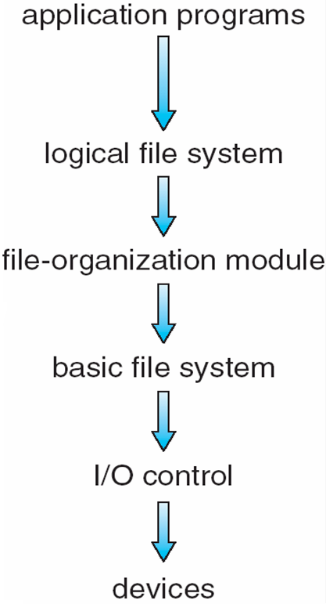
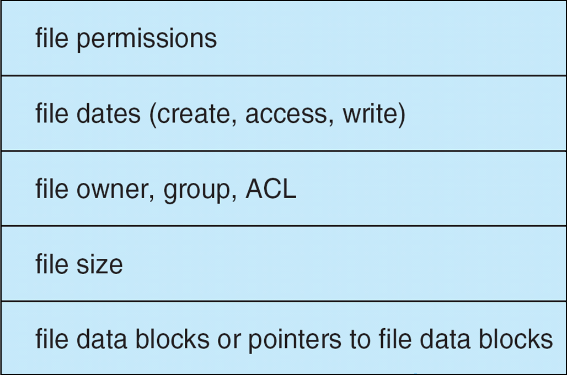
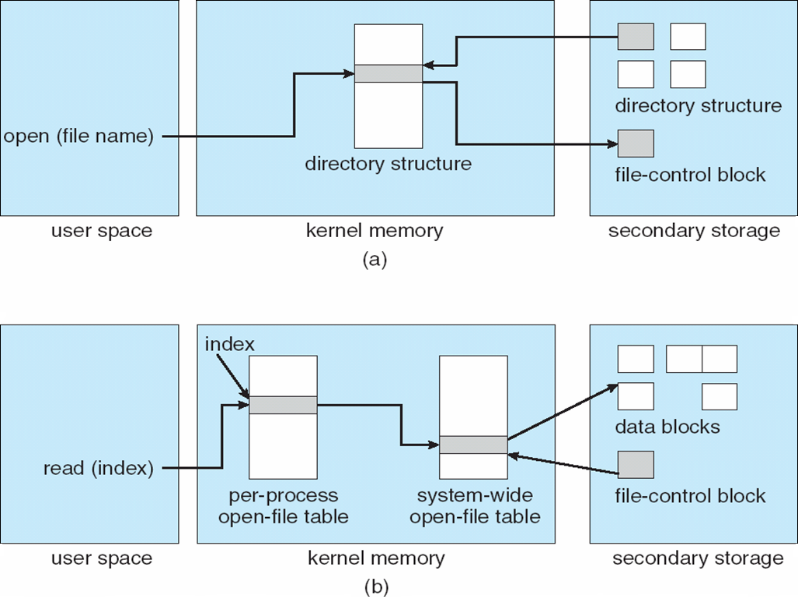
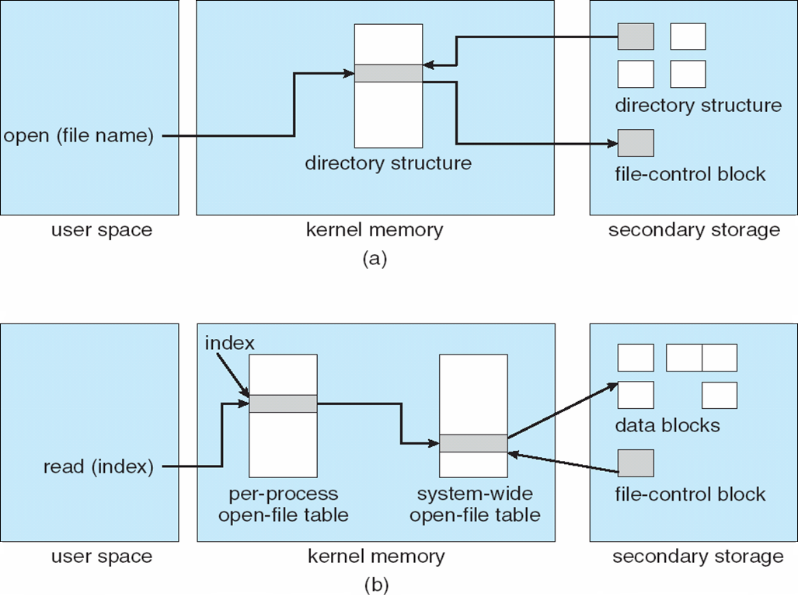
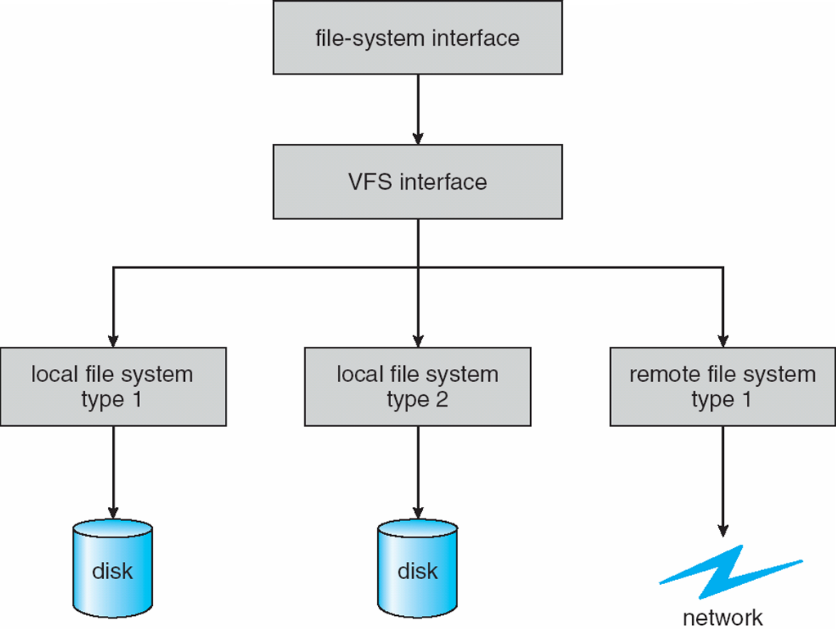
**Chap 11. 파일 시스템 구현**

**11.1 파일 시스템 구조**

* File structure
  + 논리적 저장 장치
  + 정보와 관련된 집합
  + 디스크  
    ⬝ 파일 시스템을 유지하기 위한  
     방대한 보조 저장 장치  
    ⬝ 추가 장소를 사용하지 않고,  
     재 기록 가능  
    ⬝ 디스크에 있는 임의의 블록의  
     정보를 직접 접근 가능
  + 디스크는 내부 재 작성 및  
    random access 제공
  + I/O 전송은 섹터 블록 (512 bytes)에서  
    수행
  + FCB (File Control Block)  
    ⬝ 소유, 허가 파일 내용의 위치를  
     포함하여 파일에 관한 정보를 가짐
  + 장치 드라이버 (Device Driver)  
    ⬝ 기본 파일 시스템은 적절한 장치  
     드라이버에게 디스크 상의 물리  
     블록을 읽고 쓰도록 일반적인 명령을   
     내림
  + 파일 시스템 구현을 위해  
    계층 구조를 사용  
    -> 코드의 중복 최소화  
      
