 생성한다.
modprobe	모듈을 커널에 적재하거나 제거한다.
mknod	디바이스 파일을 생성한다.

(2) 장치 파일: 리눅스에서 하드웨어 장치는 파일처럼 다룰 수 있다. /dev 디렉터리에 위치한 장치 파일을 통해 사용자 공간의 프로그램은 read(), write(), ioctl() 등의 시스템 호출을 이용해 하드워크 장치와 상호작용할 수 있다.

(3) 파일 연산 구조체 (file_operations): 각각의 디바이스 드라이버는 file_operations 구조체를 가지며, 이 안에는 open(), read(), write(), close() 등 다양한 콜백 함수 포인터가 정의되어 있다.

(4) ioctl(): ioctl() 시스템 호출은 표준 입력/출력 함수로 처리할 수 없는 복잡한

I/O 작업을 위해 사용된다.

(5) 문자 디바이스(Character Device): 문자 디바이스는 데이터를 바이트 단위로 처리한다. 이들은 대부분 입력/출력을 순차적으로 처리하며, 임의 접근을 지원하지 않는다. 키보드, 마우스 등의 입력 장치와 시리얼 포트 등은 대표적인 예시이다.

(6) 블록 디바이스(Block Device): 블록 디바이스는 데이터를 블록(일반적으로 몇 KB) 단위로 처리한다. 이들은 데이터를 저장하고 임의 접근할 수 있어야 하므로 주로 하드디스크나 SSD 같은 저장장치에 사용된다.

(7) 네트워크 디바이스(Network Device): 네트워크 디바이스 드라이버는 리눅스 시스템에서 네트워크 통신을 가능하게 하는 핵심적인 요소이다. 이 드라이버는 다양한 네트워크 인터페이스 카드에 대한 지원을 제공하며, 네트워크 통신을 위한 하드웨어 제어와 데이터 전송을 담당하며, 네트워크 통신의 안정성과 성능에 큰 영향을 미친다.

2) 디바이스 파일 시스템(devfs)

▶ 리눅스에서는 모든 것을 파일로 취급하는 '모든 것은 파일이다(Everything is a file)'라는 철학을 가지고 있다. 이 철학에 따라, 리눅스에서의 하드웨어 장치도 특수한 종류의 파일, 즉 디바이스 파일로써 시스템에 표현된다. 디바이스 파일은 주로 /dev 디렉터리 아래에 위치하며, 이를 통해 사용자 공간 프로그램이 커널 공간의 드라이버와 상호작용할 수 있게 된다.

▶ 디바이스 파일 유형

(1) 문자 디바이스 파일(Character Device Files): 문자 디바이스 드라이버를 위한 인터페이스를 제공한다. 데이터는 순차적인 바이트 스트림으로 처리되며, 장치가 데이터를 블록 단위로 처리하거나 임의 접근을 지원하지 않을 때 사용된다.

(2) 블록 디바이스 파일(Block Device Files): 블록 단위로 데이터를 읽고 쓸 수 있는 블록 디바이스 드라이버를 위한 인터페이스를 제공한다. 주요 용도는 저장장치(예: 하드디스크, SSD)와 같은 장치에 대한 접근이다.

3) 하드웨어 I/O

▶ 리눅스에서 하드웨어 I/O는 다양한 메커니즘을 통해 수행된다. 여기에는 시스템 호출, 디바이스 드라이버, 인터럽트, DMA(Direct Memory Access) 등이 포함된다.

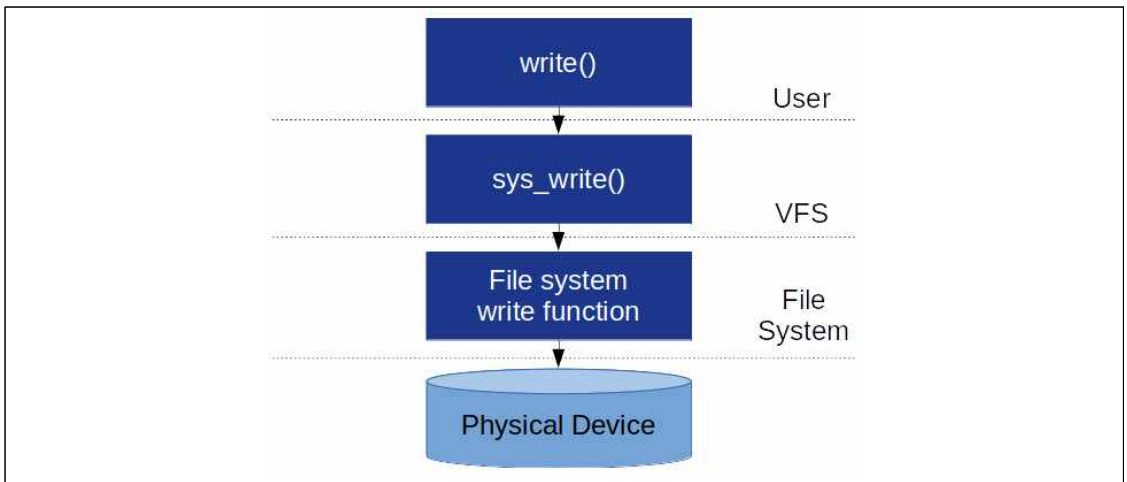
(1) 시스템 호출(System Call): 사용자 공간의 프로그램이 커널에 서비스를 요청하기 위해 사용하는 인터페이스이다. read(), write(), open(), close() 등의 시스템 호출은 파일 또는 디바이스와 상호작용하는 데 사용된다.

(2) 디바이스 드라이버(Device Driver): 하드웨어 장치를 제어하고, 커널과 사용자 공간의 애플리케이션 사이에서 데이터를 주고받는 역할을 한다. 각각의 디바이스 드라이버는 해당 하드웨어 장치의 세부 사항을 알고 있으며, 이를 통해 커널은 장치에 대한 구체적인 지식 없이도 일관된 방식으로 I/O 작업을 수행할 수 있다.
(3) 인터럽트(Interrupt): CPU가 프로그램 실행 중에 외부 이벤트(예: 하드웨어에서 데이터가 준비되었음)에 응답하도록 하는 메커니즘이다. 인터럽트가 발생하면 CPU는 현재 실행 중인 작업을 중단하고 인터럽트 처리 루틴(일반적으로 디바이스 드라이버에 의해 제공)을 실행한다.
(4) DMA(Direct Memory Access): CPU를 거치지 않고 직접 메모리와 하드웨어 장치 간에 데이터 전송을 가능하게 하는 기능이다. DMA를 통해 대량의 데이터 전송 시 CPU 부하를 크게 줄일 수 있다.

5. 파일 시스템

1) 가상 파일시스템 (VFS)

▶ 리눅스의 가상 파일 시스템(VFS, Virtual File System)은 여러 종류의 실제 파일 시스템(예: ext4, FAT32, NTFS 등)에 대한 일관된 인터페이스를 제공하는 추상화 계층이다. 이를 통해 사용자 공간 애플리케이션은 파일 시스템의 세부 구현이나 타입을 알 필요 없이 일관된 방식으로 파일 I/O 작업을 수행할 수 있다.



< VFS의 접근 구조 >

▶ VFS의 주요 기능

(1) 일관된 파일 시스템 인터페이스 제공: 파일 시스템의 종류(ext4, XFS, Btrfs, FAT 등)에 따라 제공하는 인터페이스는 각각 다르지만, VFS를 사용하면 파일 시

스템의 종류에 상관 없이 사용자는 동일한 명령과 API를 사용하여(ex. open(), read(), write(), close() 등) 파일에 접근할 수 있다.
(2) 다양한 파일 시스템 지원 : VFS는 리눅스에서 사용되는 다양한 파일 시스템을 지원한다. ext4, XFS, Btrf, FAT, NTFS 등의 파일 시스템을 지원한다.
(3) 파일 시스템 독립적인 오퍼레이션 지원: VFS는 파일 시스템 독립적인 오퍼레이션을 지원한다. 이는 파일 및 디렉토리의 생성, 삭제, 읽기, 쓰기 등의 기본적인 파일 오퍼레이션부터 메타데이터 조회, 파일 권한 관리 등의 복잡한 오퍼레이션까지 포함된다.
▶ VFS는 모든 파일 시스템에 공통적인 객체와 연산을 정의하며, 이에는 아래와 같은 것들이 포함된다.
(1) Superblock: 각각의 마운트된 파일 시스템을 대표하는 객체로서, 해당 파일 시스템에 대한 메타데이터(예: 루트 디렉터리 위치, 전체 블록과 inode 수 등)를 저장한다.
(2) Inode: 각각의 파일이나 디렉터리를 대표하는 객체로서, 해당 객체에 대한 메타데이터(예: 소유자 ID, 접근 권한, 생성/수정/접근 시간 등)와 데이터 블록 위치 정보를 저장한다.
(3) Dentry: 디렉터리 엔트리를 나타내는 객체로서, 이름과 해당 이름과 연결된 inode 사이의 매핑을 저장한다.
(4) File: 열린 파일을 나타내는 객체로서, 현재 읽기/쓰기 위치와 관련 상태 정보 등을 저장한다.
▶ VFS의 주요 구성 요소
(1) VFS 객체: VFS는 파일, 디렉토리, 링크 등의 파일 시스템 객체를 표현하는 여러 가지 VFS 객체를 정의한다. 이에는 inode 객체, file 객체, dentry 객체, superblock 객체 등이 포함된다.
(2) VFS 시스템 콜: VFS는 open, read, write, close 등의 시스템 콜을 제공한다. 이 시스템 콜은 사용자 응용 프로그램이 VFS 객체에 접근하는 데 사용되며, 실제 파일 시스템의 오퍼레이션을 호출하는 역할을 한다.
(3) VFS 오퍼레이션: VFS는 실제 파일 시스템의 오퍼레이션을 호출하는 인터페이스를 정의한다. 이 인터페이스는 실제 파일 시스템에 따라 구현된다.

2) LVM과 RAID

▶ LVM(Logical Volume Manager)과 RAID(Redundant Array of Independent Disks)는 모두 디스크 저장 공간을 관리하는 데 사용되는 기술이다. 그러나 각각은 다른 목적과 특성을 가지고 있다.

▶ LVM (Logical Volume Manager)

: LVM은 하드 드라이브를 물리적 볼륨(Physical Volumes, PVs)으로 분할하고, 이들을 볼륨 그룹(Volume Groups, VGs)으로 묶어서 하나의 큰 스토리지 풀을 만들어서 여러 개의 디스크를 하나의 논리적 공간으로 보이도록 한다. 이 스토리지 풀에서 원하는 크기와 속성으로 논리적 볼륨(Logical Volumes, LVs)을 생성할 수 있다. LVM의 주요 장점은 유연성이다. LV의 크기는 필요에 따라 증가시키거나 줄일 수 있으며, VG에 새로운 PV를 추가하거나 제거하여 전체 스토리지 용량을 동적으로 조정할 수 있다. 또한 스냅샷 기능 등 다양한 고급 기능도 제공한다.

▶ LVM의 주요 구성 요소

(1) Physical Volume (PV): 물리적 볼륨은 디스크나 디스크 파티션을 의미합니다. 이들은 LVM이 관리하는 기본 저장 공간으로 사용됩니다.

(2) Volume Group (VG): 볼륨 그룹은 하나 이상의 물리적 볼륨으로 구성된다. 볼륨 그룹은 큰 논리적 디스크로 볼 수 있으며, 이 안에 여러 개의 논리적 볼륨을 생성할 수 있다.

(3) Logical Volume (LV): 논리적 볼륨은 볼륨 그룹 내에서 생성되며, 실제 파일 시스템이 위치하는 곳이다. 논리적 볼륨은 필요에 따라 용량을 증가시키거나 줄일 수 있으며, 여러 물리적 볼륨에 걸쳐 있을 수 있다.

▶ LVM의 주요 기능

(1) 용량 관리: LVM을 사용하면 용량을 쉽게 확장하거나 축소할 수 있다. 예를 들어, 논리적 볼륨에 공간이 부족하면 볼륨 그룹에 새로운 물리적 볼륨을 추가하고 논리적 볼륨을 확장할 수 있다.

(2) 스냅샷: LVM은 논리적 볼륨의 스냅샷을 생성하는 기능을 제공한다. 이 기능은 논리적 볼륨의 특정 시점의 상태를 캡처하여 백업하거나 테스트에 사용할 수 있다.

(3) 스트라이핑과 미러링: LVM은 물리적 볼륨을 스트라이핑하거나 미러링하는 기능을 제공한다. 스트라이핑은 데이터를 여러 물리적 볼륨에 분산하여 저장하는 방법으로, I/O 성능을 향상시킬 수 있다. 미러링은 데이터를 여러 물리적 볼륨에 복제하여 저장하는 방법으로, 데이터의 안정성을 높일 수 있다.

▶ RAID (Redundant Array of Independent Disks): RAID는 여러 개의 독립적인 디스크를 하나의 논리적 단위로 묶어서 사용하는 기술이다. RAID는 데이터의 안정성과 입출력 성능을 향상시키기 위해 사용된다. 리눅스에서는 소프트웨어 RAID와 하드웨어 RAID 두 가지 방식을 지원한다. 소프트웨어 RAID는 운영 체제의 기능을 활용해 RAID를 구성하며, 하드웨어 RAID는 전용 컨트롤러 장치를 사용해 RAID를 구성한다. RAID에는 여러 가지 레벨이 있으며, 각 레벨은 데이터를 디스크에 저장하는 방식에 따라 다르다.

(1) RAID 0 (Striping): 데이터를 여러 디스크에 분산해서 저장한다. 이 방식은 입출력 성능은 향상시키지만, 어느 하나의 디스크만 고장나도 모든 데이터를 잃게 되므로 안정성은 떨어진다.
(2) RAID 1 (Mirroring): 동일한 데이터를 두 개 이상의 디스크에 복제해서 저장한다. 이 방식은 안정성은 높이지만, 저장 공간 효율이 떨어진다.
(3) RAID 5 (Striping with Parity): 데이터와 패리티 정보를 교차하여 여러 디스크에 저장한다. 이 방식은 입출력 성능과 안정성을 모두 향상시키지만, 패리티를 계산하고 저장하는 데 추가적인 작업이 필요하다.
(4) RAID 6 (Striping with Double Parity): RAID 5와 비슷하지만, 두 개의 패리티 정보를 저장한다. 이 방식은 두 개의 디스크가 동시에 고장나도 데이터를 복구할 수 있는 높은 안정성을 제공한다.
(5) RAID 10 (Striping + Mirroring): RAID 0과 RAID 1을 결합한 방식으로, 데이터를 복제한 후에 분산해서 저장한다. 이 방식은 높은 입출력 성능과 안정성을 동시에 제공한다.

3) JFS

▶ JFS(Journaled File System)는 IBM에서 개발한 64비트 저널링 파일 시스템이다. JFS는 원래 IBM의 AIX 운영 체제를 위해 개발되었지만, 후에 리눅스에도 포팅되었다.

▶ JFS의 주요특징

(1) 저널링: JFS는 메타데이터 변경을 추적하는 저널을 유지하여 시스템 충돌이나 전원 중단 등 비정상 종료 후 빠르게 복구할 수 있다.
(2) B+ 트리 인덱싱: JFS는 디렉터리와 파일의 데이터 블록을 관리하기 위해 B+ 트리 구조를 사용한다. 이로 인해 대용량 파일과 디렉터리에서도 높은 성능을 유지할 수 있다.
(3) 디스크 공간 관리: JFS는 extents라는 기법을 사용하여 연속된 데이터 블록을 하나의 단위로 처리한다. 이를 통해 디스크 공간 활용성이 향상되고, 큰 파일에 대한 I/O 성능이 개선된다.
(4) 동시성 제어: JFS는 여러 쓰레드가 동시에 파일 시스템 작업을 수행할 때 발생할 수 있는 충돌 문제를 방지하기 위한 잠금 메커니즘을 제공한다.
(5) 64비트 지원: JFS는 64비트 주소 공간을 지원하므로 매우 큰 파일과 파티션(최대 4 PB)을 처리할 수 있다.

6. 네트워크

1) 멀티플렉싱과 디멀티플렉싱

▶ 리눅스 네트워킹에서의 멀티플렉싱과 디멀티플렉싱은 여러 데이터 소스와 목적지 사이에서 데이터를 전송하는 과정을 관리하는 방법을 말한다.

(1) 멀티플렉싱(Multiplexing): 멀티플렉싱은 여러 데이터 스트림을 하나의 연결(예: 네트워크 소켓)로 결합하는 과정이다. 이는 주로 운영 체제의 네트워크 스택에서 처리되며, 각각의 데이터 스트림은 동일한 목적지에 전송될 수 있다. 리눅스에서는 여러 가지 방식으로 네트워크 멀티플렉싱을 지원한다. 'select', 'poll', 'epoll'등의 시스템 콜을 사용하여 여러 파일 디스크립터(주로 소켓)를 동시에 모니터링하고, 이 중에서 I/O 작업이 가능한 디스크립터를 찾아낸다. 이를 통해 하나의 프로세스가 여러 네트워크 연결을 동시에 처리할 수 있다. 멀티플렉싱의 주요장점으로는 블로킹(Blocking)을 방지하는 것이다. 예를 들어, 서버가 여러 클라이언트로부터의 연결 요청을 처리해야 하는 상황을 생각해 봤을 때 멀티플렉싱이 없다면, 서버는 각 클라이언트로부터의 요청을 순차적으로 처리해야 하므로, 한 클라이언트로부터의 요청 처리가 끝날 때까지 다른 클라이언트들은 대기 상태로 머물게 된다. 이는 시스템의 전체적인 처리 성능을 저하시킨다. 그러나 멀티플렉싱을 사용하면, 서버는 여러 클라이언트로부터 동시에 데이터를 받아들일 수 있다. 이는 각 클라이언트로부터의 요청을 독립적으로, 그리고 동시에 처리할 수 있음을 의미한다. 따라서 멀티플렉싱은 시스템의 처리 성능을 향상시키고, 자원 활용도를 높이는 데 효과적이다.

(2) 디멀티플렉싱(Demultiplexing): 리눅스에서의 디멀티플렉싱(demultiplexing)은 멀티플렉싱(multiplexing)의 반대 과정으로, 하나의 통신 채널로부터 여러 개의 데이터 스트림을 분리하는 작업을 의미한다. 이는 주로 네트워크 통신에서 발생하는 여러 연결을 관리하거나, 동시에 여러 데이터 스트림을 처리하는 데 사용된다. 예를 들어, 서버에서는 한 번에 여러 클라이언트로부터의 연결 요청을 받을 수 있다. 이때 서버는 각 연결 요청을 별도의 소켓에 연결하고, 이들 소켓을 디멀티플렉싱하여 각 클라이언트로부터의 데이터 스트림을 분리한다. 이렇게 분리된 데이터 스트림은 각각 독립적으로 처리될 수 있으며, 이는 서버가 동시에 여러 클라이언트와 통신할 수 있게 해준다. 리눅스에서는 'select', 'poll', 'epoll' 등의 시스템 콜을 통해 디멀티플렉싱을 수행할 수 있다. 이들 시스템 콜은 여러 파일 디스크립터 중에서 I/O 작업이 가능한 디스크립터를 찾아내는 역할을 한다. 디스크립터가 준비된 상태가 되면, 해당 디스크립터로부터 데이터를 읽어오거나 데이터를 쓸 수 있다. 디멀티플렉싱을 사용하면 하나의 프로세스나 스레드가 여러 I/O 작업을 동시에 처리할 수 있으므로, 시스템의 성능을 향상시키고 자원을 더 효율적으로 활용할 수 있다. 또한,

디멀티플렉싱을 통해 비동기 I/O 작업을 수행하면, I/O 작업이 블로킹되는 시간을 줄이고 CPU 사용률을 최적화할 수 있다.