    
* 파일 시스템 설계 문제
  + 파일 시스템이 사용자에게 어떻게  
    보여야 할지 정의
  + 논리 파일 시스템을 물리적인 보조  
    저장 장치로 map하는 알고리즘과   
    자료 구조를 만드는 것
* File System Layers
  + Device Drivers  
    ⬝ 장치 드라이버는 I/O control layer  
     에서 I/O 장치 관리  
    ⬝ commands example :  
    “read drive1, cylinder 72, track 2, sector 10, into memory location 1060”
  + Basic File System  
    ⬝ 적절한 장치 드라이버에게 디스크   
     상의 물리 블록을 읽고 쓰도록  
     일반적인 명령을 내림  
    ⬝ 각 디스크 블록은 숫자로 표시된  
     디스크 주소에 의해 식별 (드라이브  
     1의 실린더 73, 트랙2, 섹터 10)  
    ⬝ commands example :  
     “retrieve block 123”  
    ⬝ 메모리 버퍼, 캐시 (할당, 해제, 교체)  
     관리  
     🢭 Buffers : 전송 중인 데이터 보유  
     🢭 Caches : 자주 사용되는 데이터  
     보유
  + File Organization Module  
    ⬝ 파일의 논리 블록과 물리 블록들  
     양쪽을 알고 있어야 함  
    ⬝ 파일 할당 유형, 파일의 위치를  
     앎으로써 파일에 대한 논리 블록  
     주소를 물리 블록 주소로 변환 가능  
    ⬝ 디스크 공간이 비어 있는지 파악  
     하는 자유 공간 관리자 포함  
    ⬝ 디스크 할당
  + Logical File System  
    ⬝ 메타데이터 정보 관리  
     🢭 메타데이터 : 파일의 내용 자체인  
     자료를 제외한 모든 파일 시스템   
     구조 (데이터를 설명하는 데이터)  
    ⬝ 파일 제어 블록 (inodes in UNIX)을  
     유지 관리하여 파일 이름을 파일  
     번호, 파일 핸들, 위치로 변환  
    ⬝ 디렉토리 구조 관리  
     🢭 파일 이름을 심볼릭하게 줄 경우   
     처리하여 하위 단계인 파일 구성   
     모듈에게 필요한 정보를 넘겨주기   
     위해  
    ⬝ 보호, 보안을 책임

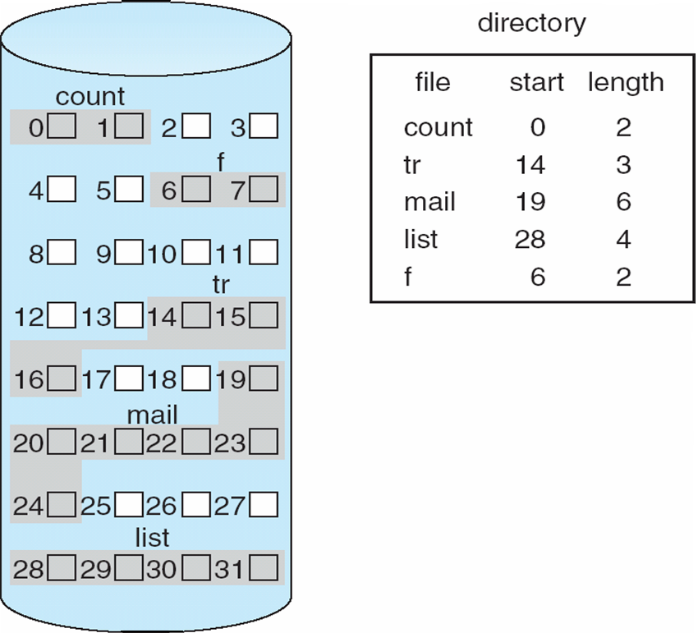
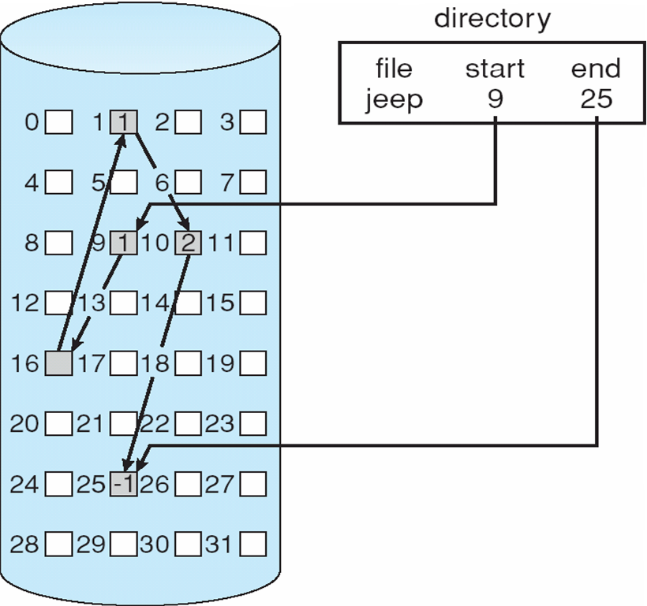
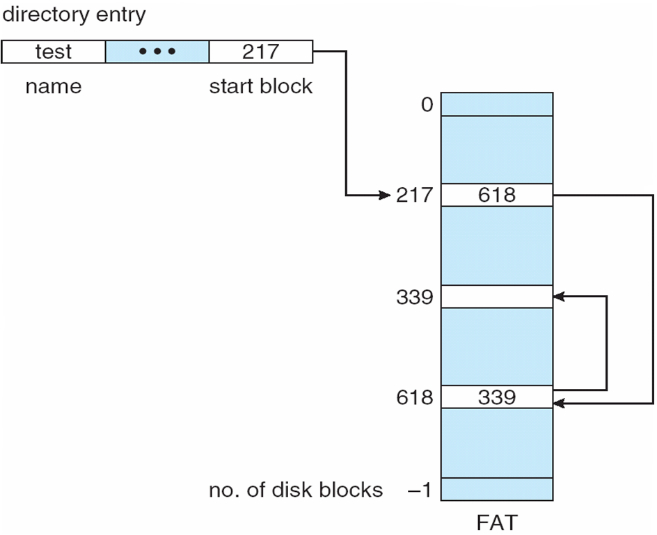
**11.2 파일 시스템 구현**

* File System Implementation
  + 파일 시스템은 디스크에 저장된 운영체제를 어떻게 부트시키는지, 블록의 총 개수, 자유 블록의 수와 위치, 디렉토리 구조, 개별 파일에 대한 정보를 디스크 상에 가지고 있음
  + 메모리 내 정보는 파일 시스템 관리와 캐싱을 통한 성능 향상을 위해 사용   
    ⬝ 이 정보들은 장착 시점에 적재,  
     장착 해제 시에 제거
* 디스크 상 구조 (On-disk Structure)
  + Boot Control Block (부트 제어 블록)  
    ⬝ 볼륨 당 하나씩 존재  
    ⬝ 시스템이 그 파티션으로부터 운영  
     체제를 부트시키는데 필요한 정보를  
     가짐  
    ⬝ 한 파티션의 첫 번째 블록
  + Volume Control Block(볼륨 제어 블록)  
    = super blocks (in UFS)  
    = Master file table (in NTFS)  
    ⬝ 볼륨 당 존재  
    ⬝ 블록의 수, 블록의 크기, 자유 블록   
     수와 포인터, 자유 FCB, 포인터 등   
     파티션 정보 포함
  + 디렉토리 구조  
    ⬝ 파일 조직화 하는데 사용  
    ⬝ UFS  
     🢭 디렉토리 구조에  
     파일 이름, 해당 inode 번호 저장  
    ⬝ NTFS  
     🢭 Master file table에 정보 저장
  + FCB (File Control Block)  
      
    ⬝ 자세한 파일 정보를 가짐  
     (파일 허가, 소유, 크기, 자료 블록의   
     위치 등)  
    ⬝ UFS (Unix File System)  
     🢭 inode라 불림  
     🢭 inode 번호, 허가, 크기, 날짜  
    ⬝ NTFS (Windows File System)  
     🢭 마스터 테이블 안에 저장  
     🢭 파일마다 한 행을 가지고  
     관계 데이터베이스 구조 사용
* 메모리 내 정보 (In-memory Structure)
  + 메모리 내 파티션 테이블  
    : 장착된 파티션 정보 포함
  + 최근 접근된 디렉토리의 디렉토리 정보를 가짐
  + Mount Table  
    : 파일 시스템 마운트, 마운트 포인트,   
     파일 시스템 유형 저장
  + 아래 두 그림은 OS에서 제공하는  
    필요한 파일 시스템 구조
  + File Open
  + File Read
  + 플러스 버퍼  
    : Secondary storage에 데이터 블록   
     보유
  + 파일 기술자 = 파일 핸들  
    ⬝ 오픈 파일 테이블의 항목에 대한  
     포인터  
    ⬝ 파일에 빠르게 접근하기 위함  
    ⬝ 모든 파일 연산은 이 포인터를 통해  
     실행 됨
  + Open()  
    : 이후 사용을 위해 파일 핸들 반환
  + Read()  
    : 읽기에서 얻은 데이터는 사용자  
     프로세스 메모리 주소로 복사 됨
* 파티션과 마운팅 (Partition and Mounting)
  + 디스크의 배치는 OS에 따라 달라짐  
    ⬝ 디스크는 여러 파티션으로 분할  
     될 수 있음  
    ⬝ 하나의 볼륨이 여러 디스크 상에  
     존재하는 여러 파티션을 아우를 수  
     있음
  + Booting Block  
    ⬝ 부트 정보는 별도의 파티션에 저장  
    ⬝ 부트 정보는 자신의 포맷을 가짐  
    ⬝ 부트 이미지는 최소한 특정 OS를  
     부트할 수 잇는 명령어들과 부가적인   
     정보를 포함  
    ⬝ 파일 시스템에서 커널을 로드하는   
     방법을 알 수 있는 충분한 코드가   
     포함된 블록의 부팅 볼륨/boot  
     loader set
  + Root Partition  
    ⬝ 운영체제 커널과 더불어  
     다른 시스템 파일을 포함 할 수 있음  
    ⬝ 부트 시 장착 됨  
     (다른 파티션은 부트 시 자동으로  
     장착되거나 수동으로 나중에 장착 됨)  
    ⬝ OS가 포함되어 있음  
     (다른 파티션에는 다른 OS들, 다른  
     파일 시스템 등이 포함되어 있음)
  + 마운트 될 때,  
    파일 시스템 일관성 체크  
    ⬝ 모든 메타데이터가 맞는가?  
     🢭 If not, 고치고, 다시 시작  
     🢭 If yes, 마운트 테이블에 추가,  
     액세스 허용
  + Mount Table  
    ⬝ 운영체제는 메모리 내부의  
     마운트 테이블에 파일 시스템이  
     장착되었다는 사실과, 파일 시스템의   
     유형 기록
* VFS (Virtual File System:가상 파일 시스템)  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
  + Unix를 포함한 대부분의 운영체제는  
    객체 지향 방법 사용  
    ⬝ 구현 단순화, 조직화, 모듈화를 위함
  + 여러 유형의 파일 시스템에 동일한  
    시스템 호출 인터페이스 (API) 사용 가능 (VFS 인터페이스 명확하게 정의)  
    ⬝ 파일 시스템의 일반적 연산을  
     구현과 분리
  + 전체 네트워크에 걸쳐 파일을 유일하게 표시할 수 있는 기법 제공  
    ⬝ vnode에 기반 (파일 표현 구조)  
    ⬝ vnode : inode나 네트워크 파일 세부   
     정보 저장   
    ⬝ 커널은 각 활성 노드(파일/디렉토리)에 대해 하나의 vnode를 가짐
  + 적절한 파일 시스템 구현 루틴에 대한 작업을 전송
  + API는 VFP 인터페이스에 있음  
    (특정 유형 파일시스템에 x)
  + VFS는 어떤 종류의 객체인지 알 필요 없이, 객체의 함수 테이블의 적절함 함수를 호출하여 원하는 작업 수행
  + VFS는 어떠한 inode를 갖는 객체가 디스크 파일인지, 디렉토리 파일인지, 원격 파일인지에 대해 신경 쓰지 않음
  + VFS는 자료가 읽혀지는 방식에 신경 쓰지 않고, 그 함수를 호출하는 됨
* VFS 구현 (in LINUX)
  + VFS를 이루는 기본 객체 유형  
    ⬝ inode : 각 파일  
    ⬝ file : 오픈 파일  
    ⬝ superblock : 전체 파일 시스템  
    ⬝ dentry : 각 디렉토리 항목
  + 필수적인 연산 집합  
    ⬝ 모든 객체는 대응되는 함수 테이블  
     포인터를 가짐  
    ⬝ 함수 테이블 : 특정 객체의 지정된   
    연산을 구현하는 처리 함수 주소 나열  
    ⬝ For example,  
     🢭 int open(. . .) – 파일 열기  
     🢭 int close(. . .) – 열린 파일 닫기  
     🢭 ssize\_t read(. . .) - 파일 읽기  
     🢭 ssize\_t write(. . .) – 파일 쓰기  
     🢭 int mmap(. . .) – 파일 메모리 매핑