2) 리눅스 TCP/IP 스택

▶ 프로토콜(Protocol)

: 컴퓨터 네트워크에서 컴퓨터나 장치 간의 통신을 위해 정의된 규칙과 규약의 집합을 말한다. 이 규칙들은 데이터의 형식, 전송 방식, 오류 제어, 보안 등의 다양한 측면을 다루며, 효율적이고 안정적인 통신을 가능하게 한다. 각각의 프로토콜은 고유한 식별자를 가지며, 이를 통해 통신에 참여하는 장치들은 어떤 프로토콜을 사용해야 하는지 알 수 있다. 프로토콜의 종류로는 TCP(Transmission Control Protocol)/IP(Internet Protocol) HTTP(Hypertext Transfer Protocol), FTP(File Transfer Protocol), SMTP(Simple Mail Transfer Protocol), DNS(Domain Name System) 등이 있다.

▶ TCP(Transmission Control Protocol)

: 인터넷 프로토콜 스택인 TCP/IP의 주요 프로토콜 중 하나로, 데이터를 패킷 단위로 나누어 인터넷을 통해 전송하고, 이를 원래의 데이터로 다시 조립하는 역할을 한다. TCP는 다음과 같은 특징을 가진다.

(1) 연결 지향성: TCP는 통신을 시작하기 전에 송신자와 수신자 간에 연결을 설정한다. 이를 통해 데이터가 정확하게 전송될 수 있도록 한다.

(2) 신뢰성: TCP는 패킷의 순서 보장, 에러 검출 및 재전송, 흐름 제어 등을 통해 데이터의 신뢰성 있는 전송을 보장한다. 만약 패킷이 손상되었거나 분실되면, TCP는 해당 패킷을 재전송한다.

(3) 흐름 제어 및 혼잡 제어: TCP는 송신자와 수신자의 데이터 처리 속도 차이를 해결하기 위해 흐름 제어를 제공하며, 네트워크의 혼잡 상황을 관리하기 위해 혼잡 제어 메커니즘을 가지고 있다.

▶ IP(Internet Protocol)

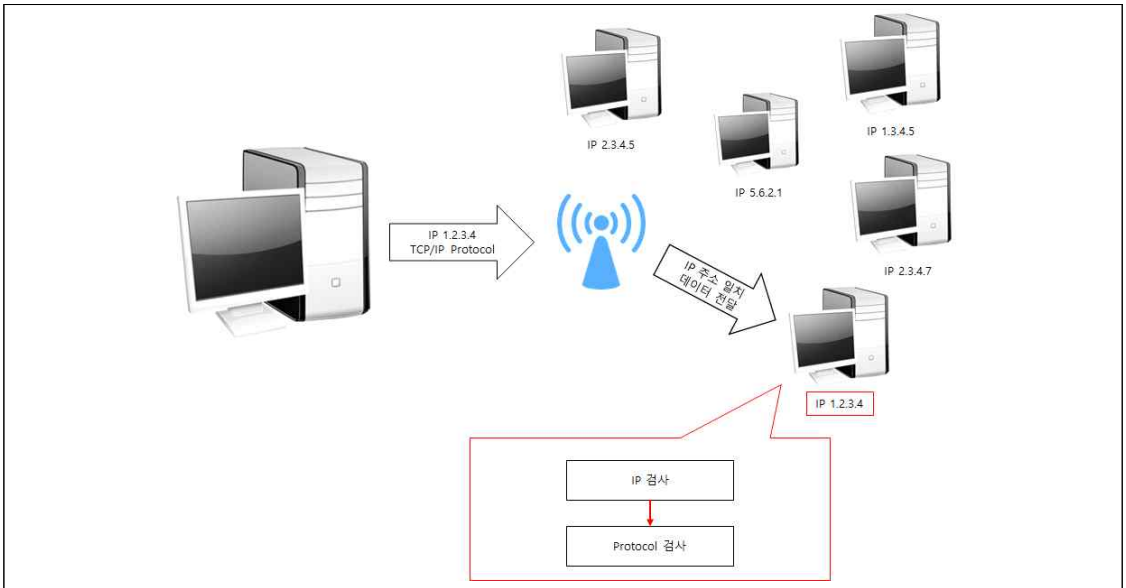
: 인터넷 프로토콜 스택인 TCP/IP에서 가장 핵심적인 프로토콜 중 하나로, 네트워크를 통해 데이터 패킷을 전송하는 방법을 정의한다. IP의 주요 역할은 데이터 패킷을 송신지에서 목적지까지 올바르게 전달하는 것이다. 이를 위해 IP는 각각 고유의 주소를 가지며 각 패킷에 대한 메타데이터를 포함하는 IP 헤더를 사용한다. IP 헤더에는 다음과 같은 정보가 포함된다.

(1) 버전: 사용된 IP 프로토콜의 버전을 나타낸다. 현재는 IPv4와 IPv6가 주로 사용된다.

(2) 소스 IP 주소: 패킷을 보낸 장치의 IP 주소이다.

(3) 목적지 IP 주소: 패킷을 받을 장치의 IP 주소이다.

(4) TTL(Time To Live): 패킷이 네트워크에 머무를 수 있는 최대 시간을 나타낸다. 이 값은 패킷이 무한히 순환하는 것을 방지하는 데 사용된다.|



< TCP/IP의 데이터 전송 >

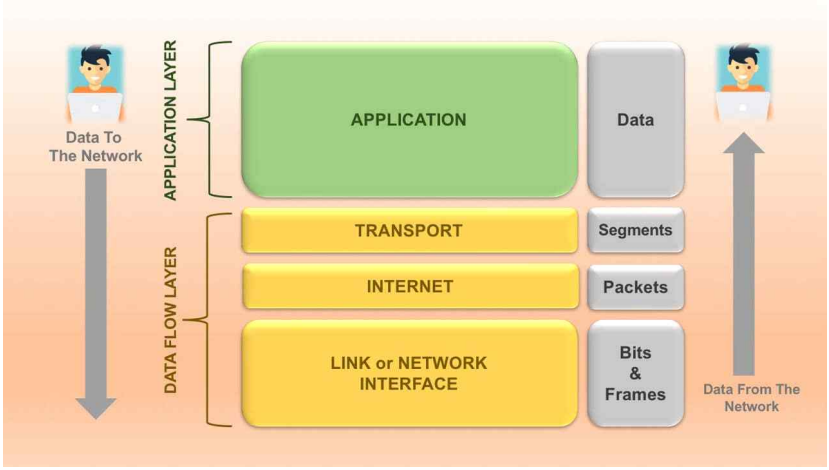
: IP는 데이터가 찾아갈 주소, TCP는 찾아온 데이터가 형식에 맞게 왔는지 확인하는 필터라고 볼 수 있다.

▶ 라우터(Router)와 라우팅(Routing)

(1) 라우터(Router): 네트워크에서 패킷을 전송하는 장치로, 한 네트워크와 다른 네트워크 간에 정보를 주고받는 역할을 한다. 라우터는 IP주소를 기반으로 패킷의 소스와 목적지를 파악하고, 이를 바탕으로 패킷을 올바른 방향으로 전달한다.

(2) 라우팅(Routing): 라우터가 패킷을 어떻게 처리하고 전달할지 결정하는 과정을 의미한다. 이 과정에서 라우터는 여러 가지 요인을 고려하여 패킷을 목적지까지 가장 효과적으로 전달하는 경로를 선택한다.

▶ TCP/IP 스택은 데이터 통신을 가능하게 하는 프로토콜들의 집합으로, 이들은 계층적인 구조로 되어 있다. 리눅스 운영체제는 네트워크 통신을 위한 TCP/IP 스택을 내장하고 있다. 리눅스 TCP/IP는 계층적 구조를 통해 데이터가 송신자에서 수신자까지 안전하고 효율적으로 전달될 수 있도록 도와준다. 리눅스의 TCP/IP 스택은 크게 4개의 계층으로 나눌 수 있다.



< TCP/IP 4계층 >

TCP/IP 4계층	역할	데이터 단위	전송 주소	예시	장비
응용 계층 (Application Layer)	응용프로그램간의 데이터 송수신	Data/ Message	-	파일 전송, 이메일,FTP,HTTP,SMTP등	-
전송 계층 (Transport Layer)	호스트간의 송수신	Segment	Port	TCP,UDP 등	게이트웨이
인터넷 계층 (Internet Layer)	데이터 전송을 위한 논리적 주소 및 경로 지정	Packet	IP	IP,ARP,ICMP,RARP등	라우터
네트워크 계층 (Network Layer)	실제 데이터인 프레임 송수신	Frame	MAC	Ethernet등	브릿지,스위치

(1) 응용 계층(Application Layer): 사용자와 가장 가까운 계층으로 사용자-소프트웨어 간 소통을 담당하며, 웹 프로그래밍을 하면서 흔히 접하는 여러 서버나 클라이언트 관련 응용 프로그램들이 동작하는 계층이다. 주로 응용 프로그램들끼리 데이터를 교환하기 위해 사용된다.

(2) 전송 계층(Transport Layer): 통신 노드 간의 데이터 전송 및 흐름에 있어 신뢰성을 보장한다. 이는 다시 말해 데이터를 적절한 애플리케이션에 제대로 전달되도록 배분함을 의미한다. 전송 계층의 대표적인 프로토콜로는 TCP와 UDP가 있다. TCP는 연결 지향형 프로토콜로 패킷에 하나의 오류라도 있으면 재전송을 위해 에러를 복구하는 반면, UDP는 패킷을 중간에 잃거나 오류가 발생해도 이에 대처

하지 않고 계속해서 데이터를 전송하는 TCP에 비해 간단한 구조를 가지는 프로토콜이다.

(3) 인터넷 계층(Internet Layer): 네트워크 상에서 데이터의 전송을 담당하는 계층으로 서로 다른 네트워크 간의 통신을 가능하게 하는 역할을 수행한다(연결성 제공). 단말을 구분하기 위해 논리적인 주소로 IP주소를 할당하게 되고 이 IP주소로 네트워크 상의 컴퓨터를 식별하여 주소를 지정할 수 있도록 해준다.

네트워크끼리 연결하고 데이터를 전송하는 기기를 '라우터'라고 하며, 라우터에 의한 네트워크 간의 전송을 '라우팅'이라고 한다. 이 라우터가 내부의 라우팅 테이블(Routing Table)을 통해 경로 정보를 등록하여 데이터 전송을 위한 최적의 경로를 찾는데, 이렇게 출발지와 목적지 간의 데이터 전송 과정을 가리켜 End-to-End 통신이라고 부른다.

(4) 네트워크 인터페이스 계층(Network Interface Layer): 물리적인 데이터의 전송을 담당하는 계층으로, 여기서는 인터넷 계층과 달리 같은 네트워크 안에서 데이터가 전송된다. 노드 간의 신뢰성 있는 데이터 전송을 담당하며, 논리적인 주소가 아닌 물리적인 주소인 MAC을 참조해 장비간 전송을 하고, 기본적인 에러 검출과 패킷의 Frame화를 담당한다.

▶ 리눅스 TCP/IP 스택은 POSIX API를 따르므로 네트워크 연결 설정 및 데이터 전송 등을 위한 시스템 호출(setsockopt(), socket(), bind(), listen(), accept(), connect(), send(), receive() 등)을 제공한다.

< 시스템 및 네트워크 프로그래밍 >

1. 프로세스 및 파일 처리

1) fork, exec 계열

▶ 'fork'와 'exec' 계열은 리눅스 및 유닉스 기반 시스템에서 프로세스 생성과 실행에 사용되는 시스템 호출입니다. 이들은 운영 체제의 핵심 기능 중 하나인 프로세스 관리에 사용됩니다.

▶ fork

: 'fork'는 리눅스와 유닉스 기반 운영 체제에서 사용하는 시스템 호출 중 하나로, 현재 실행 중인 프로세스(부모 프로세스)의 복사본을 생성(즉, "포크")하는 데 사용된다. 이렇게 생성된 복사본을 '자식 프로세스'라고 한다.

'fork' 시스템 호출이 성공하면, 부모 프로세스와 자식 프로세스는 거의 동일한 상태를 가지게 된다. 자식 프로세스는 부모 프로세스의 메모리 공간과 환경 변수, 열린 파일 디스크립터 등을 복사하여 가지게 된다. 이는 단순히 참조 카운트를 증가시키는 것이 아니라, 부모 프로세스의 주소 공간을 완전히 복제하는 '복사-쓰기' 방식을 사용한다.

▶ 'fork' 시스템 호출은 성공하면 부모 프로세스에게는 자식 프로세스의 PID를 반환하고, 자식 프로세스에게는 0을 반환한다. 이를 통해 부모 프로세스와 자식 프로세스의 코드 흐름을 분리할 수 있다. 예를 들어, 'fork' 이후에 반환값이 0인지 아닌지를 체크함으로써 프로세스가 부모인지 자식인지를 판단할 수 있다.

'fork' 시스템 호출은 새로운 프로세스를 생성하는 가장 일반적인 방법이며, 셸에서 사용자가 명령을 실행할 때나 프로세스가 자신의 작업을 병렬로 수행하려 할 때 주로 사용된다.

▶ 부모 프로세스와 자식 프로세스의 차이점

(1) 프로세스 ID(PID): 부모 프로세스와 자식 프로세스는 각각 다른 PID를 가진다.

(2) 부모 프로세스 ID(PPID): 자식 프로세스의 PPID는 부모 프로세스의 PID이다.

(3) 자원 사용량: 부모 프로세스와 자식 프로세스의 자원 사용량은 독립적으로 계산된다.

(4) 알람과 타이머: 부모 프로세스에서 설정한 알람이나 타이머는 자식 프로세스에게 상속되지 않는다.

▶ fork 계열

(1) vfork: 'vfork'는 'fork'와 유사하지만, 부모와 자식이 같은 메모리 공간을 공유하도록 하여 메모리 사용량을 줄인다. 그러나 이는 동시성 문제를 야기할 수 있기 때문에, 'vfork'를 호출한 후에는 'exec'나 '_exit'를 호출하여 새로운 프로그램을 시작하거나 프로세스를 종료해야 한다.
(2) clone: 'clone'는 리눅스 특유의 시스템 호출로, 프로세스 생성에 있어서 더욱 세밀한 제어를 가능하게 한다. 'clone'은 어떤 부분을 공유하고 어떤 부분을 복사할지를 지정할 수 있어, 쓰레드 생성 등에 사용될 수 있다.

▶ exec

: 'exec'는 Unix와 Linux에서 사용하는 시스템 호출로, 현재 프로세스의 메모리 공간에 새로운 프로그램을 로드하고 실행하는 역할을 한다. 'exec' 시스템 호출이 성공하면, 원래 프로세스의 이미지는 새로운 프로그램의 이미지로 완전히 대체되며, 새로운 프로그램의 실행을 시작한다.
'exec'는 프로세스가 자신의 메모리 공간에 다른 프로그램을 로드하고 실행하도록 하는 유일한 방법이며, 프로세스가 다른 프로그램을 실행하면서도 자신의 PID를 유지하게 된다.

▶ exec 계열

(1) execl: 이 함수는 프로그램의 경로와 프로그램에 전달할 인수들을 명시적으로 지정(list 형태)한다. 인수는 NULL 문자로 종료되어야 한다.
(2) execv: 이 함수는 프로그램의 경로와 인수 배열을 인자로(vector 형태) 받는다. 인수 배열은 NULL 포인터로 종료되어야 한다.
(3) execle: 이 함수는 'execl'과 비슷하지만, 추가적으로 환경 변수를 지정할 수 있다.
(4) execve: 이 함수는 'execv'와 비슷하지만, 추가적으로 환경 변수를 지정할 수 있다.
(5) execlp: 이 함수는 프로그램의 이름만을 인자로 받고, 시스템의 PATH 환경 변수를 기반으로 프로그램을 찾는다.
(6) execvp: 이 함수는 'execlp'와 비슷하지만, 인수를 배열로 받는다.

2) 저수준과 고수준 파일 핸들링

▶ 파일 핸들링은 컴퓨터 프로그래밍에서 중요한 개념으로, 파일을 생성, 읽기, 쓰기, 닫기 등의 작업을 수행하는 것을 의미한다. 이러한 작업은 일반적으로 저수준과 고수준으로 나눠진다.

▶ 파일 디스크립터(File Descriptor)

: 운영 체제가 열린 파일을 추적하고 관리하기 위해 사용하는 정수 값이다. 파일

디스크립터는 프로세스가 파일을 열었을 때 운영 체제에 의해 할당된다. 이 파일 디스크립터는 이후에 파일을 읽거나 쓰거나 닫는 등의 작업을 수행하는데 사용된다. 이러한 작업은 모두 저수준 파일 I/O 시스템 호출에 의해 수행되며, 이러한 시스템 호출들은 파일 디스크립터를 인자로 받는다.
유닉스 계열에서 표준 입력 표준 출력, 표준 에러 출력은 각각 0(stdin), 1(stdout), 2(stderr)로 값이 이미 정해져 있다.

▶ 저수준 파일 핸들링

: 저수준 파일 핸들링은 운영 체제가 제공하는 시스템 호출을 직접 사용하여 파일을 처리하는 방식이다. 이 방식은 파일을 바이트 단위로 읽고 쓸 수 있으며, 파일의 특정 위치로 이동하는 등의 세밀한 제어가 가능하다.
리눅스 및 유닉스에서는 'open', 'read', 'write', 'lseek', 'close' 등의 시스템 호출이 주로 사용된다. 이러한 시스템 호출은 파일 디스크립터라는 정수 값을 사용하여 파일을 참조하며, 이는 운영 체제가 파일을 관리하는 데 사용하는 내부적인 식별자이다. 저수준 파일 핸들링은 일반 파일 뿐 아니라 특수 파일(데이터 전송이나 디바이스 접근에서 사용하는 파일)도 다룰 수 있다.
저수준 파일 핸들링은 세밀한 제어가 가능하다는 장점이 있지만, 직접적으로 시스템 호출을 사용하므로 복잡하고 오류가 발생하기 쉽다.

▶ 고수준 파일 핸들링

: 고수준 파일 핸들링은 프로그래밍 언어가 제공하는 라이브러리 함수를 사용하여 파일을 처리하는 방식이다. 이 방식은 파일을 문자나 문자열, 혹은 더 복잡한 데이터 구조로 쉽게 읽고 쓸 수 있다.
C 언어에서는 'fopen', 'fread', 'fwrite', 'fseek', 'fclose' 등의 함수가 사용된다. 이러한 함수들은 FILE 포인터라는 추상적인 파일 핸들을 사용하여 파일을 참조한다.
고수준 파일 핸들링은 사용하기 쉽고 안전하지만, 저수준 파일 핸들링에 비해 제어의 세밀성이 떨어질 수 있다.
결국, 저수준과 고수준 파일 핸들링 방식 중 어느 것을 사용할지는 프로그램의 요구 사항과 프로그래머의 선호도에 따라 결정된다.

분류	저수준 파일 입출력	고수준 파일 입출력
사용수준	비직관적	직관적(사용이 쉽다)
속도	고수준 보다 빠르다	저수준 보다 느리다
I/O 유형	바이트 단위로 읽고 쓴다	버퍼 단위로 읽고 쓴다
지원	특수 파일에 대한 접근 가능	여러 가지 형식을 지원
사용 함수	open,close,read,write 등	fopen,fclose,fread,fwrite 등

2. 메모리

1) 메모리 할당 및 해제

▶ malloc

```
void* malloc(size_t size);
```

'malloc'은 'size_t' 타입의 하나의 인자를 받아, 그 크기만큼의 연속된 바이트를 할당하고, 할당된 메모리 블록의 첫 번째 바이트를 가리키는 포인터를 반환한다. 이 반환된 포인터는 'void*' 타입이므로, 필요에 따라 적절한 데이터 타입으로 형변환할 수 있다.

만약 메모리 할당에 실패하면, 'malloc'은 NULL을 반환한다. 따라서 'malloc'을 호출한 후에는 항상 반환값을 확인하여 메모리 할당이 성공적으로 이루어졌는지 확인하는 것이 좋다.