**11.3 디렉토리 구현**

* Liner List (선형 리스트)
  + 자료 블록에 대한 포인터들의 선형 리스트를 디렉토리에 사용
  + 파일 생성 과정  
    : 디렉토리를 탐색하여 같은 이름을   
     가진 파일 존재 확인  
     -> 디렉토리 끝 부분에 첨가
  + 파일 삭제 과정  
    : 디렉토리에서 이름 검색  
     -> 파일에 할당된 공간 방출
  + 장점 : 프로그램이 쉬움
  + 단점 : 파일을 찾기 위해 선형 탐색  
     -> 실행 시간이 긺
  + 정렬된 리스트  
    ⬝ 이진 탐색 가능  
    -> 평균 탐색 시간 줄임  
    ⬝ 파일을 생성, 삭제가 복잡  
    -> 정렬 유지를 위해 많은 정보 이동  
    ⬝ 복잡한 트리(예, B+ tree)가 도움 됨  
    ⬝ 장점 : 별도의 정렬 작업 없이  
     정렬된 디렉토리 리스트 생산 가능
* Hash Table (해시 테이블)
  + 해시 데이터 구조를 갖는 선형 리스트
  + 파일 이름을 제시하면 해시로부터 값을 얻어서, 그것을 포인터로 활용하여 리스트를 직접 접근  
    -> 디렉토리 검색 시간 단축
  + 충돌(collision)에 대한 보완만 한다면,  
    삽입/삭제도 쉽게 수행 가능  
    ⬝ 충돌(collision)  
     : 둘 이상의 파일 이름이  
     같은 위치를 지정하는 경우
  + 문제점  
    ⬝ 해시 테이블이 고정된 크기를 갖음  
    ⬝ 해시 테이블의 크기에 따라  
     해시 기능도 제한을 받음
  + 항목이 고정 크기이거나, 체인 오버플로우 방법을 사용하는 경우에만 적합
  + Chained-overflow hash table  
    (체인 오버플로우 해시테이블)  
    ⬝ 각 해시 항목 – 연결 리스트  
     ( 하나의 값 x)  
    ⬝ 새로운 항목을 연결 리스트에 추가  
     함으로써 추돌 해결  
    ⬝ 이름을 찾을 때, 충돌하는 테이블   
     항들의 연결 리스트를 검색  
     -> 찾기 작업은 늦어지지만,  
     선형 검색보다는 빠름

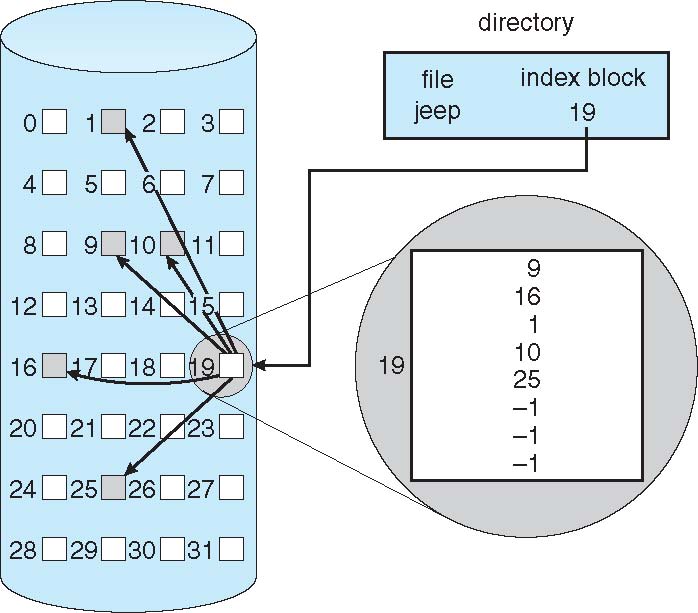
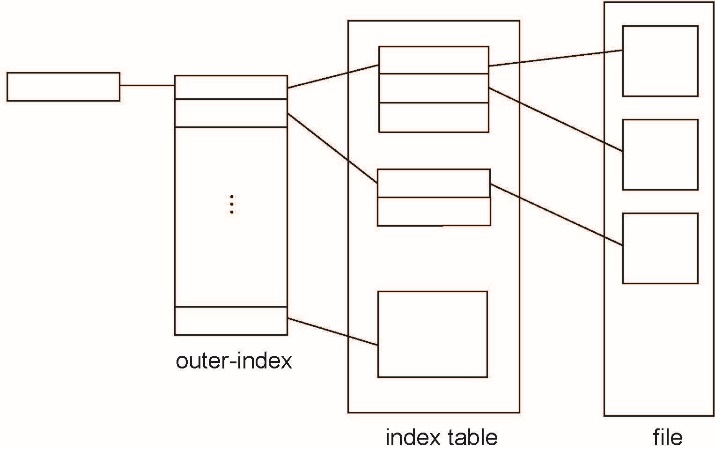
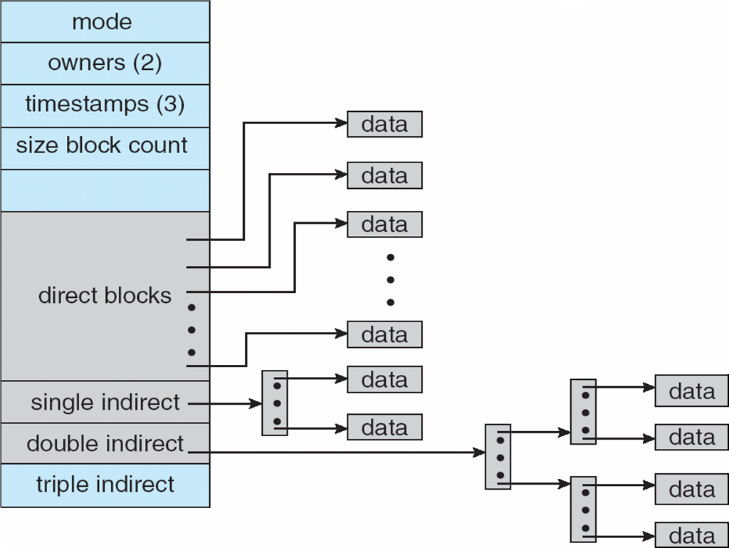
**11.4 할당 방법 (Allocation Method)**

* Allocation Method
  + 파일에 대해 디스크 블록이 할당되는 방법을 나타냄
  + 디스크의 직접 접근 특성이 파일의 구현에 융통성을 허용
  + 할당의 3 가지 방법  
    ⬝ 연속 / 연결 / index
* Contiguous Allocation (연속 할당)  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
  + 각 파일이 디스크 내에서  
    연속적인 공간을 차지하도록 요구  
    (각 파일은 연속 블록 집합 차지)
  + 필요한 디스크 탐색의 횟수 최소화
  + 대부분 최고의 성능
  + 파일 접근 시 시작 위치, 블록 수만  
    필요  
    ⬝ 순차적 접근  
     : 파일 시스템은 가장 최근 참조된   
     주소 기억하고 있다가,  
     필요할 때 다음 블록을 읽음  
    ⬝ 현재로부터 n 번째 블록 접근  
     : 현재 블록 + n을 즉시 직접 접근
  + 순차 접근, 직접 접근 모두 지원
  + 문제점  
    ⬝ 새로운 파일을 위한 가용 공간 찾기  
     🢭 자유 공간을 관리하기 위해  
     선택된 시스템이 이 작업 담당  
     🢭 First-fit / Best-fit  
     (8.3의 동적 공간 할당 문제)  
     -> First-fit이 더 빠름  
    ⬝ 파일 크기 알기  
     🢭 파일이 생성될 때 필요한 공간의   
     크기를 알아야만 할당 가능  
    ⬝ 외부 단편화  
     🢭 파일이 할당되고 반납됨에 따라  
     자유 디스크 공간이 조그만 조각  
     으로 나누어짐  
     🢭 자유 공간이 조각들로 나누어 질  
     때마다 발생  
     🢭 디스크 공간의 전체 크기와 평균   
     파일의 크기에 따라 크고 작은  
     문제가 됨  
    ⬝ compaction(밀집화) off-line, on-line  
     🢭 외부 단편화로 인하여 상당량의  
     디스크 손실을 막기 위해, user는  
     전체 파일 시스템을 다른 플로피  
     디스크 or 테이프로 복사하는  
     재압축(repacking) 루틴 수행  
     🢭 밀집화의 비용 = 시간  
     -> 오래 걸리고, 작업을 주 단위로   
     해야 하므로, 하드디스크에는  
     적합하지 않음  
     🢭 Off-line  
     : 디스크 탈착 상태에서 처리  
     : 밀집화 동안, 시스템 정상 작동 x  
     🢭 On-line  
     : 시스템 정상 가동 중에 밀집화  
     : 시스템 성능 저하 요인
  + Extent-Based (단위 기반) Systems  
    ⬝ 많은 최신 파일 시스템 (i.e., Veritas   
     File System)은 수정된 연속 할당  
     체계 사용  
    ⬝ 어느 정도의 연속된 공간만 초기에   
     할당, 양이 충분히 크지 않을 때는,   
     추후 n개의 연속된 공간을 단위  
     (extent)로 할당  
    ⬝ 단위(extent)  
     : 연속적인 디스크 블록  
    ⬝ 파일은 하나 이상의 단위들로 구성  
    ⬝ 파일 블록들의 위치  
     : 위치, 블록 수, 다음 단위의  
     첫 블록에 대한 포인터로 기록
* Linked Allocation (연결 할당)  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
  + 연속 할당의 모든 문제 해결  
    ⬝ 외부 단편화 x  
    ⬝ 자유 공간 리스트의 어떠한 자유  
     블록들을 이용하여도 무방  
     (모든 블록의 크기가 같기 때문)  
    ⬝ 생성할 파일의 크기가 미리 고정될  
     필요 x  
    ⬝ 파일은 얼마든지 커질 수 있음  
    ⬝ 디스크 공간 주기적 밀집화 x
  + 각 파일은 디스크 블록의 연결 리스트
  + 파일의 디스크 블록들이 디스크 내에 흩어져 있음
  + 디렉토리는 파일의 첫 번째와 마지막 블록에 대한 포인터를 가짐
  + 각 블록이 다음 블록에 대한 포인터를 포함  
    ⬝ 이 포인터는 사람이 사용 불가  
    ex) 한 블록이 512byte, 디스크 주소  
     (=포인터)가 4byte 이면, 사용자  
     에게는 한 블록이 508 byte로 보임
  + 각 디렉토리 항목은 파일의 첫 디스크 블록에 대한 포인터를 가짐
  + 파일은 nill 포인터에서 끝남  
    ⬝ nill : 연결의 끝에 해당하는  
     포인터 값
  + 단점  
    ⬝ 순차적 접근 파일에만 효과적  
     -> 직접 접근 방식에는 비효율적  
     🢭 i번째 블록을 찾으려면, 파일 처음  
     부터 시작해서 i번째 블록에 도달  
     할 때까지 포인터를 따라가야 함  
    ⬝ 포인터를 위한 공간 필요  
    ⬝ 해결 방안 : 클러스터로 구성
  + 클러스터(cluster) 단위로 할당  
    ⬝ 여러 블록들을 하나의 클러스터로   
     구성하여 클러스터 단위로 할당  
    ⬝ 파일의 디스크 공간에서 포인터가  
     차지하는 비율이 작아짐  
    ⬝ 디스크 처리량 향상  
    ⬝ 블록 할당과 자유 리스트 유지에  
     드는 공간을 줄임  
    ⬝ 디스크 접근 시간 향상  
    ⬝ 단점  
     🢭 내부 단편화의 증가  
     : 클러스터에서 낭비되는 공간이   
     한 블록에서 낭비되는 공간보다   
     크기 때문
  + 신뢰성 문제  
    ⬝ 블록들이 디스크에 흩어졌기 때문  
    ⬝ 하나의 포인터를 잃어버리거나,  
     잘못된 포인터 값을 가지면,  
     모든 자료를 잃음
  + 블록을 찾는 데 많은 I/O가 필요하며, 디스크 검색이 필요
  + FAT (File Allocation Table)  
      
      
      
      
      
      
      
      
      
      
      
      
      
    ⬝ 각 파티션 시작 부분이 FAT로 사용  
    ⬝ FAT 테이블은 각 디스크 블록마다  
     한 개의 항복을 가짐/ 이 항목은  
     디스크 블록 번호를 INDEX로 찾음  
    ⬝ 디렉토리 항목은 각 파일의 첫번째   
     블록 번호를 가리킴  
    ⬝ 블록 번호를 가지고, FAT 테이블로  
     가면 그 항복은 다음 블록의 블록   
     번호를 가리킴  
    ⬝ 마지막 블록의 테이블 항은 파일  
     끝은 나타내는 특수한 값을 가짐  
    ⬝ 미사용 블록은 0으로 표시  
    ⬝ 디스크에서 더 빠르고 캐시 될 수  
     있음  
    ⬝ 새로운 블록 할당이 쉬움  
     ➀ 값이 0인 테이블 항을 찾음  
     ➁ 이번 파일의 끝 값을  
     새로운 블록의 주소로 대체  
     ➂ 0을 파일의 끝 값으로 대체

R1 / 512

Q2

R2

* Indexed Allocation (색인 할당)  
    
    
    
    
    
    
    
    
    
    
    
    
  + 모든 포인터를 index block으로 관리
  + Index block  
    : 디스크 블록 주소를 모아 놓은 배열
  + Random access
  + 외부 단편화 없이 직접 접근  
    (디스크 상의 임의의 자유 공간을  
    사용할 수 있기 때문)
  + Index block의 오버헤드 문제
  + Index 위해 한 블록이 할당 되어야 함
  + 블록 액세스  
    ⬝ i번째 블록을 읽기 위해,  
     index block의 i번째 항목에서  
     포인터를 얻어서 그 블록을 읽음
  + Index block의 크기는 작을수록 좋음
  + Index block의 크기가 너무 작다면,  
    큰 파일에는 충분한 포인터 공간x
  + 큰 파일에 대한 해결 방안  
    ⬝ Linked Scheme (연결 기법)  
     🢭 여러 개의 index block 연결  
      
      
      
      
     Q1 = Index table의 블록  
     R1 = 다음 Index block의 포인터  
     (작은 파일의 경우 nill 값)  
      
      
      
      
      
     Q2 = Index table의 블록으로 이동  
     R2 = 파일의 블록으로 이동  
      
    ⬝ Multilevel index (다단계 색인)  
      