할당된 메모리는 프로그램이 명시적으로 'free' 함수를 호출하여 해제하기 전까지 계속 유지된다. 'malloc'으로 할당받은 메모리를 사용한 후에는 반드시 'free'를 호출하여 메모리를 해제해야 한다. 그렇지 않으면 메모리 누수가 발생할 수 있다.

'malloc'으로 할당받은 메모리의 초기값은 정의되어 있지 않다. 따라서 초기화되지 않은 메모리를 읽으려고 하면 예상치 못한 결과가 발생할 수 있다. 이러한 이유로, 'malloc'으로 할당받은 메모리는 사용하기 전에 적절히 초기화하는 것이 중요하다.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5; // 배열의 크기
    // 동적 메모리 할당
    arr = (int*) malloc(n * sizeof(int));
    // 메모리 할당 확인
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // 배열에 값 할당
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }

    // 배열 값 출력
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // 메모리 해제
    free(arr);

    return 0;
}
```

▶ calloc

```c
void* calloc(size_t num, size_t size);
```

'calloc'은 'size_t' 타입의 두 개의 인자를 받는다. 첫 번째 인자는 할당할 요소의 수를 나타내고, 두 번째 인자는 각 요소의 크기를 바이트 단위로 나타낸다. 따라서 'num * size'(size만큼의 메모리를 num개 할당)만큼의 메모리를 할당하고, 0으로 초기화한다.

'calloc'은 할당된 메모리 블록의 첫 번째 바이트를 가리키는 포인터를 반환한다. 이 반환된 포인터는 'void*' 타입이므로, 필요에 따라 적절한 데이터 타입으로 형변환할 수 있다.

만약 메모리 할당에 실패하면, 'calloc'은 NULL을 반환한다. 따라서 'calloc'을 호출한 후에는 항상 반환값을 확인하여 메모리 할당이 성공적으로 이루어졌는지 확인하는 것이 좋다.

'calloc'으로 할당받은 메모리는 프로그램이 명시적으로 'free' 함수를 호출하여 해제하기 전까지 계속 유지된다. 'calloc'으로 할당받은 메모리를 사용한 후에는 반드시 'free'를 호출하여 메모리를 해제해야 한다. 그렇지 않으면 메모리 누수가 발생할 수 있다.

'malloc'과 비교했을 때, 'calloc'의 주요 차이점은 할당된 메모리를 자동으로 0으로 초기화한다는 것이다. 이는 배열이나 구조체와 같은 복잡한 데이터 타입을 초기

화할 때 편리하다.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5; // 배열의 크기

    // 동적 메모리 할당
    arr = (int*) calloc(n, sizeof(int));
    // 메모리 할당 확인
    if (arr == NULL) {
        printf("Memory allocation failed");
        return 1;
    }
```

```
// 배열 값 출력 (모두 0으로 초기화되어 있음)
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // 메모리 해제
    free(arr);

    return 0;
}
```

▶ realloc

```
void* realloc(void* ptr, size_t new_size);
```

'realloc'은 두 개의 인자를 받는다. 첫 번째 인자는 이미 할당된 메모리 블록을 가리키는 포인터이고, 두 번째 인자는 새로운 메모리 블록의 크기를 바이트 단위로 나타낸다.

'realloc'은 새로운 크기의 메모리 블록을 할당하고, 기존 메모리 블록의 내용을 새 메모리 블록으로 복사한 후, 기존 메모리 블록을 해제한다. 그리고 새로 할당된 메모리 블록의 첫 번째 바이트를 가리키는 포인터를 반환한다.

만약 메모리 재할당에 실패하면, 'realloc'은 NULL을 반환한다. 메모리 재할당에 실패하더라도 기존 메모리 블록은 그대로 유지된다. 따라서 'realloc'을 호출한 후에는 항상 반환값을 확인하여 메모리 재할당이 성공적으로 이루어졌는지 확인하는 것이 좋다.

'realloc'으로 크기를 변경한 메모리는 프로그램이 명시적으로 'free' 함수를 호출하여 해제하기 전까지 계속 유지된다. 'realloc'으로 크기를 변경한 메모리를 사용한 후에는 반드시 'free'를 호출하여 메모리를 해제해야 합니다. 그렇지 않으면 메모리 누수가 발생할 수 있다.

'realloc'으로 크기를 늘린 경우, 새롭게 추가된 메모리 영역의 초기값은 정의되지 않는다. 따라서 초기화되지 않은 메모리를 읽으려고 하면 예상치 못한 결과가 발생

할 수 있다. 이러한 이유로, 'realloc'으로 크기를 늘린 후에는 새롭게 추가된 메모리 영역을 적절히 초기화하는 것이 중요하다.

2) 메모리 정렬 및 검색

▶ 메모리 정렬

: 메모리 정렬(Data Alignment)은 프로세서가 메모리에서 데이터를 효율적으로 읽고 쓰기 위한 필수적인 요소이다. 메모리 정렬이란 데이터를 메모리의 특정 위치에 배치하는 것을 말하는데, 이 위치는 데이터의 타입에 따라 달라진다.

▶ 메모리 정렬의 필요성

· 성능: 메모리 정렬은 데이터의 읽기 및 쓰기 성능에 큰 영향을 미친다. 대부분의 프로세서는 정렬된 데이터를 더 빠르게 읽고 쓸 수 있다.

· 안정성: 일부 프로세서의 경우, 정렬되지 않은 데이터에 접근하려고 하면 하드웨어 오류가 발생할 수 있다. 따라서 메모리 정렬은 프로그램의 안정성을 보장한다.

▶ 메모리 정렬의 제어

: 대부분의 프로그래밍 언어는 컴파일러가 자동으로 메모리 정렬을 처리하도록 설계되었다. 그러나 일부 언어에서는 개발자가 메모리 정렬을 직접 제어할 수 있다. 예를 들어, C++에서는 alignas 키워드를 사용하여 메모리 정렬을 지정할 수 있다.

▶ 메모리 정렬의 단점

: 메모리 정렬의 가장 큰 단점은 메모리 낭비이다. 패딩으로 인해 메모리의 일부분이 사용되지 않을 수 있다.

▶ 자연 정렬(Natural Alignment)

: 자연 정렬(Natural Alignment)은 컴퓨터 메모리에서 데이터를 배치하는 방법 중 하나로, 이 방식은 데이터의 타입에 따라 메모리 주소를 배정한다.

이는 프로세서가 데이터를 가장 효율적으로 읽고 쓸 수 있도록 돕는데, 이는 대부분의 컴퓨터 아키텍처에서 자연 정렬된 데이터에 접근하는 것이 더 빠르고, 일부 시스템에서는 자연 정렬되지 않은 데이터에 접근하려고 시도하면 오류가 발생하기 때문이다.

자연 정렬의 예를 들면, 1바이트 크기의 데이터는 어떤 주소에도 배치될 수 있지만, 2바이트 크기의 데이터는 짝수 주소에, 4바이트 크기의 데이터는 4의 배수인 주소에 배치되어야 한다.

예를 들어, 4바이트 정수는 4의 배수인 메모리 주소에 배치되어야 한다. 즉, 메모리 주소 0, 4, 8, 12 등에 배치될 수 있다. 만약 이 정수가 1, 2, 3 등의 주소에 배치되면, 프로세서가 이 정수를 읽기 위해 추가적인 메모리 액세스를 수행해야 하

므로 성능이 저하될 수 있다.
따라서, 자연 정렬은 메모리 접근 성능을 최적화하고, 하드웨어 오류를 방지하기 위한 중요한 기법이다. 개발자는 자신이 작업하는 시스템의 메모리 정렬 요구 사항을 이해하고, 이에 따라 데이터를 배치해야 한다.

▶ 패딩(Padding)

: 패딩(Padding)은 컴퓨터 과학에서 데이터를 일정한 형식이나 길이로 만들기 위해 추가하는 바이트를 말한다.
메모리 패딩은 데이터가 메모리에서 자연스럽게 정렬되도록 하기 위한 방법이다. 데이터의 자연 정렬은 프로세서가 데이터를 더 빠르게 읽고 쓸 수 있도록 해주기 때문에 중요하다. 하지만, 이를 위해선 데이터 사이에 '패딩'이라는 추가적인 공간이 필요할 수 있다.
예를 들어, 구조체에서는 다양한 타입의 데이터가 함께 저장되므로, 이들 데이터가 모두 자연 정렬될 수 있도록 패딩 바이트를 추가할 수 있다.

```
struct example {
    char a;
    int b;
};
```

위 C 언어의 구조체 예시에서 'a'는 1바이트, 'b'는 4바이트이다. int 타입의 'b'는 4바이트 단위로 정렬되어야 하므로, 'a' 다음에 3바이트의 패딩이 추가된다. 따라서 이 구조체의 크기는 8바이트가 된다.
패딩은 성능 최적화와 함께 메모리 낭비라는 단점을 가지고 있다. 불필요한 공간이 추가되기 때문에 메모리 사용량이 증가할 수 있다. 따라서, 패딩을 최소화하는 것이 중요하며, 이를 위해 데이터를 크기 순서대로 배치하거나, 패킹(Packing) 지시어를 사용하여 패딩을 제거할 수 있다.

▶ 바이트 정렬(Byte Alignment)

: 컴퓨터 시스템의 메모리는 비트 단위로 연산이 가능하지만 메모리 관리는 바이트 단위로 한다. 따라서 운영체제는 메모리를 1바이트 단위로 번호를 매겨 관리하고 그 번호를 '메모리 주소'로 지정한다. 바이트 정렬(Byte Alignment)은 데이터를 메모리에 배치하는 방식을 말하며, 이는 컴퓨터 아키텍처에 따라 빅 엔디언(Big Endian) 또는 리틀 엔디언(Little Endian) 방식으로 처리된다.
(1) 빅 엔디언(Big Endian): 빅 엔디언 방식에서는 가장 큰 바이트, 즉 가장 중요한 바이트(Most Significant Byte, MSB)가 가장 낮은 주소에 저장된다. 예를 들어, 4바이트 정수 0x12345678은 메모리에 12 34 56 78 순서로 저장된다.
(2) 리틀 엔디언(Little Endian): 리틀 엔디언 방식에서는 가장 작은 바이트, 즉

가장 덜 중요한 바이트(Least Significant Byte, LSB)가 가장 낮은 주소에 저장된다. 예를 들어, 4바이트 정수 0x12345678은 메모리에 78 56 34 12 순서로 저장된다.

▶ 메모리 검색

: 메모리 검색은 데이터를 찾는 과정을 의미하며, 이는 프로그래밍에서 매우 중요한 부분이다. 여기에는 여러 가지 검색 알고리즘이 있다. 대표적인 검색 알고리즘으로는 순차 검색(Sequential Search), 이진 검색(Binary Search), 해시 검색(Hashing) 등이 있다.

(1) 순차 검색(Sequential Search): 가장 기본적인 검색 방법으로, 배열이나 리스트의 처음부터 끝까지 순서대로 검색하는 방법이다. 구현이 간단하지만, 데이터의 양이 많아질수록 검색 시간이 길어진다.

(2) 해시 검색(Hashing): 해시 함수를 사용해 데이터를 검색하는 방법이다. 해시 함수는 임의의 길이의 데이터를 고정된 길이의 데이터로 해시 테이블에 매핑하는 함수이다. 이를 이용하면 원하는 데이터를 직접 참조하여 검색 시간을 크게 줄일 수 있다.

(3) 이진 검색(Binary Search): 정렬된 데이터에 대해 중간값을 선택하여 찾고자 하는 값과 비교하는 방법이다. 찾고자 하는 값이 중간값보다 크면 중간값보다 큰 부분에 대해, 작으면 중간값보다 작은 부분에 대해 같은 방법으로 검색을 진행한다. 이 방법은 데이터가 미리 정렬되어 있어야 사용할 수 있지만, 검색 속도가 매우 빠르다.

3) 메모리 Lock

▶ 메모리 락(Memory Locking)

: 메모리 잠금(또는 메모리 락, Memory Locking)은 특정 메모리 영역이나 페이지를 물리적 메모리에 고정시키는 것을 의미한다. 일반적으로, 운영 체제는 필요에 따라 메모리를 디스크로 스왑아웃(swap out)하여 물리적 메모리를 관리하는데, 메모리를 잠그면 해당 메모리 영역이 디스크로 스왑아웃되지 않게 된다.

메모리 잠금의 주요 이점은 디스크 스왑으로 인한 지연을 방지하여 성능을 향상시키는 것이다. 특히 Real-time 시스템에서는 스왑의 지연이 큰 문제가 될 수 있으므로, 필요한 메모리 영역을 잠그는 방식을 사용한다.

C 언어에서는 'mlock'과 'munlock', 'mlockall'과 'munlockall' 같은 함수를 사용하여 메모리를 잠글 수 있다.

번호	함수	기능
1	mlock	특정 메모리 영역을 잠그는데 사용된다. 'mlock' 함수는 잠글 메모리의 시작 주소와 크기를 인자로 받는다.
2	munlock	'mlock'으로 잠긴 메모리 영역의 잠금을 해제하는데 사용된다.
3	mlockall	모든 현재 매핑된 페이지와 이후에 매핑될 페이지를 잠그는데 사용된다.
4	munlockall	'mlockall'로 잠긴 모든 페이지의 잠금을 해제하는데 사용된다.

이러한 메모리 잠금 기능은 권한이 필요하며, 일반적으로 root 사용자나 적절한 권한을 가진 사용자만 사용할 수 있다. 또한, 잠길 수 있는 메모리의 양은 시스템에 따라 제한될 수 있다.

메모리 잠금은 필요한 경우에만 사용해야 한다. 메모리를 너무 많이 잠그면 다른 프로세스나 시스템의 작동에 문제가 생길 수 있다. 따라서, 필요한 최소한의 메모리만 잠그고, 사용이 끝나면 반드시 잠금을 해제해야 한다.

3. IPC(Inter-Process Communication)

1) 메모리맵(mmap)

▶ 'mmap'은 메모리 맵을 생성하는 데 사용되는 유닉스 시스템 콜이다. 이를 사용하면 파일의 일부 영역에 메모리 주소를 부여할 수 있다. 이렇게 메모리 주소를 부여하면 마치 변수를 사용하는 것처럼 파일의 데이터를 읽거나 쓸 수 있다. 또한 메모리 주소를 나타내는 포인터와 배열을 사용하여 파일의 데이터를 다룰 수 있다. 또한 mmap은 프로세스간 통신(IPC)에 사용될 수 있다. 두 개 이상의 프로세스가 같은 메모리 맵을 공유하도록 설정하면, 이들은 해당 메모리 영역을 통해 데이터를 공유할 수 있다. 할당된 메모리는 munmap을 이용해 해제할 수 있다.

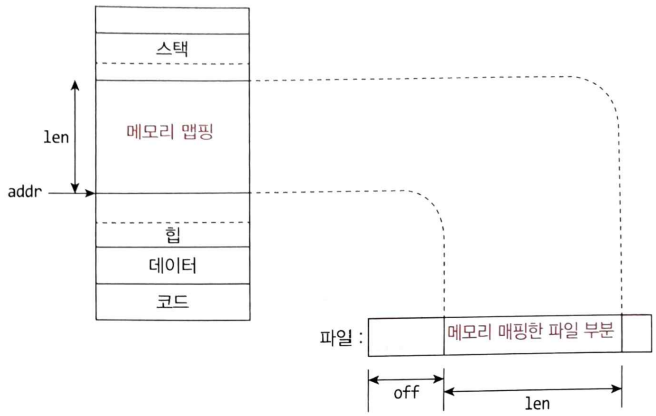


그림 10.15 메모리 맵핑

< mmap을 사용한 메모리 맵핑 >

: mmap() 시스템 호출은 열린 파일의 일부 영역(off뒷 부분부터 len크기)에 메모리 주소를 부여하고 메모리 맵핑된 영역의 시작 주소(addr)를 반환한다.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

· addr: 메모리 맵핑에 부여할 메모리 시작 주소로 이 값이 NULL이면 시스템이 적당한 시작 주소를 선택한다.
· length: 맵핑할 파일 영역의 크기로 메모리 맵핑의 크기와 같다.
· prot: 맵핑된 메모리 영역에 대한 보호 정책을 나타낸다.
 - PROT_READ(읽기)
 - PROT_WRITE(쓰기)
 - PROT_EXEC(실행)
 - PROT_NONE(접근 불가)
· flag: 메모리 맵핑의 특성을 나타낸다.
 - MAP_FIXED: 반환 주소가 addr와 같아야 한다.
 - MAP_FIXED가 지정되지 않고 addr가 0이 아니면, 커널은 addr를 단지 힌트로 사용한다.
 - MAP_SHARED나 MAP_PRIVATE 중의 하나가 반드시 지정되어야 한다.
 - MAP_SHARED: 부여된 주소에 쓰면 파일에 저장된다.
 - MAP_PRIVATE: 부여된 주소에 쓰면 파일의 복사본이 만들어지고 이후부터는 복사본을 읽고 쓰게 된다.
· fd: 대상 파일의 파일 디스크립터
· offset: 맵핑할 파일 영역의 시작 위치

```c
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd;
    char *p;

    // 파일을 엽니다.
    fd = open("test.txt", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // mmap을 사용하여 파일을 메모리에 매핑합니다.
    p = mmap(0, 4096, PROT_READ, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    // 파일의 내용을 출력합니다.
    printf("%s", p);

    // 매핑을 해제합니다.
    munmap(p, 4096);

    close(fd);

    return 0;
}
```

▶ 메모리 관련 함수

(1) memset(): 특정 메모리 영역을 주어진 값으로 설정하는 데 사용된다. 첫 번째 인수는 설정할 메모리의 시작 주소이며, 두 번째 인수는 설정할 값이고, 세 번째 인수는 설정할 바이트의 수이다. 설정이 완료된 메모리의 포인터를 반환한다.

```
char str[50];
memset(str, 'a', sizeof(str));
```

(2) memcmp(): 두 메모리 블록을 비교하는 데 사용된다. 첫 번째와 두 번째 인수로 전달된 메모리 블록을 세 번째 인수로 전달된 바이트 수만큼 비교한다. 두 메모리 블록이 동일하면 0을 반환하고, 첫 번째 메모리 블록이 두 번째 보다 작으면 음수를, 크면 양수를 반환한다.

```
char str1[] = "hello";
char str2[] = "world";
int result = memcmp(str1, str2, sizeof(str1));
```

(3) memchr(): 메모리에서 특정 문자를 찾는 데 사용된다. 첫 번째 인수로 전달된 메모리에서 두 번째 인수로 전달된 문자를 세 번째 인수로 전달된 바이트 수만큼 찾는다. 찾은 문자의 위치를 반환하며, 찾지 못하면 NULL을 반환한다.

```
char str[] = "hello";
char *p = memchr(str, 'e', sizeof(str));
```

(4) memmove(): 메모리 블록을 다른 위치로 이동하는 데 사용된다. 첫 번째 인수로 전달된 대상 메모리에 두 번째 인수로 전달된 원본 메모리의 내용을 세 번째 인수로 전달된 바이트 수만큼 복사한다. 원본과 대상 메모리가 겹치는 경우에도 안전하게 복사를 수행하며, 복사가 완료된 대상 메모리의 포인터를 반환한다.

```
char str[] = "hello";
memmove(str + 1, str, strlen(str));
```

(5) memcpy(): 메모리 블록을 복사하는 데 사용된다. 첫 번째 인수로 전달된 대상 메모리에 두 번째 인수로 전달된 원본 메모리의 내용을 세 번째 인수로 전달된 바이트 수만큼 복사한다. 원본과 대상 메모리가 겹치지 않는 경우에 사용하며, 복사가 완료된 대상 메모리의 포인터를 반환한다.

```
char str1[5];
char str2[] = "hello";
memcpy(str1, str2, sizeof(str1));
```

2) 공유 메모리

▶ 프로세스는 지역 변수, 동적 변수, 전역 변수와 같은 데이터를 저장하기 위한 자신만의 메모리 영역을 가지고 있는데 이 메모리 영역은 다른 프로세스가 접근해서 함부로 데이터를 읽거나 쓰지 못하도록 커널에 의해서 보호된다. 만약 다른 프로세스의 메모리 영역을 침범하려고 하면 커널은 침범 프로세스에 SIGSEGV(할당된 메모리의 범위를 벗어나는 곳을 접근할 때 발생하는 세그멘테에션 위반 시그널)을 보내게 된다.

▶ 다수의 프로세스가 동시에 작동하는 리눅스의 특성상 프로세스의 메모리 영역은 반드시 보호되어야 하나 프로세스 사이에 메모리 영역에 있는 데이터를 공유하기 위해서는 다른 프로세스도 사용할 수 있도록 할 필요가 있다. 공유 메모리는 프로세스 사이에 메모리 영역을 공유해서 사용할 수 있도록 해준다.

▶ 공유 메모리의 사용 과정

(1) 공유 메모리 생성: 특정 크기의 메모리 블록을 생성하고, 이를 식별할 수 있는 고유한 키를 할당한다.

(2) 공유 메모리 연결: 생성된 공유 메모리를 프로세스의 주소 공간에 연결(attach)한다. 이렇게 연결된 메모리 블록은 일반적인 메모리와 마찬가지로 접근할 수 있다.

(3) 공유 메모리 사용: 연결된 메모리를 읽고 쓰는 작업을 수행한다.

(4) 공유 메모리 해제: 더 이상 필요하지 않은 공유 메모리를 프로세스의 주소 공간에서 해제(detach)한다.

(5) 공유 메모리 제거: 모든 프로세스가 공유 메모리의 사용을 완료했으면, 시스템에서 공유 메모리를 제거한다.

▶ 공유 메모리 생성 shmget()

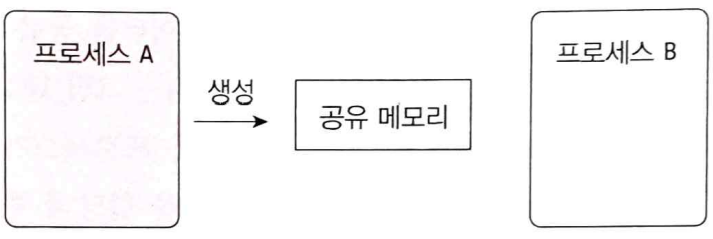


그림 10.12 공유 메모리 생성

< 공유 메모리 생성 >

: shmget() 시스템 호출의 기본적인 역할은 키(key)가 가리키는 새로운 공유 메

모리를 생성하고 생성된 공유 메모리를 식별하는 식별자 ID를 반환하는 것이다.

```
int shmget(key_t key, size_t size, int shmflg);
```

· key: IPC_PRIVATE(항상 새로운 공유 메모리를 생성하고자 할 때 사용) 또는 키 생성 함수인 ftok(char *path, int id)함수로 생성한 키를 사용한다.
· size: 공유할 메모리의 크기를 나타낸다. 기존 공유 메모리를 사용하는 경우에는 이 값은 무시된다.
· shmflg
- IPC_CREAT: 새로운 공유 메모리를 생성한다. 생성할 공유 메모리의 접근권한을 OR(|)하여 함께 지정한다.
- IPC_EXCL: IPC_CREAT와 함께 사용되면 해당 공유 메모리가 이미 존재하면 실패한다.
- 이 플래그가 0이면 지정한 key와 연관된 기존의 공유 메모리를 찾아서 준다.

▶ 공유 메모리 연결 shmat()

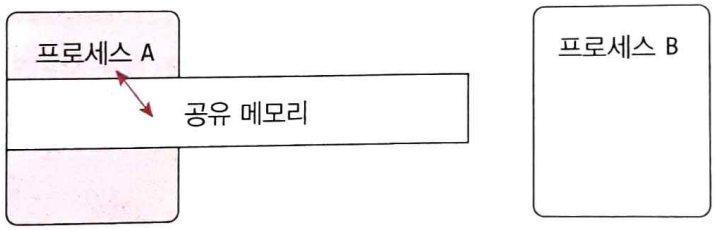


그림 10.13 공유 메모리에 연결

< 공유 메모리 연결 >

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

: shmget()을 통해 생성된 공유 메모리(shmid)를 사용하기 위해서는 프로세스A(이 시스템 호출을 실행하는 프로세스)의 특정 메모리 위치(shmaddr)에 연결해야 한다. shmaddr가 NULL일 경우에는 커널에서 적절한 주소를 선정하여 반환한다. shmflg는 공유 메모리에 대한 읽기/쓰기 권한을 지정한다.
이 시스템 호출이 정상적으로 동작한다면 적절한 포인터를 반환하고 실패하면 (void*) -1을 반환한다. 여기서 void*를 반환하는 이유는 공유 메모리에 있는 데이터가 어떤 자료형 인지 모르게 때문에 무엇이든 받을 수 있는 void* 자료형으로 반환한다.

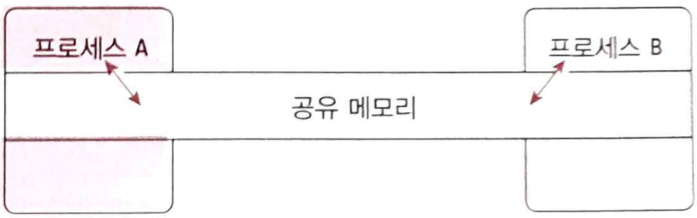


그림 10.14 다른 프로세스도 공유 메모리에 연결

< 공유 메모리 연결 >

: 공유 메모리를 사용하여 다른 프로세스와 데이터를 공유하기 위해서는 위 그림과 같이 다른 프로세스도 이 공유 메모리에 연결해야 한다.

▶ 공유 메모리 연결 해제 shmdt()

```
int shmdt(const void *shmaddr);
```

: 공유 메모리 사용을 마치면 연결을 해제해야 한다. 이 시스템 호출은 shmat()에서 받은 공유 메모리에 대한 연결 포인터를 전달받아 공유 메모리에 대한 연결을 해제한다. 이는 공유 메모리 자체를 제거하는 것은 아님에 주의해야 한다. shmdt()는 성공 시 0, 실패 시 –1을 반환한다.

▶ 공유 메모리 제어 shmctl()

```
int shmctl(int shmid, int 층, struct shmid_ds *buf);
```

: cmd는 shmid가 나타내는 공유 메모리를 제어하기 위한 명령어로 다음과 같은 명령어를 사용할 수 있다.

- IPC_RMID: 공유 메모리를 제거한다.
- IPC_SET: 공유 메모리의 정보(사용자 ID, 그룹 ID, 접근권한 등)를 buf에서 지정한 값으로 바꾼다.
- IPC_STAT: 현재 공유 메모리의 정보를 buf에 저장한다.
- SHM_LOCK: 공유 메모리를 잠금한다.
- SHM_UNLOCK: 공유 메모리 잠금을 해제한다.

buf는 공유 메모리에 관한 정보를 저장할 수 있는 shmid_ds 구조체에 대한 포인터이다.

3) 세마포어

▶ 세마포어(Semaphore)

: 세마포어는 다중 프로세싱 환경에서 동기화 문제를 해결하는 데 사용되는 중요한

도구이다. 이는 단일 혹은 다수의 공유 자원에 여러 프로세스 또는 스레드에 대한 동시 접근을 제어하고, 임계 영역에 대한 상호 배제를 보장하는 역할을 한다.

▶ 세마포어 변수(S) : 공유된 자원에 접근하려고 하는 두 개 이상의 프로세스에 대해 부여되는 변수이다. 0/1 혹은 0/양수 값을 가지며 P연산, V연산에 의해서만 접근 가능하다. 여기서 1 또는 양수는 공유된 자원의 수를 의미한다. S의 값이 0이면 공유 자원에 접근 할 수 없다는 것을 의미한다.

▶ 세마포어는 정수 값과 함께 작동하는 변수로 볼 수 있으며, 기본적으로 두 가지 주요 연산, 즉 wait (또는 P 연산)과 signal (또는 V 연산)을 지원한다.

(1) Wait (P) 연산: 세마포어의 값이 0보다 크면 값을 감소시키고, 그렇지 않으면 (세마포어의 값이 0이면) 프로세스를 대기 상태로 만든다. 이것은 일반적으로 임계 영역에 진입하기 전에 수행된다.

→ 사용할 수 있는 공유 자원 있으면 세마포어 값을 –1 해준 뒤 사용하고, 사용할 수 있는 공유 자원이 없으면 대기 하도록 하는 함수이다.

(2) Signal (V) 연산: 세마포어의 값을 증가시키고, 만약 어떤 프로세스가 대기 중이라면 하나를 깨운다. 이것은 일반적으로 임계 영역에서 나온 후에 수행된다.

→ 공유 자원을 다 사용한 후에 세마포어 값을 +1 해주고, 대기하고 있는 프로세스나 스레드에게 공유 자원을 사용할 수 있다고 알려주는 함수이다.

4) 메시지큐

▶ 메시지 큐는 프로세스 간에 데이터를 주고받는 방법 중 하나로, 메시지를 큐에 저장하고 필요한 프로세스가 메시지를 읽어가는 방식으로 동작한다. 이는 프로세스 간에 정보를 교환하거나 동기화하는 데 사용된다.

▶ 메시지 큐를 사용하면 프로세스는 메시지를 생성하여 큐에 삽입하고, 다른 프로세스는 큐에서 메시지를 받아 처리한다. 이렇게 하면 프로세스는 서로 독립적으로 실행되면서도 정보를 주고받을 수 있다.

▶ 메시지 큐의 생성 과정

(1) 메시지 큐 생성: 메시지 큐를 생성하고, 이를 식별할 수 있는 고유한 키를 할당한다.

(2) 메시지 전송: 메시지를 생성하여 큐에 삽입한다. 메시지는 일반적으로 특정 형식을 가지며, 데이터 외에도 메시지의 종류를 나타내는 타입 정보를 포함한다.

(3) 메시지 수신: 큐에서 메시지를 받아 처리한다. 메시지를 받을 때는 보통 메시지의 타입에 따라 필요한 메시지를 선택하여 받을 수 있다.

(4) 메시지 큐 제거: 더 이상 필요하지 않은 메시지 큐를 시스템에서 제거한다.

▶ 메시지 큐 함수

(1) msgget(): 새로운 메시지 큐를 생성하거나 이미 존재하는 메시지 큐에 접근하는 데 사용된다. 첫 번째 인수로는 메시지 큐를 식별하는 키를, 두 번째 인수로는 메시지 큐의 권한과 생성 옵션을 전달한다. 성공 시 메시지 큐의 식별자를 반환하고, 실패 시 -1을 반환한다.

```
int msqid = msgget(key, 0666 | IPC_CREAT);
```

(2) msgsnd(): 메시지를 메시지 큐에 보내는 데 사용된다. 첫 번째 인수로는 메시지 큐의 식별자를, 두 번째 인수로는 보낼 메시지를, 세 번째 인수로는 메시지의 크기를, 마지막 인수로는 메시지 전송 옵션을 전달한다. 성공 시 0을 반환하고, 실패 시 -1을 반환한다.

```
struct msgbuf {
    long mtype;       /* message type */
    char mtext[100];  /* message text */
};

struct msgbuf buf;
buf.mtype = 1;
strcpy(buf.mtext, "hello");
msgsnd(msqid, &buf, sizeof(buf.mtext), 0);
```

(3) msgrcv(): 메시지 큐에서 메시지를 받는 데 사용된다. 첫 번째 인수로는 메시지 큐의 식별자를, 두 번째 인수로는 받을 메시지를, 세 번째 인수로는 메시지의 크기를, 네 번째 인수로는 받을 메시지의 타입을, 마지막 인수로는 메시지 수신 옵션을 전달한다. 성공 시 받은 메시지의 크기를 반환하고, 실패 시 -1을 반환한다.

```
struct msgbuf buf;
msgrcv(msqid, &buf, sizeof(buf.mtext), 1, 0);
```

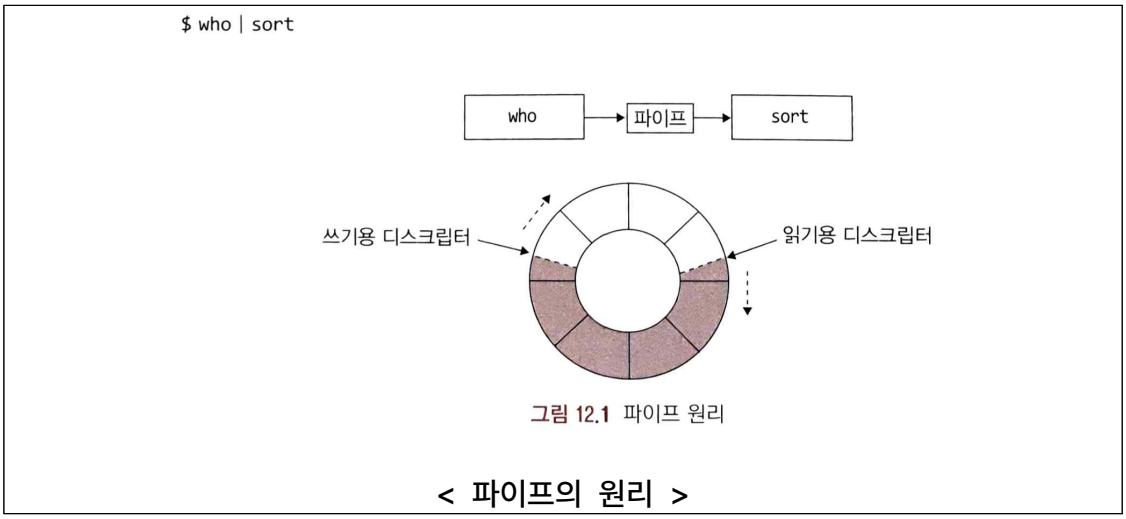
(4) msgctl(): 메시지 큐를 제어하는 데 사용된다. 첫 번째 인수로는 메시지 큐의 식별자를, 두 번째 인수로는 수행할 제어 명령을, 세 번째 인수로는 메시지 큐의 속성을 담은 구조체를 전달한다. 성공 시 0을 반환하고, 실패 시 -1을 반환한다.

```
msgctl(msqid, IPC_RMID, NULL);
```

4. I/O 인터페이스 및 멀티플렉싱

1) PIPE와 FIFO

▶ 파이프(PIPE)의 개념



< 파이프의 원리 >

: 파이프는 유닉스와 유닉스 계열 시스템에서 프로세스 간 통신을 위해 사용되는 메커니즘이다. 일반적으로 한 프로세스에서 다른 프로세스로 데이터를 전달하는데 사용된다. 예를 들어 $ who | sort 라고 명령어를 입력하면 who 명령어의 표준 출력은 파이프를 통해 모두 sort 명령어의 표준입력이 된다.

파이프는 수도 파이프 한쪽 끝에서 물을 보내고 다른 쪽 끝에서 물을 받는 것처럼 파이프는 일반 파일과 달리 두 개의 파일 디스크립터를 갖는다. 하나는 쓰기용이고 다른 하나는 읽기용이다. 위 그림과 같이 한 프로세스는 쓰기용 파일 디스크립터를 이용하여 파이프에 데이터를 보내고(쓰고) 다른 프로세스는 읽기용 파일 디스크립터를 이용하여 그 파이프에서 데이터를 받는다(읽는다). 파이프를 사용할 때는 보내는 프로세스와 받는 프로세스가 정해져 있어 데이터를 한 방향으로만 보낼 수 있다.

파이프의 또 하나의 특징은 수도 파이프를 통해 보낼 수 있는 물의 양에 제한이 없는 것처럼 파이프를 통해 보낼 수 있는 데이터의 양에 제한이 없다는 점이다. 단지 한 프로세스가 파이프에 데이터를 쓰는 속도가 너무 빠르면 순간적으로 파이프가 가득 찰 수는 있지만 시간이 지나면서 다른 프로세스가 파이프로부터 데이터를 읽어 가면 다시 파이프에 데이터를 보낼 수 있는 상태가 된다. 이를 위해 자동으로 파이프에 쓰기가 잠시 중지(blocking)된다.

▶ 파이프 만들기 pipe()

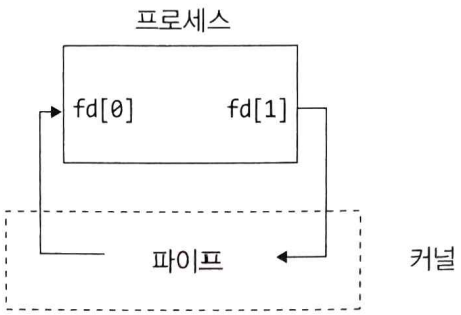


그림 12.2 파이프 생성

< 파이프 생성 >

```
int pipe(int fd[2]);
```

: pipe() 시스템 호출은 그림과 같이 하나의 파이프를 만들고 그 파이프에 대한 두 개의 파일 디스크립터 fd[0]과 fd[1]을 제공한다. fd[0]은 읽기용이고 fd[1]은 쓰기용이다.

파이프를 사용하려면 파일사용과 같이 read(), write() 시스템 호출을 사용하면 되는데 파이프에 데이터를 보내려면 write(fd[1],...) 시스템 호출을 사용하여 파이프에 쓰면 된다. 파이프에서 데이터를 받으려면 read(fd[0],...) 시스템 호출을 사용하여 파이프로부터 읽으면 된다.

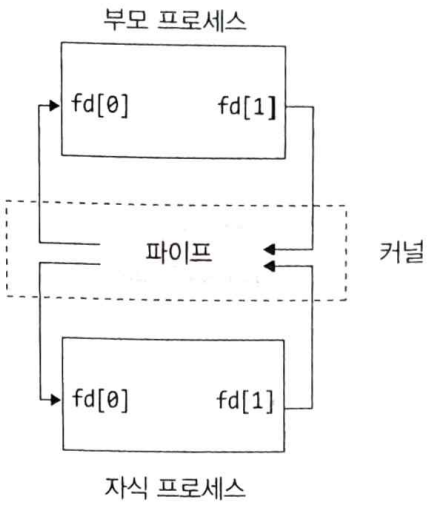


그림 12.3 자식 생성 후의 파이프

< 자식 생성 후 파이프 >

: 파이프는 파일과 달리 이름이 없으므로 서로 관련된 (주로 부모-자식) 프로세스 사이의 데이터를 전송하는 데에만 사용될 수 있다. 부모 프로세스가 파이프를 만든

후 자식 프로세스를 생성하면 자식 프로세스는 부모 프로세스를 복제하여 만들어지므로 부모 프로세스가 만든 파이프도 당연히 공유하여 사용할 수 있다. 위 그림은 자식 프로세스 생성 후에 부모 프로세스와 자식 프로세스 사이의 파이프 공유 상황을 보여준다.

▶ 파이프를 이용한 데이터 전송

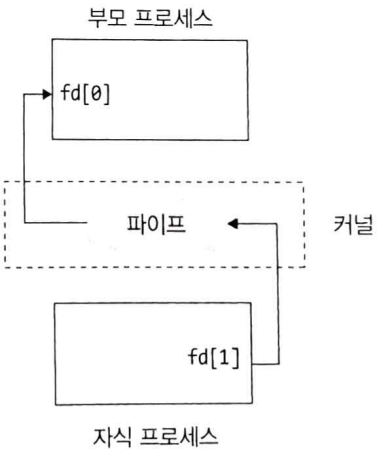


그림 12.4 파이프를 통해 자식에서 부모로 데이터 보내기

< 파이프를 이용한 데이터 전송 >

(1) 한 프로세스가 파이프를 생성한다.

(2) 그 프로세스가 자식 프로세스를 생성한다.

(3) 쓰는 프로세스는 읽기용 파이프 디스크립터를 닫는다.
읽는 프로세스는 쓰기용 파이프 디스크립터를 닫는다.

(4) 각 프로세스는 write()와 read() 시스템 호출을 사용하여 파이프를 통해 데이터를 송수신한다.