      
      
      
      
      
      
      
      
      
      
      
     🢭 첫 번째 수준의 index block은  
     여러 개의 두 번째 수준 index  
     block들에 대한 포인터를 가짐  
     🢭 두 번째 수준의 index block은  
     실제 자료 블록들을 가리킴  
     🢭 파일 크기에 따라 3, 4번째  
     수준으로 계속됨  
     🢭 한 블록 크기가 4,096 바이트라면,  
     1,024개의 4바이트 포인터를  
     한 index block에 저장  
     / 2단계 색인 기법을 쓰면,   
     1,048,576개(1,024^2개) 가능  
      
      
      
      
     Q1 = outer-index 블록으로 이동  
     R1 = 다음 Index table의 포인터  
      
      
      
      
     Q2 = index table의 블록으로 이동  
     R2 = 파일의 블록으로 이동  
      
    ⬝ Combined Scheme (결합 기법)  
      
      
      
      
      
      
      
      
      
      
      
      
      
      
    🢭 디렉토리 내 index block이  
     15개의포인터를 가짐  
     -> 처음 12개 : 직접 블록  
     -> 다음 3개 : 간접 블록  
    🢭 간접 블록을 통하여 간접적으로  
     파일 블록을 찾을 수 있음  
    🢭 파일 포인터가 32비트일 경우 2^32,  
     즉, 4GB 주소 지정 가능  
     -> 32비트 파일 포인터로 처리할 수   
     있는 것 보다 더 많은 index block

LA / (512 x 511)

Q1

R1

LA / (512 x 512)

Q1

R1

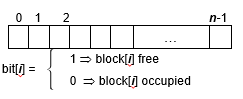
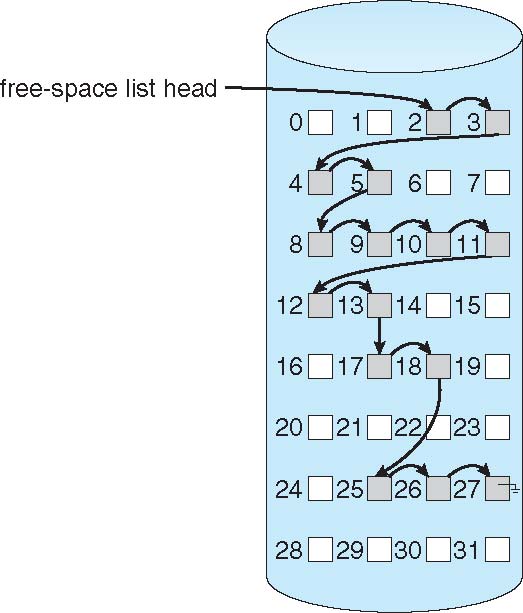
R1 / 512

Q2

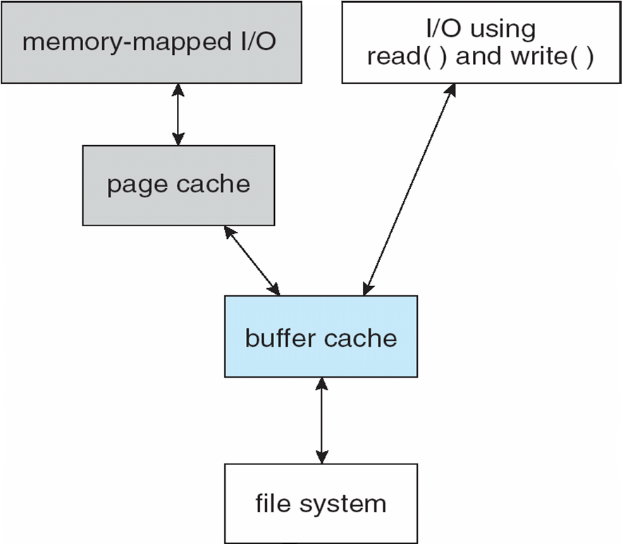
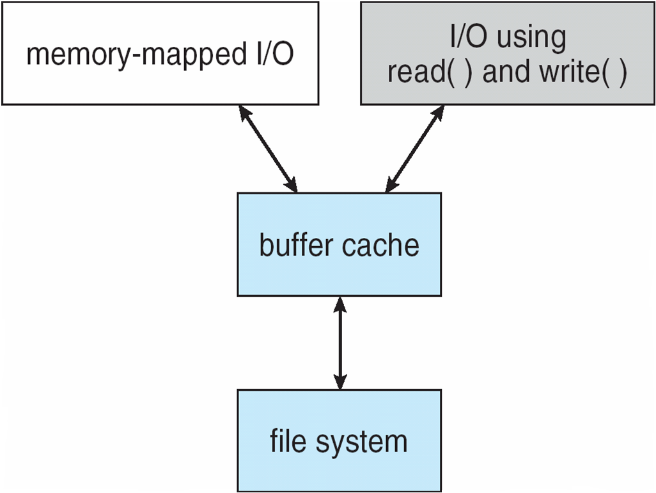
R2

* 성능
  + 최상의 방법은 파일 액세스 유형에  
    따라 다름  
    ⬝ Contiguous Allocation (연속 접근)  
     🢭 순차적, 직접적인 접근  
    ⬝ Linked Allocation (연결 접근)  
     🢭 순차적  
    ⬝ Indexed Allocation (색인 접근)  
     🢭 단일 블록 액세스  
     : index block 읽은 후,  
     자료 블록 읽기  
     🢭 Clustering  
     : 처리량 개선,  
     CPU 오버헤드 줄일 수 있음
  + 디스크 I/O 하나를 저장하기 위한 지침을 실행 경로에 추가하는 것이 좋음