(5) 각 프로세스가 살아 있는 파이프 디스크립터를 닫는다.

: 예를 들어 자식 프로세스에서 부모 프로세스로 데이터를 보내는 경우에는 자식 프로세스가 읽기용 파이프 디스크립터 fd[0]을 닫고 부모 프로세스는 쓰기용 파이프 디스크립터 fd[1]을 닫으면 된다.

▶ 자식 프로세스의 표준 출력을 파이프를 통해 부모 프로세스에게 보내는 방법은 자식 프로세스에서 dup2(fd[1],1); 함수를 이용해서 쓰기용 파이프를 표준출력에 복제해주고, close(fd[1]);을 통해 파이프를 닫아주면 된다.

▶ 파이프 라이브러리 함수

(1) 파이프 열기 popen()

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
//성공하면 파이프를 위한 파일 포인터를, 실패하면 NULL을 반환한다.
int pclose(FILE *fp);
//성공하면 command 명령어의 종료 상태를, 실패하면 –1을 반환한다.
```

: 부모와 자식 프로세스 간에 파이프를 통한 입출력을 받기 위해서 pipe()함수로 파이프를 만들고, 자식 프로세스를 만들고, 그 출력을 파이프를 통해 부모에게 전달하는 과정을 거쳤다. popen()함수는 이 모든 과정을 알아서 해주고, 이 파이프를 가리키는 FILE 포인터만을 반환한다.

fopen은 'r(read)' 혹은 'w(write)'의 매개변수를 받아 역할을 정할 수 있다.

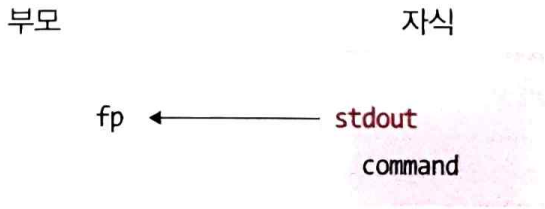


그림 12.8 popen(command, "r")의 역할

< popen(command,"r") >

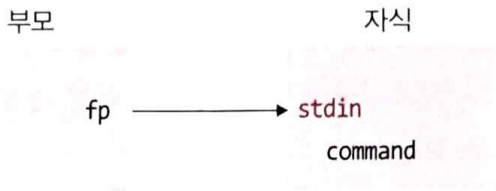


그림 12.9 popen(command, "w")의 역할

< popen(command,"w") >

: fp = popen(command,"r");은 읽기 모드(자식→부모로 전달)로 파이프를 사용할 수 있고, fp = popen(command,"w");은 쓰기 모드(부모→자식으로 전달)로 파이프를 사용할 수 있다.

▶ FIFO

: 파이프는 이름이 없으므로 부모 자식과 같은 서로 관련된 프로세스 사이의 통신에는 사용될 수 있으나 서로 관련 없는 프로세스 사이에는 사용될 수 없다. 이러한 문제점을 해결하기 위한 파이프가 named pipe(이름 있는 파이프)이다. 이름 있는

파이프는 일반 파일처럼 이름이 있으며 파일 시스템 내에 존재하므로 fifo file(fifo 파일)이라고도 한다. fifo는 first-in-first-out의 약어로 먼저 들어간 데이터가 먼저 나오는 파이프의 특성을 반영한 용어이다.

▶ FIFO의 생성

```
$ mknod myPipe p
$ chmod ug+rw myPipe
$ ls –1 myPipe
prw-rw-r— 1 (user) (group)  0 3월 11 13:03 myPipe
```

: 이름 있는 파이프를 만드는 방법은 두 가지가 있다. 첫 번째 방법은 p옵션과 함께 mknod명령어를 사용하여 만드는 것이다. 예를 들어 위의 표와 같이 myPipe라는 이름의 파이프를 만들고 이 파이프를 소유자나 같은 그룹이 읽고 쓸 수 있도록 접근권한을 변경하면 된다.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
path를 갖는 이름 있는 파이프를 생성하고 접근권한은 mode로 설정한다. 성공하면 0을, 실패하면 –1을 반환한다.
```

: 두 번째 방법은 프로그램 내에서 mkfifo() 시스템 호출을 사용하여 만드는 것이다. 이름 있는 파이프를 사용하기 위해서는 다른 파일처럼 먼저 파이프를 열어야 한다. 쓰는 프로세스는 쓰기 전용으로 파이프를 열고 write() 시스템 호출을 사용하여 데이터를 보낸다. 읽는 프로세스는 읽기 전용으로 파이프를 열고 read() 시스템 호출을 사용하여 데이터를 받는다.

▶ 이름 있는 파이프를 사용할 때는 데드락(deadlock)에 주의해야 한다. 예를 들어, 프로세스가 데이터를 쓰기 위해 FIFO 파일을 열려고 하지만 아무도 해당 파일을 읽기 위해 열지 않은 경우, 프로세스는 FIFO 파일이 열릴 때까지 블로킹되거나 대기하게 된다. 이러한 문제를 방지하기 위해, FIFO 파일을 O_NONBLOCK 플래그와 함께 열어 논블로킹 모드로 설정할 수 있다.

2) 소켓

▶ 파이프는 단방향 통신이라는 단점을 가지고 있는데 이 한계를 개선한 것이 소켓을 통한 네트워크 통신(NetworkCommunication)이다.

▶ 소켓(socket)은 네트워크에서 데이터를 주고받는 데 사용되는 엔드포인트이다. 네트워크를 통해 두 대의 컴퓨터나 같은 컴퓨터 내의 두 프로세스 간에 데이터를

주고받는 통신 메커니즘이 필요한 경우, 소켓을 이용한다.
소켓은 IP 주소와 포트 번호의 조합으로 구성되며, 이를 통해 네트워크 상의 특정 위치를 지정할 수 있다.

▶ 클라이언트–서버 모델

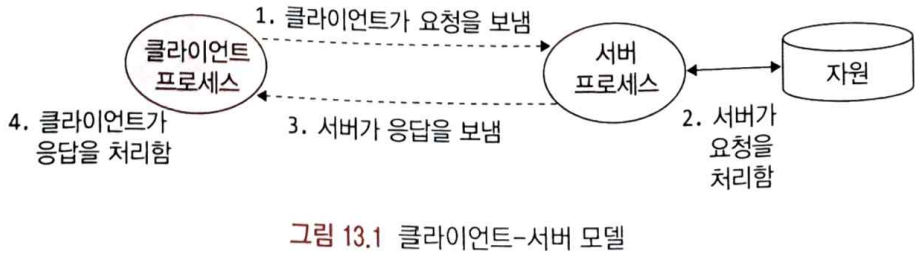


그림 13.1 클라이언트–서버 모델

< 클라이언트–서버 모델 >

: 대부분의 네트워크 응용 프로그램들은 위와 같은 클라이언트-서버 모델을 기반으로 동작한다. 클라이언트-서버 모델은 하나의 서버 프로세스와 여러 개의 클라이언트 프로세스들로 구성된다. 동작 과정은 위와 같다.
소켓은 양방향 통신 방법으로 클라이언트-서버 모델을 기반으로 프로세스 사이의 통신에 매우 적합하다.

▶ 소켓의 종류

(1) 유닉스 소켓(AF_UNIX): 같은 호스트 내의 프로세스 사이의 통신 방법이다.
(2) 인터넷 소켓(AF_INET): 인터넷에 연결된 서로 다른 호스트에 있는 프로세스 사이의 통신 방법이다.

▶ 소켓의 연결 과정

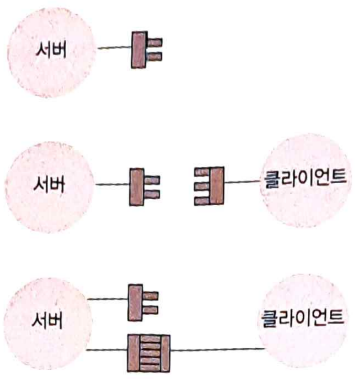


그림 13.2 소켓 연결

< 소켓 연결 과정 >

: 소켓을 이용한 통신을 하기 위해서는 먼저 소켓 연결이 이루어져야 하는데 이를 위한 전체 적인 과정은 위의 그림과 같으며 진행 과정은 다음과 같다.

(1) 서버가 소켓을 만든다.

(2) 클라이언트가 소켓을 만든 후 서버에 연결 요청을 한다.

(3) 서버가 클라이언트의 연결 요청을 수락하면 소켓 연결이 이루어진다.

: 서버는 연결 요청을 수락할 때 원래 소켓의 복사본 소켓을 만들어 이 복사본 소켓을 클라이언트 소켓과 연결하고 원래 소켓은 그대로 유지한다. 이렇게 하는 이유는 서버는 다른 클라이언트로부터 소켓 연결 요청을 계속해서 받을 수 있기 때문이다.

▶ 시스템 호출을 통한 소켓 연결

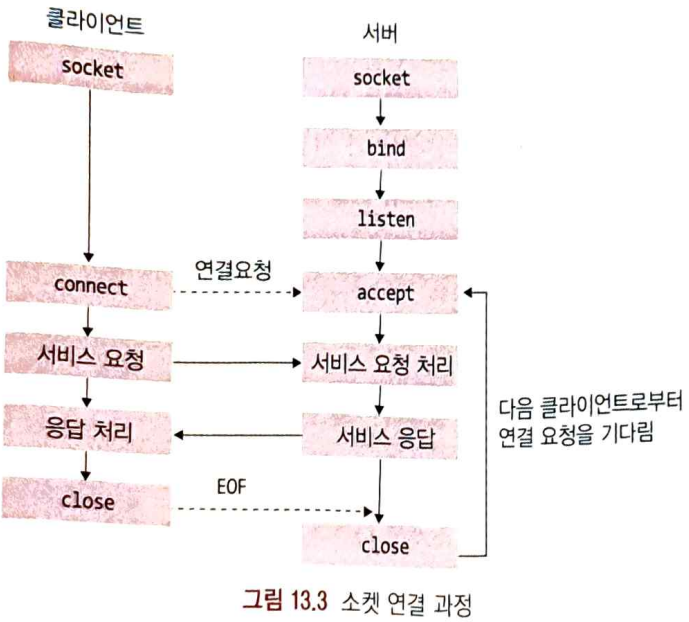


그림 13.3 소켓 연결 과정

< 시스템 호출을 이용한 소켓 연결 과정 >

- 서버

(1) 서버는 socket() 시스템 호출을 이용하여 소켓을 만들고 이름을 붙인다.

(2) 서버는 listen() 시스템 호출을 이용하여 대기 큐를 만든다.

(3) 서버는 클라이언트로부터 연결 요청을 accept() 시스템 호출을 이용하여 수락한다.

(4) 소켓 연결이 이루어지면 서버는 보통 자식 프로세스를 생성하고 자식 프로세스로 하여금 클라이언트로부터 서비스 요청을 받아 처리한 후 클라이언트에게 응답하게 한다.

▶ 소켓 연결을 위한 시스템 호출 종류

(1) 소켓 만들기 soket()

int socket(int domain, int type, int protocol); 소켓을 생성하고 소켓을 위한 파일 디스크립터를 반환한다. 실패하면 –1을 반환한다.

: 소켓을 생성하는 시스템 호출로 UNIX 소켓(AF_UNIX) 혹은 인터넷 소켓(AF_INET)을 생성할 수 있다. domain은 사용할 소켓의 종류에 때라 AF_UNIX 혹은 AF_INET으로, type은 SOCK_STREAM으로, protocol은 DEFAULT_PROTOCOL으로 한다.

(2) 소켓에 이름(주소) 주기 bind()

int bind(int fd, struct sockaddr* address, int addressLen); 소켓에 대한 이름 바인딩이 성공하면 0을 실패하면 –1을 반환한다.

: socket() 시스템 호출에 의해 생성된 소켓은 별도의 이름이 없다. 그러나 보통 소켓은 이름이 필요한데, 이는 클라이언트가 연결 요청을 하기 위해서는 서버 소켓의 이름을 알아야 하기 때문이다. bind() 시스템 호출을 이용하여 소켓의 이름을 정할 수 있다. address는 sokeaddr_un 구조체에 대한 포인터이며 addressLen는 이 구조체의 크기이다. fd가 나타내는 소켓의 이름을 address 내의 이름으로 정한다.

(3) 소켓 큐 생성 listen()

int listen(int fd, int queueLength); 소켓 fd에 대한 연결 요청을 기다린다. 성공하면 0을 실패하면 –1을 반환한다.

: 하나의 서버 소켓에 대해 여러 개의 클라이언트로부터 연결 요청이 거의 동시에 이루어질 수 있다. 이 경우에 이러한 연결 요청들은 수락될 때까지 대기 큐에 들어가게 되는데 listen() 시스템 호출은 이 대기 큐의 길이를 정한다. 이 길이에 따라 대기할 수 있는 연결 요청의 최대 개수가 정해진다.

(4) 소켓에 연결 요청 connect()

int connect(int fd, struct sockaddr* address, int addressLen); fd가 나타내는 클라이언트 소켓과 address가 나타내는 서버 소켓과의 연결을 요청한다. 성공하면 0을 실패하면 –1을 반환한다.

: connect() 시스템 호출은 fd가 나타내는 클라이언트 소켓과 address가 나타내는 서버 소켓과의 연결을 요청한다. 성공하면 fd를 서버 소켓과의 통신에 사용할 수 있다.

(5) 소켓 연결 요청 수락 accept()

int accept(int fd, struct sockaddr* address, int* addressLen); 클라이언트부터의 소켓 fd에 대한 연결 요청을 수락한다. 성공하면 새로 만들어진 복사본 소켓의 파일 디스크립터를, 실패하면 –1을 반환한다.

: 서버는 accept() 시스템 호출을 사용하여 클라이언트로부터의 연결 요청을 수락한다.

(6) 소켓을 통한 데이터 송수신 send()와 recv()

int send(int fd, const void *msg, size_t len, int flags);
크기가 len인 msg 메시지를 fd 소켓으로 보낸다. 보내진 바이트 수를 반환하거나 오류 발생 시 –1을 반환한다.
 	
int recv(int fd, void *buf, size_t len, int flags);
fd 소켓으로부터 len 크기만큼 메시지를 받아 buf에 저장한다. 받은 바이트 수를 반환하거나 오류 발생 시 –1을 반환한다.

: 소켓 연결이 이루어지면 소켓을 통해 데이터를 주고받을 수 있다. 소켓도 연결이 되면 파일과 마찬가지로 파일 디스크립터를 반환하므로 이 파일 디스크립터를 통하여 소켓에 데이터를 보내거나 받을 수 있다. 구체적으로는 파일에서 사용했던 write() 시스템 호출을 사용하여 소켓에 메시지를 보낼 수 있고 read() 시스템 호출을 사용하여 소켓으로부터 메시지를 받을 수 있다. read(), write() 시스템 호출 외에 소켓을 위한 시스템 호출인 send(), recv()을 사용하여 메시지를 보내거나 받는 것도 가능하다.

3) selct, pselect

▶ '다중 입출력(Multiplexed I/O)'은 프로그램(단일 스레드)에서 여러 개의 파일을 작업하고자 할 때 사용할 수 있는 메커니즘이다. '단일 스레드'에서 여러 개의 파일을 작업하고자 할 때 사용할 수 있는 다른 방법으로는 '논블록(Non-Block)입출력'을 사용하는 방법도 있지만, 논블록 입출력은 프로그래밍이 까다롭다. 'select'는 '블록/비동기적 입출력'에서의 '다중 입출력' 모델이다.

▶ select는 다수의 파일을 순서대로 작업하는 것이 아닌, 작업할 준비가 된 파일에 대해서만 작업을 하기 위해 고안된 함수이다. 예를 들어, A, B, C, D라는 각각의 파일이 있고 순서대로 작업한다고 했을 때 read()함수로 B를 읽으려고 했으나 B가 내용이 비어있다고 하면 프로그램은 B에서 읽을 내용이 있을 때까지 '블록(Block)'된다. 이 때 작업이 준비된 파일만 다룬다면 블록이 발생하지 않을 것이다. 또 다른 예로, 서버에 100개의 클라이언트가 연결되어 있을 때, 서버에서는

100개의 스레드를 생성하여 처리할 수는 없다. 따라서 100개의 socket에 대해서 순회하며 지켜보다가 하나의 socket에 읽을 데이터가 생겼다면(event 발생) 그 socket에서 read하고 처리한다.

▶ select()는 Unix와 Unix-like 시스템에서 제공하는 시스템 호출로, 여러 개의 파일 디스크립터를 동시에 모니터링할 수 있게 해준다. 이를 위해 해당 파일 디스크립터가 입출력을 수행할 준비가 되거나 마지막 매개변수인 timout변수에 정해진 시간이 경과할 때까지만 '블록'(Block)'된다. 이는 네트워크 프로그래밍에서 특히 중요한 함수로, 소켓이 읽기, 쓰기, 예외 상황을 위해 준비되었는지 확인하는 데 사용된다.

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- nfds: 모니터링하려는 파일 디스크립터 중 가장 큰 값에 1을 더한 값이다.
- fd_set: 파일 디스크립터의 집합을 나타내는 데이터 타입으로, FD_ZERO, FD_SET, FD_ISSET, FD_CLR 등의 매크로와 함께 사용된다.
- readfds: 읽기 이벤트를 모니터링하려는 파일 디스크립터의 집합이다. select()는 해당 파일 디스크립터들이 '읽기'가 가능한지 감시한다.
- writefds: 쓰기 이벤트를 모니터링하려는 파일 디스크립터의 집합이다. select()는 해당 파일 디스크립터들이 '쓰기'가 가능한지 감시한다.
- exceptfds: 예외 이벤트를 모니터링하려는 파일 디스크립터의 집합이다. select()는 예외가 발생했거나 대역을 넘어서는 데이터(소켓)가 존재하는지 감시한다.
- timeout: select() 함수가 블로킹되는 최대 시간을 지정합니다. NULL이면 무한히 대기한다.

▶ select() 함수는 timeout이 지난 후, 또는 모니터링하고 있는 파일 디스크립터 중 하나가 읽기, 쓰기, 예외 상황을 위해 준비되면 반환한다. 반환 값은 준비된 파일 디스크립터의 수를 나타낸다.

다만, select() 함수는 몇 가지 알려진 한계가 있다. 예를 들어, 하나의 프로세스가 모니터링할 수 있는 파일 디스크립터의 수가 제한되어 있고, 대량의 파일 디스크립터를 관리하는 데 비효율적인 경우가 있다. 이러한 한계를 극복하기 위해 poll(), epoll(), kqueue() 등의 대안적인 메커니즘이 개발되었다.

▶ pselect()는 select()와 유사한 시스템 호출이지만, 몇 가지 추가적인 기능을 제공한다. select()와 마찬가지로 여러 파일 디스크립터를 동시에 모니터링하는 데 사용되지만, 시그널 마스크를 변경하는 기능과 나노초 단위의 타임아웃을 지정할 수 있는 기능이 추가되었다.

```
int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t *sigmask);
```

- timeout: select()에서는 struct timeval을 사용하여 마이크로초 단위의 타임아웃을 지정했지만, pselect()에서는 struct timespec을 사용하여 나노초 단위의 타임아웃을 지정할 수 있어 정밀도가 더 높다.
- sigmask: pselect() 호출 동안에 블록하려는 시그널 집합을 지정한다. 이 인자를 사용하면, select()를 사용하여 I/O 이벤트를 기다리는 동안 발생할 수 있는 경쟁 상태를 방지할 수 있다.

▶ pselect() 함수의 반환 값과 에러 조건은 select()와 동일하다. 준비된 파일 디스크립터의 개수를 반환하며, 에러가 발생하면 -1을 반환하고 errno에 적절한 값을 설정한다.

pselect()는 select()가 가진 문제점을 일부 해결하지만, 여전히 대량의 파일 디스크립터를 효율적으로 처리하는 데 한계가 있다. 이러한 한계를 극복하기 위해 poll(), epoll(), kqueue() 등의 대안적인 메커니즘이 사용되곤 한다.

▶ select()와 pselect()의 차이점

(1) select() 함수의 만료 시간 값은 timeval 구조체로 하지만, pselect() 함수는 timespec 구조체를 사용한다. timespec 구조체는 나노초 단위로 표현하기 때문에 시간 정밀도가 높다.

(2) pselect() 함수의 만료시간 인자는 const 로 선언되어 있기 때문에 pselect() 호출에 의해 값이 변경되지 않는다.

(3) pselect() 함수의 마지막 인자 sigmask 를 통해서 신호 마스크를 지정할 수 있다. sigmask 가 NULL 이면 신호에 관해서 select() 함수와 동일한 기능을 하게된다. NULL이 아니면 sigmask 가 가리키는 신호 마스크가 자동으로 설정되어 차단되고 pselect() 호출이 반환 될 때 신호 마스크가 복원되어 실행하게 된다.

4) Non-Blocking I/O

▶ 동기(Synchronous)와 비동기(Asynchronous)

(1) 동기(Synchronous): 동기(Synchronous) 방식은 어떤 작업이 순차적으로 실행된다. 즉, 한 작업이 끝나야 다음 작업이 시작될 수 있다. 예를 들어, 함수 호출이 동기 방식이라면, 해당 함수가 반환될 때까지 프로그램의 실행이 멈추고 기다린다. 동기 방식은 코드의 흐름을 쉽게 이해하고 예측할 수 있다는 장점이 있지만, 이전 작업이 끝나기를 기다리는 동안 시간이 소요될 수 있어 효율성이 떨어질 수 있다.

(2) 비동기(Asynchronous): 비동기(Asynchronous) 방식은 작업이 병렬적으로

실행된다. 즉, 한 작업이 완료되지 않아도 다른 작업이 시작될 수 있다. 예를 들어, 비동기 함수 호출이라면, 해당 함수를 호출하고 바로 반환하여 다음 코드를 실행하며, 함수의 결과는 나중에 콜백 함수 등을 통해 처리한다. 비동기 방식은 동시에 여러 작업을 처리할 수 있어 효율성이 높지만, 코드의 흐름을 이해하거나 디버깅하는 것이 복잡해질 수 있다.

(3) 콜백 함수(Callback Function): 어떤 이벤트가 발생하였을 때 시스템에 의해 호출되는 함수를 의미한다. 콜백 함수는 특정 함수에 파라미터로 전달되어, 해당 함수 내부에서 필요한 시점에 호출된다.

예를 들어, 비동기 작업을 처리할 때 콜백 함수가 많이 사용된다. 비동기 함수를 호출할 때 콜백 함수를 파라미터로 제공하면, 비동기 작업이 완료되었을 때 시스템이 자동으로 해당 콜백 함수를 호출하여 작업의 결과를 처리하게 된다.

▶ Blocking I/O와 Non-Blocking I/O

(1) Blocking I/O(동기식): 특정 작업(주로 I/O 작업)이 완료될 때까지 프로그램 실행을 멈추고 기다리는 방식을 의미한다. 예를 들어, 파일에서 데이터를 읽는 작업을 수행할 때, Blocking 방식을 사용하면 데이터를 모두 읽어들일 때까지 프로그램이 대기하게 된다. 이 방식은 프로그래밍이 단순하고 직관적이지만, 작업이 완료될 때까지 프로그램이 멈추게 되므로 효율성이 떨어질 수 있다.

(2) Non-Blocking I/O(비동기식): 특정 작업이 즉시 완료될 수 없는 경우에도 프로그램 실행을 멈추지 않고 계속 진행하는 방식을 의미한다. 예를 들어, 파일에서 데이터를 읽는 작업을 수행할 때, Non-Blocking 방식을 사용하면 데이터를 바로 읽어올 수 없는 경우에도 프로그램은 계속 실행된다. 이 경우, 데이터가 준비되었을 때 처리할 수 있는 방법(예: 콜백 함수)이 필요하다. Non-Blocking 방식은 동시에 여러 작업을 처리할 수 있어 효율성이 높지만, 프로그래밍이 복잡해질 수 있다.

▶ Blocking I/O와 Non-Blocking I/O의 차이점

- Blocking I/O(동기식)

(1) 요청한 작업을 마칠 때까지 계속 대기한다.

(2) 스레드 관점으로 본다면, 요청한 작업을 마칠 때까지 계속 대기하며 return값을 받을 때까지 한 스레드를 계속 사용/대기한다.

(3) return값을 받아야 끝난다.

- Non-Blocking I/O(비동기식)

(1) 요청한 작업을 마칠 수 없다면 즉시 return한다.

(2) 스레드 관점으로 본다면, 하나의 스레드가 여러 개의 I/O를 처리 가능하다..

5) poll, epoll

▶ poll()과 epoll()은 Linux에서 사용하는 I/O 멀티플렉싱 기술이다. I/O 멀티플렉싱이란 여러 I/O 이벤트를 동시에 감시하고, 준비된 이벤트를 처리하는 기술을 의미한다. 이를 통해 하나의 프로세스나 스레드가 다수의 I/O 연산을 동시에 처리할 수 있다.

(1) poll(): poll() 함수는 여러 파일 디스크립터를 동시에 모니터링하고, 어떤 파일 디스크립터가 읽기, 쓰기, 예외 상황 등의 이벤트를 처리할 준비가 되었는지 알려준다. poll() 함수는 파일 디스크립터의 배열을 인자로 받아, 각 파일 디스크립터의 상태를 체크한다. 이 함수는 Non-Blocking 방식으로 동작하여, I/O 이벤트가 준비되지 않았을 경우에도 즉시 반환한다.(select()와 비슷하다.)

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

- struct pollfd *fds: pollfd 구조체의 배열을 가리키는 포인터이다. 각 pollfd 구조체는 모니터링하려는 파일 디스크립터와 이벤트 플래그, 이벤트 상태를 저장한다.
- nfds_t nfds: 모니터링하려는 파일 디스크립터의 수이다.
- int timeout: 대기할 최대 시간(밀리초)이다. 이 값이 0이면 즉시 반환하고, -1이면 이벤트가 발생할 때까지 무한 대기한다.

(2) epoll(): epoll()은 poll()과 비슷하게 여러 파일 디스크립터를 모니터링하지만, 이벤트가 발생한 파일 디스크립터만을 반환하는 점에서 차이가 있다. poll() 함수는 모든 파일 디스크립터를 순회하며 상태를 체크하지만, epoll() 함수는 준비된 이벤트가 있는 파일 디스크립터만을 처리하므로, 모니터링해야 하는 파일 디스크립터의 수가 많을 경우 효율적이다. 또한, epoll() 함수는 epoll_create(), epoll_ctl(), epoll_wait() 등의 서브함수들을 통해 동작을 제어한다.

- epoll_create()

```
#include <sys/epoll.h>
int epoll_create(int size);
```

: 이 함수는 size만큼 epoll 인스턴스(I/O event 저장공간)를 생성하고, 그에 대한 파일 디스크립터를 반환한다.

- epoll_ctl()

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

: 이 함수는 epoll 인스턴스에 파일 디스크립터를 추가하거나, 제거하거나, 상태를

변경하는 데 사용된다.

- epoll_wait()

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

: 이 함수는 등록된 파일 디스크립터 중에서 이벤트가 발생한 것들을 대기하고, 이벤트가 발생한 디스크립터들을 반환한다.

5. 스레드

1) 스레드 개념

▶ 스레드 (Thread)

- 스레드는 하나의 프로세스 내에서 실행되는 독립적인 실행 흐름이다.
- 같은 프로세스 내에서 여러 개의 스레드가 동시에 작업할 수 있으며, 이들은 같은 메모리 공간과 자원을 공유한다.
- 스레드 간에 데이터 공유와 통신은 비교적 쉽고 빠르게 이루어진다. 하지만 적절한 동기화가 필요할 수 있다.
- 한 스레드가 예외 또는 오류 상황으로 인해 종료되면 다른 스레드들도 영향을 받게 된다.

2) 스레드 생성 및 제어

▶ 스레드 생성(Java)

(1) Thread 클래스를 상속받는 방법

```java
class MyThread extends Thread {
    public void run() {
        // 스레드가 실행할 코드를 여기에 작성한다.
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start(); // 스레드를 시작한다.
    }
}
```

< Thread 클래스 상속 >

: 위 예제에서 MyThread 클래스는 Thread 클래스를 상속받아서 새로운 스레드를 정의한다. run() 메소드는 새로 생성된 스레드에서 실행되는 코드를 정의하는 곳이다.

(2) Runnable 인터페이스를 구현하는 방법

```java
class MyRunnable implements Runnable {
    public void run() {
        // 스레드가 실행할 코드를 여기에 작성한다.
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // 스레드를 시작한다.
    }
}
```

< Runnable 인터페이스 구현 >

: 위 예제에서 MyRunnable 클래스는 Runnable 인터페이스를 구현한다. run() 메소드는 새로 생성된 스레드에서 실행되는 코드를 정의하는 곳이다. Runnable 객체를 Thread 생성자의 인자로 전달하여 스레드를 생성하고, start() 메소드를 호출하여 스레드를 시작한다.

▶ 스레드 제어(Java)

(1) 스레드 실행: 스레드를 실행하기 위해서는 start() 메소드를 호출해야 한다. start() 메소드는 새로운 스레드를 생성하고, 스레드의 run() 메소드를 호출하게 된다.

```
Thread thread = new Thread();
thread.start();
```

(2) 스레드 일시 중지: 스레드를 일시 중지시키기 위해서는 sleep(long milliseconds) 메소드를 사용할 수 있다. 이 메소드는 주어진 밀리초 동안 현재 스레드를 일시 중지시킨다.

```
try {
    Thread.sleep(1000); // 1초 동안 스레드를 일시 중지
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

(3) 스레드 종료: 스레드를 종료하기 위해서는 스레드의 run() 메소드가 종료되도록 해야 한다. 일반적으로 run() 메소드 내의 코드가 모두 실행되면 스레드는 자동적으로 종료된다.

(4) 스레드 대기: 스레드를 대기 상태로 만들기 위해서는 wait() 메소드를 사용할 수 있다. 이 메소드는 Object 클래스의 메소드로, 호출하는 스레드를 대기 상태로 만든다. 이후 notify() 또는 notifyAll() 메소드를 호출하여 대기 중인 스레드를 깨워야 한다.

(5) 스레드 우선순위 설정: 스레드의 우선순위를 설정하려면 setPriority(int newPriority) 메소드를 사용할 수 있다. 이 메소드를 사용하면 스레드 스케줄러는 우선순위가 높은 스레드에게 더 많은 실행 시간을 할당하게 된다.

```
thread.setPriority(Thread.MAX_PRIORITY); // 스레드 우선순위를 최대로 설정
```

(6) 스레드 동기화: 여러 스레드가 공유 데이터를 안전하게 처리하도록 하기 위한 방법으로는 synchronized 키워드를 사용하는 방법과 Lock 인터페이스를 활용하는 방법이 있다.

3) 스레드간 동기화

▶ 스레드간 동기화는 여러 스레드가 동시에 공유 데이터를 접근하거나 수정할 때, 데이터의 일관성을 유지하기 위한 방법이다. 스레드 동기화 기법에는 뮤텍스(Mutex), 세마포어(Semaphore), 모니터(Monitor), 조건 변수(Condition Variables), 배리어(Barrier) 등이 있다. 이는 스레드가 공유 데이터를 동시에 수

정함으로써 발생하는 문제들, 예를 들어 경쟁 조건(race condition)이나 데드락(deadlock) 등을 해결하기 위해 사용된다. 자바에서는 다음과 같은 방법들을 통해 스레드간 동기화를 수행할 수 있다.

(1) synchronized 키워드: synchronized 키워드는 특정 객체에 대해 한 번에 하나의 스레드만 접근할 수 있도록 락(lock)을 걸어주는 역할을 한다. synchronized 키워드는 메소드 또는 블록에 적용할 수 있다.

```
public synchronized void myMethod() {
    // 동기화 블록
}

synchronized(myObject) {
    // 동기화 블록
}
```

(2) volatile 키워드: volatile 키워드는 변수를 메인 메모리에 저장하게 하여, 여러 스레드가 동일한 값을 보도록 한다. 이는 스레드가 자신의 작업 메모리를 사용하지 않고 매번 메인 메모리에서 값을 읽어오도록 강제함으로써, 변수의 가장 최신의 값을 볼 수 있게 한다.

```
private volatile boolean myFlag;
```

(3) Lock 인터페이스: java.util.concurrent.locks 패키지의 Lock 인터페이스를 사용하여 더 세밀한 제어를 할 수 있다. Lock 인터페이스는 lock(), unlock(), tryLock() 등의 메소드를 제공한다.

```
Lock lock = new ReentrantLock();

lock.lock();
try {
    // 동기화 블록
} finally {
    lock.unlock();
}
```

(4) Atomic 클래스: java.util.concurrent.atomic 패키지의 Atomic 클래스들은 원자적 연산을 지원한다. 원자적 연산은 중간에 인터럽트 되지 않는 연산을 의미한다. 예를 들어, AtomicInteger 클래스는 incrementAndGet(), decrementAndGet() 등의 메소드를 제공하여 원자적 연산을 수행할 수 있다.

```
AtomicInteger atomicInteger = new AtomicInteger(0);

int newValue = atomicInteger.incrementAndGet();
```

4) 프로세스의 모듈화

▶ 프로세스 모듈화는 소프트웨어 개발에서 중요한 개념 중 하나로, 프로세스를 재사용 가능한 개별 모듈로 분리하는 것을 의미한다. 이는 코드의 가독성, 유지 보수성, 테스트 용이성을 향상시키며, 재사용성을 높여 개발 시간을 절약하는 데 도움이 된다.

프로세스 모듈화의 핵심 개념은 다음과 같다.

(1) 단일 책임 원칙(Single Responsibility Principle): 각 모듈은 하나의 기능만 수행하도록 설계되어야 한다. 이렇게 하면 각 모듈이 자신의 역할에만 집중할 수 있으므로 코드의 가독성과 유지 보수성이 향상된다.

(2) 모듈 간의 느슨한 결합(Loose Coupling): 모듈 간의 의존성은 최소화되어야 한다. 이렇게 하면 한 모듈의 변경이 다른 모듈에 미치는 영향을 최소화할 수 있다.

(3) 모듈의 재사용성(Reusability): 모듈은 가능한 한 범용적으로 설계되어야 하며, 특정 목적이나 상황에 구애받지 않도록 해야 한다. 이렇게 하면 해당 모듈을 다양한 상황에서 재사용할 수 있다.

(4) 모듈의 독립성(Independence): 각 모듈은 독립적으로 작동할 수 있어야 한다. 이는 모듈 간의 상호 작용을 최소화하며, 모듈을 독립적으로 테스트하거나 업데이트할 수 있게 한다.

5) pthread API : 스레드의 생성, 종료

▶ 스레드의 생성은 주로 POSIX(포지스) 스레드 라이브러리인 pthread를 사용하여 수행된다. pthread 라이브러리는 스레드의 생성, 종료, 동기화 등 다양한 스레드 관련 작업을 수행할 수 있는 함수를 제공한다.

▶ 스레드의 생성 pthread_create()

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine) (void *), void *arg);
```

- pthread_t *thread: 새로 생성된 스레드의 ID를 저장할 변수의 주소이다.

- const pthread_attr_t *attr: 새로 생성하는 스레드의 속성을 지정한다. NULL을 전달하면 기본 속성의 스레드가 생성된다.
- void *(*start_routine) (void *): 새로 생성하는 스레드에서 실행할 함수의 주소이다. 이 함수는 void 포인터를 인자로 받고, void 포인터를 반환하는 형태여야 한다.
- void *arg: start_routine 함수에 전달할 인자이다.

▶ POSIX 표준에 따른 pthreads 라이브러리에서 스레드의 종료는 주로 다음 세 가지 방법으로 이루어진다.

(1) 스레드 함수의 리턴: 스레드가 실행하는 함수가 리턴을 수행하면 해당 스레드는 종료된다. 이때 함수의 리턴 값은 다른 스레드에서 pthread_join() 함수를 호출하여 얻을 수 있다.

```
void* myThread(void* arg) {
    // do something...
    return NULL;
}
```

(2) pthread_exit() 함수의 호출: 스레드는 언제든지 pthread_exit() 함수를 호출하여 자신을 명시적으로 종료할 수 있다. 이 함수의 인자는 종료 상태를 나타내며, 다른 스레드에서 pthread_join() 함수를 호출하여 이 값을 얻을 수 있다.

```
void* myThread(void* arg) {
    // do something...
    pthread_exit(NULL);
}
```

(3) pthread_cancel() 함수의 호출: 다른 스레드는 pthread_cancel() 함수를 호출하여 특정 스레드를 종료시킬 수 있다. 이 함수는 종료할 스레드의 ID를 인자로 받는다. 하지만 이 방법은 스레드가 어느 시점에서 종료되는지 알 수 없으므로, 리소스의 정리 등에 문제가 생길 수 있다.

```
pthread_t thread;
// create a thread...
pthread_cancel(thread);
```

6) Mutex와 조건 변수

▶ 상호배제(Mutual Exclusion: Mutex)는 여러 개의 프로세스나 스레드가 하나의 공유된 자원에 동시에 접근하지 못하도록 하는 개념이다. 상호배제는 한 번에

하나의 스레드나 프로세스만이 특정 코드 섹션(일반적으로 공유 자원에 대한 접근)을 실행할 수 있도록 한다.

▶ 상호배제 매커니즘

(1) 임계 영역 (Critical Section): 공유 자원에 접근하는 코드 영역을 임계 영역이라고 한다. 임계 영역은 상호배제가 필요한 부분으로, 한 번에 하나의 프로세스 또는 스레드만이 해당 영역에 진입할 수 있다.

(2) 락 (Lock): 락은 상호배제를 구현하기 위한 동기화 기법 중 하나이다. 락은 임계 영역 진입 전에 획득되어 다른 프로세스나 스레드가 해당 임계 영역에 진입하지 못하도록 막는다. 락을 획득한 후 작업이 완료되면 반드시 락을 해제해야 다른 프로세스나 스레드가 해당 임계 영역에 접근할 수 있게 된다. 공유 자원이 사용되고 있을 경우에 Lock변수의 값은 1이며, 공유 자원이 비어있을 경우에 Lock 변수의 값은 0으로 나타낸다(세마포어와 반대).

(3) 동기화 기법(Synchronization) : 두 개 이상의 프로세스를 한 시점에서 자원 접근을 동시에 처리하게 할 수 없게 하기 위해 각 프로세스에 대한 자원 접근 처리 순서를 결정하는 것으로 상호 배제의 한 형태이다. 대표적인 동기화 기법으로는 세마포어와 모니터가 있다.

▶ pthread API를 이용한 Mutex

(1) Mutex 생성: pthread_mutex_t 형식의 변수를 선언하여 Mutex를 생성할 수 있다. 이 변수는 Mutex를 위한 메모리 공간을 할당하며, 이후 Mutex의 연산에 사용된다.

```
pthread_mutex_t mutex;
```

(2) Mutex 초기화: pthread_mutex_init 함수를 사용하여 Mutex를 초기화한다. 이 함수는 첫 번째 인자로 Mutex의 주소를, 두 번째 인자로 Mutex의 속성을 지정한다. 일반적으로 두 번째 인자는 NULL로 설정하여 기본 속성을 사용한다.

```
pthread_mutex_init(&mutex, NULL);
```

(3) Mutex 잠금: pthread_mutex_lock 함수를 사용하여 Mutex를 잠글 수 있다. 이 함수는 Mutex를 인자로 받으며, 해당 Mutex가 이미 잠겨 있다면 호출한 스레드는 Mutex가 해제될 때까지 대기한다.

```
pthread_mutex_lock(&mutex);
// critical section...
```

(4) Mutex 해제: pthread_mutex_unlock 함수를 사용하여 Mutex를 해제한다. 이 함수는 Mutex를 인자로 받으며, Mutex를 해제하여 다른 스레드가 해당 Mutex를 잠글 수 있도록 한다.

```
// critical section...
pthread_mutex_unlock(&mutex);
```

(5) Mutex 제거: pthread_mutex_destroy 함수를 사용하여 Mutex를 제거한다. 이 함수는 Mutex를 인자로 받으며, Mutex를 사용한 후 반드시 호출하여야 한다.

```
pthread_mutex_destroy(&mutex);
```

▶ 조건 변수(Condition Variables)는 여러 스레드가 특정 조건이 충족될 때까지 기다리도록 하는 동기화 메커니즘이다. Condition Variables는 주로 Mutex 또는 Monitor와 함께 사용되며, Mutex로 보호되는 데이터에 대한 특정 조건의 충족 여부를 검사하고, 조건이 충족되지 않은 경우 스레드를 대기 상태로 전환한다. 조건 변수는 단지 변수를 통해 신호를 주고 받는 기능만을 제공할 뿐 자체 잠금 기능이 없다.