11.5 자유 공간의 관리

* 파일 시스템은 자유 공간을 리스트로 유지하고 관리
  + 디스크의 공간은 제한되어 있으므로, 삭제된 파일들이 차지하던 공간을  
    새로운 파일을 위해 재사용 해야함
  + 리스트에 모든 자유 블록 등록
  + 새로운 파일 생성  
    : 자유 공간 리스트 탐색을 통해  
     새로운 파일을 위한 공간 할당  
     -> 할당된 공간은  
     자유 공간 리스트로부터 삭제
  + 파일 삭제  
    : 파일이 쓰던 공간은  
     자유 공간 리스트에 추가
* Bit Vector
  + 각 블록은 1비트로 표현
  + 블록 번호 계산  
    (워드의 비트 수)\*(값이 0인 워드의 수)  
    +첫 번째 1비트의 오프셋
  + 연속된 자유 블록 찾는 일이 효율적
  + Example:  
    block size = 4KB = 2^12 bytes  
    disk size = 2^40 bytes (1TB)  
    n = 2^40/2^12 = 2\*18 bits (32MB)  
    if cluster of 4blocks->8MB of memory
* Linked List (연결 리스트)  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
  + n번째 자유 블록은 다음 자유 블록을 가리키는 포인터를 가짐
  + 연속 공간을 쉽게 얻을 수 없음  
    (매 번 디스크를 접근해야 하므로  
     비효율적)
  + 공간 낭비는 없음
  + 전체 목록을 순회할 필요는 없음
* Grouping
  + 첫 번째 자유 블록 내에 n 개의 블록 주소 저장  
    ⬝ 처음 n-1개 실제로 비어 있는  
     블록의 주소  
    ⬝ 마지막 1개는 n-1개의 빈 블록 주소  
     를 가지고 있는 자유 블록을 가리킴
  + 다수 개의 자유 블록 주소들을 쉽게 찾을 수 있음
* Counting (계수)
  + 연속된 자유 블록의 첫 번째 블록의 주소와 연속된 블록의 count만 유지
  + 자유 공간 리스트의 각 항은 하나의 디스크 주소와 블록의 개수로 구성
  + 연속 할당 알고리즘, 클러스터링을 통해 공간 할당할 경우 유용
* Space Maps
  + ZFS (SOLARIS file system)에서 사용
  + 매우 큰 파일 시스템에서 메타 데이터 I/O를 고려  
    ⬝ 비트 맵과 같은 전체 데이터 구조는   
     메모리에 적합하지 않음  
     (수 천개의 I/O)
  + 장치 공간은 metaslab 단위로 나누고,  
    metaslab를 관리  
    ⬝ 주어진 볼륨에는 수 백개의   
     metaslab가 포함될 수 있음
  + 각 metaslab에는 연관된 space map이 있음  
    ⬝ 계산 알고리즘 사용
  + 로그 파일에 대한 레코드이다.  
    ⬝ 모든 블록 활동의 로그, 시간 순서,   
     카운팅 형식
  + Metaslab activity – space map을 균형 트리 구조의 메모리로 로드하고, 오프셋으로 인덱싱  
    ⬝ 로그를 해당 구조로 재생  
    ⬝ 연속된 자유 블록을 단일 항목으로   
     결합

**10.6 효율과 성능**

* Efficiency (효율성)에 영향을 미치는 것
  + 디스크 할당, 디렉토리 알고리즘  
    ⬝ inode를 미리 할당하고, 파티션 전체  
     에 분산 할당함으로써 성능 향상  
    ⬝ inode들을 가급적 자료 블록 부근에  
     위치하도록 함으로써 성능 향상  
    ⬝ 클러스터링 기법 : 내부 단편화를   
     허용하는 대신 파일 탐색, 전송  
     성능 향상
  + 파일의 디렉토리 항목 내에 저장되어야 할 정보의 종류
  + Metadata 구조의  
    사전 할당/필요에 따른 할당
  + 고정 크기/다양한 크기의 데이터 구조
* Performance (성능)에 영향을 미치는 것
  + Data와 metadata를 가까이 유지
  + Buffer cache  
    : 자주 사용되는 블록에 대한  
     main memory의 별도 섹션
  + OS의 필요나 apps의 요청에 의한  
    동기식(Synchronous) 쓰기  
    ⬝ NO Buffering/Caching  
     : 디스크 드라이브에 도착한 후  
     실행을 계속할 수 있음  
    ⬝ 비동기식 쓰기  
     : 더 일반적, 버퍼 가능, 더 빠름
  + Free-behind(지연 자유화) and  
    read-ahead(미리 읽기)  
    : 순차적 접근을 최적화하는 기술
  + Read는 Write보다 느림
* Page Cache  
    
    
    
    
    
    
    
    
    
    
    
    
    
  + 가상 메모리 기법을 사용하여  
    파일 자료를 페이지로 캐시
  + Memory-mapped I/O는  
    Page Cache를 사용
  + 파일 시스템을 통한 Routine I/O는  
    Buffer Cache 사용
* 통합 버퍼 캐시 (Unified buffer cache)  
    
    
    
    
    
    
    
    
    
    
    
  + 다수의 시스템이 페이지 캐싱을 사용하여 프로세스 페이지와 파일 자료 모두 캐시
  + 이중 캐싱 방지  
    ⬝ 이중 캐싱  
    : 페이지 캐시, 버퍼 캐시 두 번 캐싱  
    -> 메모리, CPU, I/O 사이클 낭비

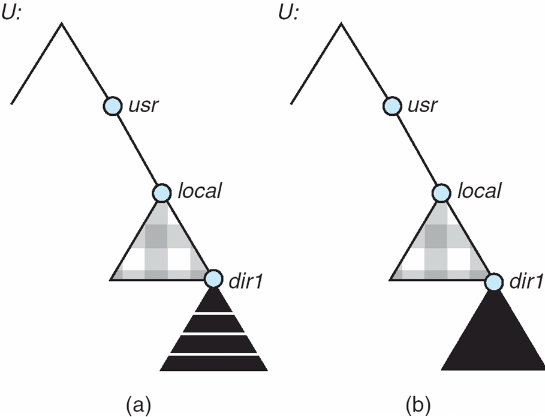
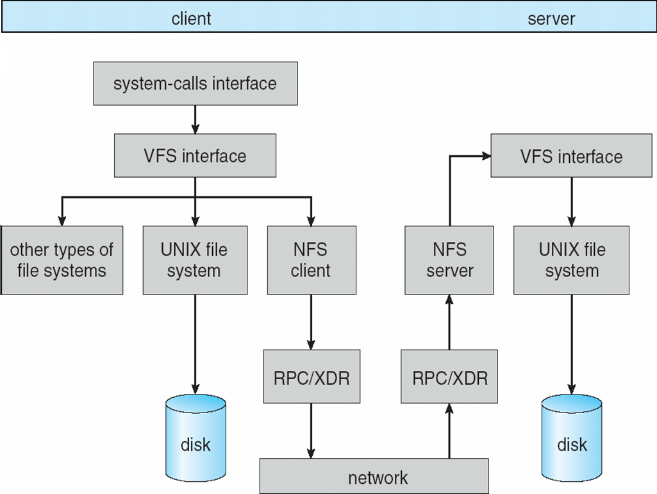
**11.7 복구 (Recovery)**

* Consistency Checking (일관성 검사)
  + 디렉토리 구조에 있는 자료와 디스크에 있는 자료를 비교하고 불일치가 발견되면 그것을 복구하려 시도
* Back up (백업)
  + 시스템 프로그램을 이용하여 디스크의 내용을 다른 저장 장치에 백업
* Restore (복원)
  + 파일 또는 전체 디스크가 훼손되더라도 백업으로부터 쉽게 복원 가능

**11.8 로그 구조 파일 시스템**

* Log-Structured File System
  + 각 메타데이터 변경을 파일 시스템에 트랜잭션으로 기록
  + 모든 트랜잭션은 로그에 기록  
    ⬝ 트랜잭션(변경)은 로그에 기록되면   
     그들은 commit된 것으로 간주되고,  
     시스템 호출은 사용자 프로세스로  
     복귀하여 실행을 계속하도록 허용  
    ⬝ 개별 장치, 디스크의 섹션에 대한   
     경우도 있음  
    ⬝ 파일 시스템이 아직 변경되지 않을   
     수 있음
  + 로그안의 트랜잭션은 파일 시스템  
    구조에 비동기적으로 기록 됨  
    ⬝ 파일 시스템 구조가 수정되면,  
     트랜잭션은 로그에서 제거됨
  + 파일 시스템이 고장(crash)나면,  
    로그에 남아있는 트랜잭션들은 반드시 실행을 완료해야 함  
    ⬝ 로그 파일에는 0개 이상의  
     트랜잭션이 있을 것  
    ⬝ 이 트랜잭션들은 commit되었다해도,   
     파일 시스템에서 실행이 완료되지  
     않은 것이기 때문에 완료되어야 함
  + 고장(crash)으로부터의 빠른 복구는 메타데이터의 불일치 가능성을 제거함