▶ Condition Variables는 다음 두 가지 주요 연산을 제공한다.

(1) wait: 이 연산은 현재 스레드를 대기 상태로 전환하고, 연결된 Mutex의 잠금을 해제한다. 이를 통해 다른 스레드가 해당 Mutex를 획득하고 공유 데이터를 수정할 수 있게 한다. 조건이 충족되면 (일반적으로 다른 스레드에 의해 signal 또는 broadcast 연산이 호출됨), wait 연산은 반환하고 Mutex의 잠금을 다시 획득한다.(주의할 점은 wait()함수를 호출하기 전에 먼저 뮤텍스의 잠금을 설정해야 한다. 조건 변수의 신호를 받았다 하더라도 뮤텍스의 잠금이 설정되어 있으면 해제될 때까지 대기하게 된다.)

(2) signal/broadcast: signal 연산은 하나의 대기 중인 스레드를 깨우며, broadcast 연산은 모든 대기 중인 스레드를 깨운다. 이 연산들은 대기 중인 스레드가 조건을 다시 검사하고 계속 실행하도록 한다.

▶ pthread API를 이용한 Condition Variables

(1) Condition Variables 생성: pthread_cond_t 형식의 변수를 선언하여 Condition Variables를 생성할 수 있다. 이 변수는 Condition Variables를 위한 메모리 공간을 할당하며, 이후 Condition Variables의 연산에 사용된다.

```
pthread_cond_t cond;
```

(2) Condition Variables 초기화: pthread_cond_init 함수를 사용하여 Condition Variables를 초기화한다. 이 함수는 첫 번째 인자로 Condition Variables의 주소를, 두 번째 인자로 Condition Variables의 속성을 지정한다. 일반적으로 두 번째 인자는 NULL로 설정하여 기본 속성을 사용한다.

```
pthread_cond_init(&cond, NULL);
```

(3) wait 연산: pthread_cond_wait 함수를 사용하여 특정 조건이 충족될 때까지 스레드를 대기 상태로 만든다. 이 함수는 첫 번째 인자로 Condition Variables를, 두 번째 인자로 Mutex를 받는다. 이 함수는 Mutex를 자동으로 해제하고, 조건이 충족될 때까지 스레드를 대기 상태로 만든다.

```
pthread_mutex_t mutex;
// initialize mutex...
pthread_mutex_lock(&mutex);
while(condition is not met) {
    pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);
```

(4) signal/broadcast 연산: pthread_cond_signal 또는 pthread_cond_broadcast 함수를 사용하여 대기 중인 스레드를 깨울 수 있다. pthread_cond_signal 함수는 하나의 스레드를 깨우며, pthread_cond_broadcast 함수는 모든 스레드를 깨운다.

```
// condition has been met...
pthread_cond_signal(&cond); // or pthread_cond_broadcast(&cond);
```

(5) Condition Variables 제거: pthread_cond_destroy 함수를 사용하여 Condition Variables를 제거한다. 이 함수는 Condition Variables를 인자로 받으며, Condition Variables를 사용한 후 반드시 호출하여야 한다.

```
pthread_cond_destroy(&cond);
```

7) Barrier, 여러 가지 locks

▶ Barrier는 여러 스레드가 동시에 특정 지점까지 도달하면 실행을 계속하도록 하는 동기화 메커니즘이다. Barrier는 주로 병렬 컴퓨팅에서 사용되며, 특정 단계가 모든 스레드에 의해 완료된 후에만 다음 단계로 진행되도록 하는데 사용된다.

▶ Barrier의 동작 과정

(1) Barrier는 초기화 단계에서 특정 개수의 스레드를 기다리도록 설정된다.

(2) 스레드가 Barrier에 도달하면, Barrier는 해당 스레드를 대기 상태로 전환시킨다. 이는 Barrier에 설정된 스레드 수만큼 스레드가 도달할 때까지 반복된다.

(3) 설정된 수의 스레드가 모두 Barrier에 도달하면, Barrier는 모든 대기 중인 스레드를 깨우고 실행을 계속하도록 한다.

▶ Read-Write Lock

: Read-Write Lock은 리소스에 대한 읽기와 쓰기 연산을 분리한다. 여러 스레드가 동시에 리소스를 읽을 수 있지만, 쓰기 연산은 한 번에 하나의 스레드만 수행할 수 있다.

▶ Spinlock

: Spinlock은 Mutex와 비슷하지만, 잠금 상태인 동안 스레드는 대기 상태가 아닌 "spin" 상태로 유지된다. 즉, 스레드는 계속해서 잠금 해제를 확인하며 CPU 시간을 소비한다. Spinlock은 잠금 시간이 짧을 것으로 예상되는 경우에 유용하며, 컨텍스트 전환 오버헤드를 줄이는 데 도움이 된다.

8) 스레드의 응용

▶ 스레드는 여러 가지 상황에서 용이하게 사용되며 주요 응용 사례는 다음과 같다.

(1) 병렬 컴퓨팅: 멀티코어 또는 다중 프로세서 시스템에서는 여러 스레드를 동시에 실행하여 작업을 병렬로 처리할 수 있다. 이를 통해 전체 작업 시간을 줄이고 시스템의 처리량을 향상시킬 수 있다.

(2) 비동기 처리: 스레드는 비동기 처리에 유용하게 사용된다. 예를 들어, 웹 서버는 각 클라이언트 요청을 별도의 스레드로 처리하여, 하나의 요청이 블로킹되더라도 다른 요청에 영향을 미치지 않도록 할 수 있다.

(3) 백그라운드 작업: 스레드는 백그라운드에서 실행되는 작업에 유용하게 사용된다. 예를 들어, 사용자 인터페이스를 제공하는 애플리케이션은 사용자 입력을 처리하는 메인 스레드와 별도로, 네트워크 통신이나 데이터베이스 액세스 등의 작업을 백그라운드 스레드에서 처리할 수 있다.

(4) 실시간 처리: 실시간 시스템에서는 스레드의 우선순위를 조절하여 중요한 작업을 먼저 처리하도록 할 수 있다. 이는 실시간 제약 조건을 만족시키는 데 필요하다.

(5) 게임 개발: 게임 개발에서는 물리 시뮬레이션, AI, 렌더링 등 여러 작업을 동시에 처리해야 하는 경우가 많으므로 스레드가 활발하게 사용된다.

(6) 데이터베이스 시스템: 데이터베이스 시스템에서는 동시에 여러 클라이언트의 요청을 처리해야 하므로, 각 요청을 별도의 스레드로 처리하는 것이 일반적이다.

6. 시그널 기본

1) 시그널의 정의

▶ 시그널 (Signal): 시그널은 일종의 소프트웨어 인터럽트로, 프로세스에게 특정 사건이 발생했음을 알리는 방법이다. 예를 들어, 프로세스가 정의할 수 없는 연산을 수행하려고 하면 운영체제는 해당 프로세스에게 SIGFPE (부동소수점 오류) 시그널을 보낸다.

2) 시그널 핸들러

▶ 시그널 핸들러(Signal Handler) : 프로세스가 특정 시그널을 포착했을 때 수행해야 할 별도의 함수이다. 프로세스는 시그널을 포착하면 현재 작업을 일시 중단하고 시그널 핸들러를 실행하고, 실행이 끝나면 중단된 작업을 재개한다.

3) 시그널전송 에러처리

▶ 시그널은 운영체제가 프로세스나 스레드에게 특정 이벤트가 발생했음을 알리는 메커니즘이다. 시그널을 전송하는 함수 중 하나는 kill() 함수이다. 이 함수는 특정 프로세스에게 시그널을 전송하는 역할을 한다. 그러나 kill() 함수를 호출할 때 발생할 수 있는 여러 가지 에러 상황들이 있다.

(1) ESRCH 에러: 이 에러는 지정된 프로세스 ID가 존재하지 않을 때 발생한다. 이 경우, 프로세스 ID를 다시 확인하거나, 프로세스가 종료되었는지 확인하는 등의 조치가 필요하다.

(2) EPERM 에러: 이 에러는 시그널을 전송하는 프로세스가 대상 프로세스에게 시그널을 보낼 권한이 없을 때 발생한다. 이 경우, 권한을 확인하거나 필요하다면 슈퍼유저 권한으로 실행하는 등의 조치가 필요하다.

(3) EINVAL 에러: 이 에러는 전송하려는 시그널 번호가 유효하지 않을 때 발생한다. 이 경우, 시그널 번호를 다시 확인하거나, 지원되는 시그널 번호 범위 내에 있는지 확인하는 등의 조치가 필요하다.

▶ 에러 처리 방법

(1) 에러 확인: 시그널 관련 함수를 호출한 후에는 항상 반환값을 확인하여 에러가 발생했는지 확인해야 한다. 예를 들어, kill(), sigaction(), signal(), sigprocmask() 등의 함수는 실패하면 -1을 반환한다.

(2) 에러 유형 확인: 함수가 에러를 반환하면 errno 전역 변수를 확인하여 에러의 원인을 파악할 수 있다. errno는 마지막으로 실패한 시스템 호출 또는 라이브러리 함수에서 발생한 에러 코드를 저장한다.

(3) 에러 처리: 에러의 원인에 따라 적절한 에러 처리를 수행해야 한다. 에러 메시지를 출력하거나, 재시도하거나, 프로그램을 종료하는 등의 조치를 취할 수 있다.

7. 컴퓨터 네트워크

1) 컴퓨터 네트워크 기본

▶ 컴퓨터 네트워크는 컴퓨터들이 서로 통신할 수 있도록 연결된 시스템을 의미한다. 이는 데이터를 교환하고 공유함으로써 효과적인 정보 전달과 자원 공유를 가능하게 한다.

▶ 컴퓨터 네트워크 주요 개념

(1) 프로토콜(Protocol)

: 컴퓨터 네트워크에서 컴퓨터나 장치 간의 통신을 위해 정의된 규칙과 규약의 집합을 말한다. 이 규칙들은 데이터의 형식, 전송 방식, 오류 제어, 보안 등의 다양한 측면을 다루며, 효율적이고 안정적인 통신을 가능하게 한다. 각각의 프로토콜은 고유한 식별자를 가지며, 이를 통해 통신에 참여하는 장치들은 어떤 프로토콜을 사용해야 하는지 알 수 있다. 프로토콜의 종류로는 TCP(Transmission Control Protocol)/IP(Internet Protocol) HTTP(Hypertext Transfer Protocol), FTP(File Transfer Protocol), SMTP(Simple Mail Transfer Protocol), DNS(Domain Name System) 등이 있다.

(2) IP 주소(Internet Protocol Address): 인터넷 프로토콜(IP) 네트워크에서 장치를 식별하고 위치를 지정하는 데 사용되는 숫자 또는 문자열이다. 컴퓨터, 스마트폰, 프린터, 라우터 등 네트워크에 연결된 모든 장치는 고유한 IP 주소를 가지고 있다.

- IPv4: 현재 가장 일반적으로 사용되는 버전이다. IPv4 주소는 4개의 1바이트(8비트) 숫자로 구성되며, 각 숫자는 0부터 255까지의 범위를 가진다. 이 숫자들은 점('.')으로 구분되어 표현된다. 예를 들어, "192.168.0.1"은 IPv4 주소이다.
- IPv6: IPv6는 IPv4 주소의 고갈 문제를 해결하기 위해 만들어진 새로운 버전이다. IPv6 주소는 16바이트(128비트) 숫자로 구성되며, 8개의 4개 문자 그룹으로 표현된다. 각 그룹은 콜론(':')으로 구분되어 표현되며, 각 문자는 16진수 형태로 표현된다. 예를 들어, "2001:0db8:85a3:0000:0000:8a2e:0370:7334"는 IPv6 주소이다.
- IP 주소는 정적(static) 또는 동적(dynamic)일 수 있다. 정적 IP 주소는 수동으로 설정되며, 네트워크에 연결되는 동안 일정하게 유지된다. 반면에 동적 IP 주소는 DHCP(Dynamic Host Configuration Protocol)와 같은 프로토콜을 통해 자동으로 할당되며, 시간이 지남에 따라 변경될 수 있다.
- 또한, IP 주소는 공용(public) 또는 사설(private)일 수 있다. 공용 IP 주소는

인터넷 상에서 고유하며, 외부 네트워크에서 접근 가능하다. 반면에 사설 IP 주소는 특정 로컬 네트워크 내에서만 유니크하며, 외부 네트워크에서는 접근할 수 없다.

(3) 서브넷 마스크(Subnet Mask): IP 주소의 네트워크 부분과 호스트 부분을 구분하는 데 사용되는 32비트 숫자이다. 이는 IP 주소와 동일하게 4개의 옥텟으로 표현되며, 각 옥텟은 점('.')으로 구분된다. 서브넷 마스크는 일반적으로 이진수로 표현될 때 '1'들이 연속적으로 나오고, 그 뒤에 '0'들이 연속적으로 나오는 형태를 가진다.

예를 들어, "255.255.255.0"은 흔히 사용되는 서브넷 마스크이다. 이 서브넷 마스크를 이진수로 표현하면 "11111111.11111111.11111111.00000000"이 된다. 이 서브넷 마스크에서 '1'이 차지하는 비트는 네트워크 부분을, '0'이 차지하는 비트는 호스트 부분을 나타낸다.

- 서브넷 마스크는 IP 주소와 함께 사용되어 특정 IP 주소가 어느 네트워크에 속하는지를 결정하는 데 사용된다. IP 주소의 비트와 서브넷 마스크의 비트를 각각 AND 연산하면 네트워크 주소를 얻을 수 있다.
- 서브네팅(Subnetting)은 큰 네트워크를 작은 서브넷으로 분할하는 과정에서 사용되며, 이는 IP 주소 공간을 효율적으로 관리하고 네트워크 트래픽을 제어하는 데 도움이 된다. 서브넷 마스크의 크기에 따라 서브넷의 수와 각 서브넷의 호스트 수가 결정된다.

(4) 게이트웨이(Gateway): 컴퓨터 네트워크에서 서로 다른 네트워크를 연결하는 데 사용되는 하드웨어 또는 소프트웨어를 나타낸다. 게이트웨이는 서로 다른 네트워크 프로토콜 간의 통신을 가능하게 하며, 이를 통해 서로 다른 네트워크 간의 데이터 전송을 조정한다.

- 게이트웨이는 통신 프로토콜을 변환하여 서로 다른 네트워크 간의 통신을 가능하게 한다. 예를 들어, 로컬 네트워크와 인터넷 간의 통신을 가능하게 하는 라우터는 게이트웨이의 일종이다.
- 게이트웨이는 네트워크 계층에서 작동하며, IP 주소를 사용하여 패킷을 전달한다. 게이트웨이는 패킷의 목적지 IP 주소를 확인하고, 해당 패킷을 적절한 네트워크 또는 장치로 전달하는 라우팅 테이블을 사용한다.
- 게이트웨이는 또한 네트워크 보안을 향상시키는 데 사용될 수 있다. 예를 들어, 방화벽은 게이트웨이의 일종으로, 네트워크 트래픽을 모니터링하고 악성 트래픽을 차단하여 네트워크를 보호한다.
- 게이트웨이는 네트워크에 연결된 장치가 인터넷과 같은 외부 네트워크에 접근할 수 있도록 하는 기본 경로 역할을 하며, 이것이 '디폴트 게이트웨이(Default

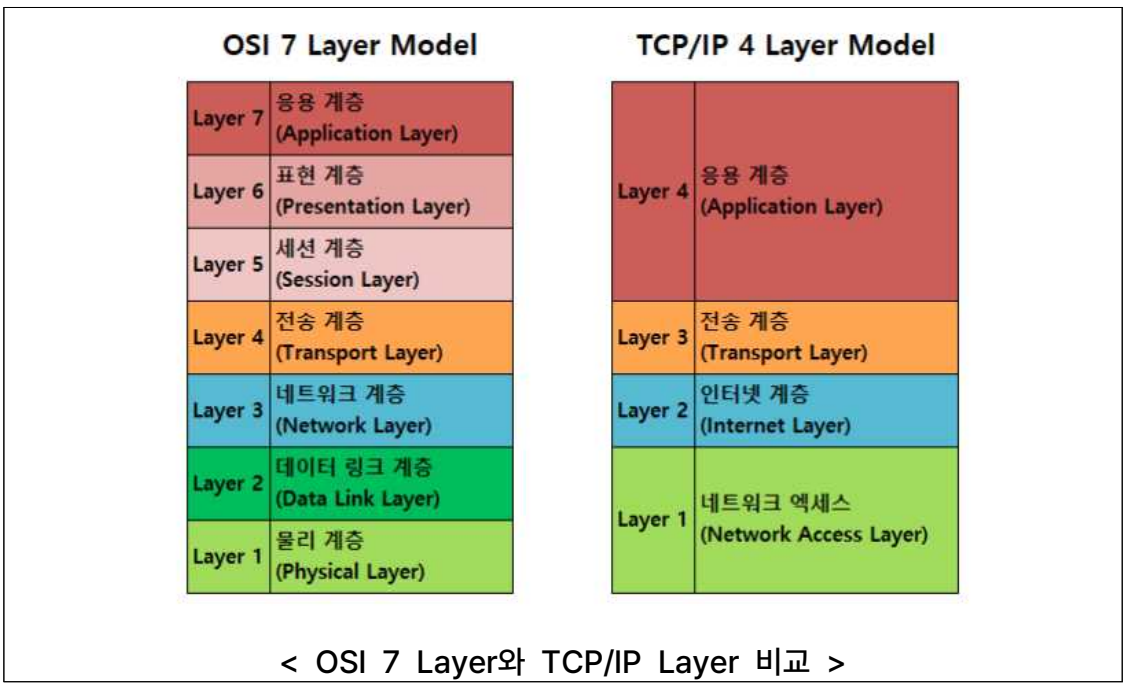
Gateway)'라 불리는 이유이다.
(5) 라우터(Router): 네트워크에서 패킷을 전송하는 장치로, 한 네트워크와 다른 네트워크 간에 정보를 주고받는 역할을 한다. 라우터는 IP주소를 기반으로 패킷의 소스와 목적지를 파악하고, 이를 바탕으로 패킷을 올바른 방향으로 전달한다.
(6) 패킷(Packet): 컴퓨터 네트워크에서 데이터를 전송하는 기본 단위이다. 디지털 네트워크에서 데이터는 패킷으로 나뉘어 전송되고, 수신측에서 다시 원래의 형태로 재조립된다.
- 패킷은 헤더(Header)와 페이로드(Payload) 두 부분으로 구성된다.
- 헤더: 헤더는 패킷의 메타데이터를 포함하며, 패킷의 출발지와 목적지, 패킷의 크기, 패킷의 순서, 프로토콜 타입 등의 정보를 포함한다. 헤더의 정보는 패킷이 올바르게 전송되고 수신되는데 필요한 정보를 제공한다.
- 페이로드: 페이로드는 실제로 전송되는 데이터를 포함한다. 이는 문서, 이미지, 비디오, 오디오 등 다양한 형태의 데이터가 될 수 있다.
- 패킷 기반 네트워크에서 데이터의 전송은 패킷 스위칭(Packet Switching)이라는 방법을 통해 이루어진다. 이는 각 패킷이 네트워크를 통해 독립적으로 전송되고, 각 패킷이 서로 다른 경로를 통해 목적지에 도달할 수 있음을 의미한다. 이 방식은 네트워크의 효율성을 높이고, 네트워크의 병목 현상을 줄이는 데 도움이 된다.
(7) DNS(Domain Name System): 인터넷에서 도메인 이름을 IP 주소로 변환하는 시스템이다. DNS는 사람들이 웹사이트를 방문하거나 인터넷 서비스를 사용할 때, 기억하기 쉬운 도메인 이름(예: www.example.com)을 해당 서비스의 실제 IP 주소(예: 192.0.2.1)로 변환하는 역할을 한다.
(8) HTTP(Hypertext Transfer Protocol): 웹 브라우저와 웹 서버 간에 정보를 주고받는 방법을 정의하는 프로토콜이다. 이는 월드 와이드 웹(World Wide Web)의 핵심 구성 요소로, 웹 페이지의 텍스트, 이미지, 비디오 등의 리소스를 전송하는 데 사용된다.
- HTTP는 클라이언트-서버 모델을 사용한다. 사용자의 웹 브라우저(클라이언트)가 웹 서버에 요청(Request)을 보내고, 웹 서버가 해당 요청에 대한 응답(Response)을 반환하는 방식이다.
(9) VPN(Virtual Private Network): 인터넷을 통해 가상의 사설 네트워크를 생성하는 기술이다. VPN을 사용하면, 사용자는 인터넷을 통해 안전하게 원격 네트워크에 연결할 수 있다.
- 보안: VPN은 인터넷 연결에 암호화 계층을 추가하여, 외부의 눈에서 데이터를 보호한다. 이는 특히 공공 와이파이와 같이 안전하지 않은 네트워크를 사용할 때