**11.9 NFS**

* NFS (Network File System)  
    
  
  + LAN(또는 WAN)을 거쳐 원격 파일을 접근하기 위한 소프트웨어 시스템의   
    구현과 명세
  + 구현 : Sun 워크스테이션과 다른 하드웨어에서 실행 되는 것으로, Solaris 운영체제의 일부
  + TCP 또는 UDP/IP 중 하나 사용
  + 서로 연결된 워크스테이션의 집합을 독립적인 파일 시스템을 가진 독립적인 기계들의 집합으로 간주
  + 목적 : 파일 시스템들 사이에서 일정 수준의 공유를 투명하게 허용하는 것
  + 시스템의 독립성을 위해서,  
    원격 파일 시스템의 공유는  
    다른 시스템에는 영향을 미치지 않고,  
    단지 클라이언트에게만 영향을 줌
  + 원격 디렉토리가 로컬 파일 시스템의 디렉토리 상에 장착되게 함  
    ⬝ 마운트 된 디렉토리는 로컬 파일  
     시스템의 서브 트리처럼 보이며,  
     로컬 디렉토리의 기존 서브 트리를   
     대체  
    ⬝ 로컬 디렉토리는 새로 마운트 된  
     디렉토리의 루트의 이름이 됨
  + 마운트 명령을 위한 매개변수로 원격 디렉토리를 지정하는 일은 투명하지 않은 방식으로 실행되며,  
    원격 디렉토리의 위치(호스트의 이름)이 반드시 제공되어야 함  
    ⬝ 일단 마운트 되면, 원격 디렉토리의  
     파일을 투명하게 접근 가능
  + 접근 권한이 허용되면,  
    모든 파일 시스템(또는 파일 시스템 내의 디렉토리)을 로컬 디렉토리에  
    원격으로 마운트 가능
  + NFS는 서로 다른 기계들, 운영체제, 네트워크 구조로 구성된 다른 환경에서 작동하는 것
  + 독립성 : NFS 명세는 이들 매체에 독립적이기 때문에 다른 구현들을 이용 가능하게 함
  + 독립성은 두 개의 구현 독립적 인터페이스 간에서 사용되는 XDC(External Data Representation) 프로토콜 위에 구축된 RPC 프리미티브를 통해서  
    이루어 짐
  + NFS 명세는 마운트 기법에 의해 제공되는 서비스와 실제 원격 파일 접근 서비스를 구분함
* NFS Mount Protocol
  + 서버와 클라이언트 사이의 초기 논리적 연결을 생성하기 위해 사용됨
  + 마운트 명령은 장착될 원격 디렉토리 이름, 그것을 저장하는 서버 기계의 이름을 포함
  + 마운트 요구는 적절한 RPC로 마운트 되고, 지정된 서버 기계 상에서 실행되는 마운트 서버로 전달 됨
  + 서버는 수출 리스트(Export list)를 유지하고, 이 리스트는 그들을 마운트 할 수 있도록 하는 기계의 이름, 마운트 가능한 로컬 파일 시스템을 지정
  + 서버가 자신의 수출 리스트가 허용하는 마운트 요구를 받았을 때, 서버는 클라이언트에게 마운트 된 파일 시스템의 파일 핸들을 반환  
    ⬝ 파일 핸들  
     : 파일 시스템 식별자와 수출된 파일   
     시스템 내의 마운트 된 디렉토리를  
     가리키는 inode 번호로 구성
  + 마운트 작업은 서버 측에는 영향을 미치지 않음
* NFS Protocol
  + 원격 파일 연산을 위한 원격 프로시저 호출의 집합 제공  
    ⬝ 디렉토리 내의 파일 검색  
    ⬝ 디렉토리 항목 집합의 읽기  
    ⬝ 링크와 디렉토리들의 조작  
    ⬝ 파일 속성의 접근  
    ⬝ 파일 읽기와 쓰기
  + NFS 서비스의 주요 특징은 무상태성  
    ⬝ 각각의 요구들은 매개변수들의  
     완전한 집합을 제공해야함
  + 결과들이 클라이언트에게 반환되기 전에 변경된 자료가 서버의 디스크에   
    기록되어야 함
  + NFS 프로토콜은 병행성 제어 기법  
    (concurrency-control mechanism)을 제공하지 않음
* NFS 구조의 3계층
  + UNIX file-system interface  
    ⬝ open, read, write, close 호출과 파일   
     설명자를 기반으로 함
  + VFS (Virtual File System) layer  
    ⬝ 로컬 파일을 원격 파일과 구별  
    ⬝ 로컬 파일은 파일 시스템 유형에  
     따라 구별  
    ⬝ 원격 요청에 대한 NFS 프로토콜  
     프로시저 호출
  + NFS service layer  
    ⬝ 구조의 바닥  
    ⬝ NFS 프로토콜 구현
* Path-Name Translation (경로 이름 반환)
  + 파일 경로 이름을 파싱하는 것을 포함
  + 세 개의 디렉토리 항목으로 나뉘어짐  
    ➀ usr ➁ local ➂ dir
  + 경로를 구성 요소 이름으로 분리하고, 구성 요소 이름과 디렉토리 vnode의 모든 쌍에 대한 별도의 NFS lookup 호출을 실행함으로써 이루어짐
  + Lookup을 더 빠르게 하기 위해서,  
    클라이언트 측의 디렉토리 이름 룩업 캐시는 원격 디렉토리 이름들에 대한 vnode를 갖고 있음
* Remote Operations (원격 연산)
  + 파일 연산을 위한 정규 UNIX 시스템 호출들과 NFS의 RPC 프로토콜들 간에는 거의 1:1 대응 관계가 있음  
    (파일 열기/닫기 제외)
  + NFS는 원격 서비스의 패러다임을 따르지만, 실제로는 성능 향상을 위해 버퍼링과 캐싱 기술 사용
  + 파일 블록 캐시  
    ⬝ 파일을 열 때, 커널은 캐시 된  
     속성들을 다시 가져올지 또는  
     이전 것을 다시 사용할지를 결정  
     하기 위해 원격 서버를 검사  
    ⬝ 캐시 된 파일 블록들은 대응되는  
     캐시 된 속성들이 최신 버전일  
     경우에만 사용함
  + 파일 속성(inode 정보) 캐시  
    ⬝ 서버로부터 새로운 속성들이 도착될   
     때마다 캐시 된 속성이 갱신됨  
    ⬝ 클라이언트는 서버가 자료를 디스크  
     에 기록한 것을 확일하기 전까지는  
     지연 쓰기 블록을 자유화하지 않음