중요하다.
- 개인정보 보호: VPN은 사용자의 실제 IP 주소를 숨기고, VPN 서버의 IP 주소로 대체한다. 이를 통해 온라인 활동의 추적을 방지하고 개인정보를 보호할 수 있다.
- 지리적 제한 우회: VPN은 사용자가 원격 VPN 서버를 통해 인터넷에 접속하므로, 사용자의 실제 위치와 상관없이 특정 지역의 인터넷 서비스에 접근할 수 있게 해준다. 이를 통해 지역에 따라 제한된 콘텐츠를 볼 수 있게 된다.
(10) XML(eXtensible Markup Language): 데이터를 저장하고 전송하는 데 사용되는 마크업 언어이다. HTML과 유사하게 태그를 사용하여 데이터를 구조화하지만, HTML이 웹 페이지의 표현을 위해 설계된 반면, XML은 데이터의 구조를 정의하는 데 초점이 맞춰져 있다. XML은 사용자가 직접 태그를 정의할 수 있어, 다양한 종류의 데이터를 표현하는 데 유연하다.
- 태그(Tag): XML 데이터를 구조화하는 데 사용되는 기본 단위이다. 시작 태그(<tag>)와 종료 태그(</tag>)로 이루어져 있다. 태그 이름은 사용자가 임의로 지정할 수 있다.
- 요소(Element): 시작 태그와 종료 태그, 그리고 그 사이에 위치한 내용을 합친 것을 요소라고 한다. 요소는 다른 요소를 포함할 수 있으며, 이를 통해 계층적인 데이터 구조를 만들 수 있다.
- 속성(Attribute): 요소의 추가적인 정보를 제공하는 데 사용된다. 속성은 이름과 값의 쌍으로 이루어져 있으며, 시작 태그 안에 위치한다. 예를 들어, <book title="AI Guide">에서 'title'이 속성의 이름이고, "AI Guide"는 속성의 값이다.
- 선언(Declaration): XML 문서의 최상단에 위치하며, 이 문서가 XML 문서임을 알리고, 사용한 XML의 버전과 문자 인코딩 방식을 명시한다. 예를 들어, <?xml version="1.0" encoding="UTF-8"?>는 XML 선언이다.
- 주석(Comment): XML 문서 안에서 설명을 추가하거나, 코드를 비활성화하는 데 사용된다. 주석은 <!-- comment --> 형식으로 표현되며, XML 파서에 의해 무시된다.
- 레이아웃(Layout): 레이아웃은 디자인 요소(view)가 배치되는 구조나 계획을 의미한다. 웹 페이지, 앱, 인쇄 매체 등에서 요소의 위치, 크기, 형태 등을 결정하는 데 사용된다.
- 뷰(View): 앱 안에 들어가는 각각의 화면 구성 요소이다.
- 위젯(Widget): 레이아웃 안에서 컨트롤 역할을 한다.
(11) HTML(HyperText Markup Language): 웹 페이지를 만들기 위한 표준 마크업 언어이다. 웹 페이지의 구조를 정의하고, 웹 브라우저에게 웹 콘텐츠를 어떻

게 표시할지 알려주는 역할을 한다.

- 태그(Tag): HTML 문서의 기본 구성 단위로, 웹 브라우저에게 어떤 정보를 어떻게 표시할지 알려준다. 대부분의 태그는 시작 태그(<tag>)와 종료 태그(</tag>)로 이루어져 있다. 예를 들어, <h1>은 제목을 나타내는 태그이며, <p>는 단락을 나타내는 태그이다.
- 요소(Element): 시작 태그, 종료 태그, 그리고 그 사이의 내용을 합친 것을 요소라고 한다. 요소는 다른 요소를 포함할 수 있다.
- 속성(Attribute): 태그의 추가 정보를 제공하는 데 사용된다. 속성은 이름과 값의 쌍으로 이루어져 있으며, 시작 태그 안에 위치한다. 예를 들어, <img src="image.jpg">에서 'src'는 속성 이름이고, "image.jpg"는 속성 값이다.
- HTML 문서 구조: HTML 문서는 보통 <!DOCTYPE html>로 시작하며, <html> 태그로 전체 내용을 감싸고, <head> 태그 안에는 메타 정보와 CSS, JavaScript 파일 링크 등이 위치하고, <body> 태그 안에는 실제로 웹 브라우저에 표시될 콘텐츠가 위치한다.

2) OSI 계층 프로토콜



< OSI 7 Layer와 TCP/IP Layer 비교 >

: OSI(Open Systems Interconnection) 모델은 ISO(국제표준기구)에서 만든 네트워킹 프로토콜의 상호작용을 이해하기 위한 개념적 프레임워크이다. 이 모델은 7개의 계층으로 구성되어 있으며, 각 계층은 특정한 네트워크 기능을 담당한다.

(1) 물리 계층(Physical Layer): 비트 단위의 데이터를 전송하는 데에 필요한 하

드웨어 인터페이스를 관리한다. 이 계층은 케이블, 허브, 리피터 등의 장비를 포함한다.

(2) 데이터 링크 계층(Data Link Layer): 물리 계층을 통해 전송되는 데이터의 오류 검출 및 흐름 제어를 담당한다. 이 계층의 사례로는 이더넷(Ethernet)과 PPP(Point-to-Point Protocol)가 있다.

(3) 네트워크 계층(Network Layer): 패킷의 경로 설정과 IP 주소 지정을 담당한다. IP(Internet Protocol)와 ICMP(Internet Control Message Protocol) 등이 이 계층에 속한다.

(4) 전송 계층(Transport Layer): 데이터 전송의 신뢰성과 흐름 제어를 담당한다. TCP(Transmission Control Protocol)와 UDP(User Datagram Protocol)가 이 계층에 속한다.

(5) 세션 계층(Session Layer): 컴퓨터들 사이의 통신 세션을 설정, 관리, 종료한다. 이 계층의 프로토콜로는 NetBIOS 등이 있다.

(6) 표현 계층(Presentation Layer): 데이터의 암호화, 압축, 변환을 담당하여 애플리케이션 계층에서 이해할 수 있는 형태로 표현한다. 이 계층의 사례로는 SSL(Secure Sockets Layer)과 TLS(Transport Layer Security)가 있다.

(7) 응용 계층(Application Layer): 최종 사용자에게 제공되는 애플리케이션과 직접적으로 상호작용한다. HTTP, FTP(File Transfer Protocol), SMTP(Simple Mail Transfer Protocol) 등이 이 계층에 속한다.

3) TCP, UDP, IP, TFTP

▶ TCP(Transmission Control Protocol)

: 인터넷 프로토콜 스택인 TCP/IP의 주요 프로토콜 중 하나로, 데이터를 패킷 단위로 나누어 인터넷을 통해 전송하고, 이를 원래의 데이터로 다시 조립하는 역할을 한다. TCP는 다음과 같은 특징을 가진다.

(1) 연결 지향성: TCP는 통신을 시작하기 전에 송신자와 수신자 간에 연결을 설정한다. 이를 통해 데이터가 정확하게 전송될 수 있도록 한다.

(2) 신뢰성: TCP는 패킷의 순서 보장, 에러 검출 및 재전송, 흐름 제어 등을 통해 데이터의 신뢰성 있는 전송을 보장한다. 만약 패킷이 손상되었거나 분실되면, TCP는 해당 패킷을 재전송한다.

(3) 흐름 제어 및 혼잡 제어: TCP는 송신자와 수신자의 데이터 처리 속도 차이를 해결하기 위해 흐름 제어를 제공하며, 네트워크의 혼잡 상황을 관리하기 위해 혼잡 제어 메커니즘을 가지고 있다.

▶ IP(Internet Protocol)

: 인터넷 프로토콜 스택인 TCP/IP에서 가장 핵심적인 프로토콜 중 하나로, 네트워크를 통해 데이터 패킷을 전송하는 방법을 정의한다. IP의 주요 역할은 데이터 패킷을 송신지에서 목적지까지 올바르게 전달하는 것이다. 이를 위해 IP는 각각 고유의 주소를 가지며 각 패킷에 대한 메타데이터를 포함하는 IP 헤더를 사용한다. IP 헤더에는 다음과 같은 정보가 포함된다.

(1) 버전: 사용된 IP 프로토콜의 버전을 나타낸다. 현재는 IPv4와 IPv6가 주로 사용된다.

(2) 소스 IP 주소: 패킷을 보낸 장치의 IP 주소이다.

(3) 목적지 IP 주소: 패킷을 받을 장치의 IP 주소이다.

(4) TTL(Time To Live): 패킷이 네트워크에 머무를 수 있는 최대 시간을 나타낸다. 이 값은 패킷이 무한히 순환하는 것을 방지하는 데 사용된다.

▶ UDP(User Datagram Protocol): 인터넷 프로토콜 스택의 전송 계층에 속하는 프로토콜이다. UDP는 TCP(Transmission Control Protocol)와 함께 가장 많이 사용되는 전송 계층 프로토콜 중 하나이다.

UDP는 연결이 없는(Connectionless) 프로토콜로, 패킷을 전송하기 전에 목적지와의 연결을 설정하지 않는다. 또한, 패킷의 도착을 확인하거나 재전송하는 메커니즘이 없어, TCP보다 간단하지만 신뢰성이 떨어질 수 있다.

UDP를 사용하는 이유는 다음과 같다.

(1) 속도: 연결 설정이나 패킷 확인 과정이 없기 때문에, TCP보다 데이터 전송이 빠르다.

(2) 오버헤드 감소: 패킷 확인이나 재전송과 관련된 정보가 없으므로, 패킷 헤더의 크기가 작다.

(3) 멀티캐스팅과 브로드캐스팅 지원: UDP는 한 번에 여러 목적지에 패킷을 전송하는 멀티캐스팅과 브로드캐스팅을 지원한다.

▶ IP(Internet Protocol)

: 인터넷 프로토콜 스택인 TCP/IP에서 가장 핵심적인 프로토콜 중 하나로, 네트워크를 통해 데이터 패킷을 전송하는 방법을 정의한다. IP의 주요 역할은 데이터 패킷을 송신지에서 목적지까지 올바르게 전달하는 것이다. 이를 위해 IP는 각각 고유의 주소를 가지며 각 패킷에 대한 메타데이터를 포함하는 IP 헤더를 사용한다. IP 헤더에는 다음과 같은 정보가 포함된다.

(1) 버전: 사용된 IP 프로토콜의 버전을 나타낸다. 현재는 IPv4와 IPv6가 주로 사용된다.

(2) 소스 IP 주소: 패킷을 보낸 장치의 IP 주소이다.

(3) 목적지 IP 주소: 패킷을 받을 장치의 IP 주소이다.

(4) TTL(Time To Live): 패킷이 네트워크에 머무를 수 있는 최대 시간을 나타낸다. 이 값은 패킷이 무한히 순환하는 것을 방지하는 데 사용된다.

▶ TFTP(Trivial File Transfer Protocol)

: 매우 간단한 파일 전송 프로토콜이다. TFTP는 TCP보다 더 간단한 UDP를 사용하여 작동하며, 이는 TFTP가 더 적은 메모리와 컴퓨팅 리소스를 필요로 하지만, 신뢰성이나 데이터 순서 보장 등의 TCP의 기능은 빠져있다는 것을 의미한다.

TFTP의 주요 특징은 다음과 같다.

(1) 단순성: TFTP는 매우 단순한 프로토콜로, 클라이언트가 서버에 연결하고 파일을 요청하거나 전송하는 기본적인 기능만을 제공한다. 이는 TFTP를 임베디드 시스템이나 부트스트래핑(컴퓨터 시스템의 초기화 과정) 등의 상황에서 유용하게 만든다.

(2) UDP 사용: TFTP는 UDP(User Datagram Protocol)를 사용하여 데이터를 전송한다. UDP는 TCP보다 단순하며, 연결 설정 과정이 없고, 데이터의 순서나 신뢰성을 보장하지 않는다. 이는 TFTP가 빠른 속도로 데이터를 전송할 수 있게 하지만, 네트워크의 상태에 따라 데이터가 손실되거나 순서가 뒤바뀔 수 있음을 의미한다.

(3) 블록 단위 전송: TFTP는 데이터를 고정 크기의 블록으로 나누어 전송한다. 각 블록은 번호가 매겨져 있어, 수신자가 블록을 올바른 순서로 재조립할 수 있게 한다.

(4) 오류 검출: TFTP는 각 블록에 대한 확인응답(ACK)을 사용하여 데이터의 정확성을 검사한다. 수신자가 블록을 제대로 받으면 서버에 확인응답을 보내어, 서버가 다음 블록을 보낼 수 있게 한다. 만약 블록이 손실되면, 확인응답이 오지 않아 서버가 같은 블록을 다시 보내게 된다.

4) 클라이언트/서버 프로그램

▶ 클라이언트–서버 모델

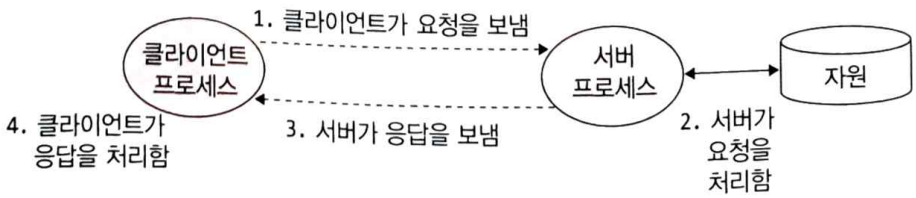


그림 13.1 클라이언트–서버 모델

< 클라이언트–서버 모델 >

: 대부분의 네트워크 응용 프로그램들은 위와 같은 클라이언트-서버 모델을 기반으로 동작한다. 클라이언트-서버 모델은 하나의 서버 프로세스와 여러 개의 클라이언트 프로세스들로 구성된다. 동작 과정은 위와 같다.
소켓은 양방향 통신 방법으로 클라이언트-서버 모델을 기반으로 프로세스 사이의 통신에 매우 적합하다.

8. 소켓 프로그래밍

1) 소켓의 정의

▶ 소켓(socket)은 네트워크에서 데이터를 주고받는 데 사용되는 엔드포인트이다. 네트워크를 통해 두 대의 컴퓨터나 같은 컴퓨터 내의 두 프로세스 간에 데이터를 주고받는 통신 메커니즘이 필요한 경우, 소켓을 이용한다.
소켓은 IP 주소와 포트 번호의 조합으로 구성되며, 이를 통해 네트워크 상의 특정 위치를 지정할 수 있다.

2) TCP 소켓

▶ 스트림 소켓(TCP): 연결 지향적이며, 데이터를 순서대로 안정적으로 전송한다. 데이터의 경계가 없으며, 전송 중에 데이터가 손실되거나 순서가 바뀌지 않는다.
TCP 소켓의 특징은 다음과 같다.
(1) 연결 지향적: 스트림 소켓은 데이터를 전송하기 전에 먼저 목적지와의 연결을 설정한다. 이렇게 하여 데이터가 정확한 목적지로 전송되는 것을 보장한다.
(2) 신뢰성: TCP 기반의 스트림 소켓은 데이터의 도착을 확인하고, 필요한 경우 재전송한다. 이를 통해 데이터가 정확하게 전송되는 것을 보장한다.
(3) 순서 보장: 스트림 소켓은 패킷의 순서를 유지합니다. 이는 데이터가 전송된 순서대로 도착하는 것을 보장한다.
(4) 양방향 통신: 스트림 소켓은 동시에 양방향으로 데이터를 전송할 수 있다.

3) UDP 소켓

▶ 데이터그램 소켓(UDP): 비연결 지향적이며, 데이터를 개별적인 패킷으로 전송한다. 데이터의 경계가 있으며, 전송 중에 데이터가 손실되거나 순서가 바뀔 수 있다.
UDP 소켓의 특징은 다음과 같다.
(1) 연결이 없음: 데이터그램 소켓은 데이터를 전송하기 전에 목적지와의 연결을

설정하지 않는다. 이는 데이터 전송을 시작하는 데 필요한 시간을 줄이고, 네트워크 자원을 절약할 수 있다.

(2) 신뢰성 없음: UDP 기반의 데이터그램 소켓은 데이터의 도착을 확인하지 않고, 재전송을 하지 않는다. 이는 네트워크 트래픽을 줄이지만, 데이터가 손실되거나 순서대로 도착하지 않을 수 있다.

(3) 단방향 또는 양방향 통신: 데이터그램 소켓은 단방향 또는 양방향으로 데이터를 전송할 수 있다. 하지만 양방향 통신을 위해서는 별도의 메커니즘을 구현해야 한다.

4) 소켓 프로그래밍 응용

▶ 소켓 프로그래밍은 네트워크로 연결된 두 컴퓨터 간의 통신을 가능하게 하는 기술이다. 이를 통해 애플리케이션은 네트워크를 통해 데이터를 주고 받을 수 있다. 소켓 프로그래밍은 다양한 응용 분야에서 사용되는데, 대표적인 예시는 다음과 같다.

(1) 웹 서버와 클라이언트: 웹 서버는 클라이언트의 HTTP 요청을 받아 처리하고, 그 결과를 클라이언트에 반환하는 역할을 한다. 이 과정은 소켓을 통해 이루어진다.

(2) 이메일 서버와 클라이언트: 이메일 클라이언트는 사용자의 메일을 이메일 서버에 전송하고, 이메일 서버는 이를 받아 처리하여 수신자에게 전달한다. 이 과정 역시 소켓을 통해 이루어진다.

(3) 인스턴트 메시징: 카카오톡, 페이스북 메신저, 슬랙 등의 인스턴트 메시징 애플리케이션은 사용자 간의 실시간 통신을 가능하게 한다. 이때 소켓은 메시지의 주고받음을 담당한다.

(4) 멀티플레이어 온라인 게임: 온라인 게임에서는 여러 플레이어가 동시에 게임을 즐길 수 있다. 이때 각 플레이어의 상태 정보를 실시간으로 주고 받는 데 소켓이 사용된다.

(5) IoT(Internet of Things): IoT 디바이스는 서버와 통신하여 데이터를 주고받는다. 이때 소켓이 이러한 통신을 가능하게 한다.