2주차 - 데이터베이스, 운영체제

CHAPTER 3. 운영체제

- 운영체제 (OS, operating system)
  - 사용자가 컴퓨터를 쉽게 다루게 해주는 인터페이스
  - 펌웨어 ? 운영체제와 유사하나 소프트웨어를 추가적으로 설치할 수 없는 것

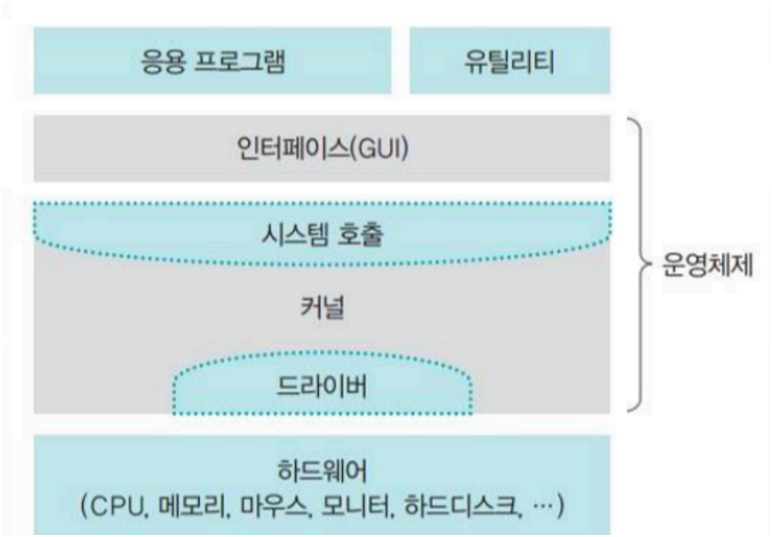
3.1 운영체제와 컴퓨터

3.1.1 운영체제의 역할과 구조

운영체제의 역할

1. CPU 스케줄링과 프로세스 관리
2. 메모리 관리
3. 디스크 파일 관리
4. I/O 디바이스 관리

운영체제의 구조



💡 **GUI**

- 사용자가 전자장치와 상호작용할 수 있도록 하는 사용자 인터페이스의 한 형태
- 단순 명령어 창이 아닌 아이콘을 마우스로 클릭하는 단순한 동작으로 컴퓨터와 상호작용

드라이버

- 하드웨어를 제어하기 위한 소프트웨어

CUI

- 그래픽이 아닌 명령어로 처리하는 인터페이스

시스템콜

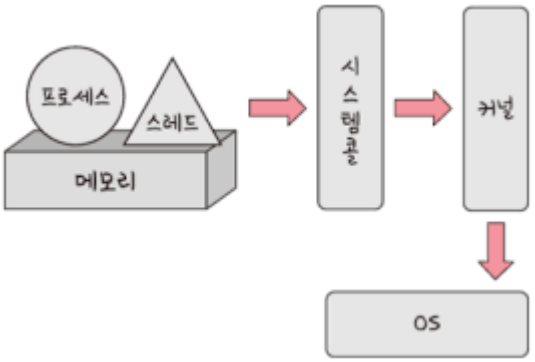
- 운영체제가 커널에 접근하기 위한 인터페이스
- 유저 프로그램이 운영체제의 서비스를 받기 위해 커널 함수를 호출할 때 씀

- 유저 프로그램이 I/O 요청으로 트랩 발동 → 올바른 I/O 요청인지 확인 → 유저 모드가 시스템 콜을 통해 커널 모드로 변환되어 실행
  - 이 과정을 통해 컴퓨터 자원에 대한 직접 접근을 차단
  - 프로그램을 다른 프로그램으로부터 보호

💡 **I/O 요청**

- 입출력 함수, 데이터베이스, 네트워크, 파일 접근 등에 관한 일

- 프로세스나 스레드에서 운영체제로 어떤 요청을 할 때 시스템 콜과 커널을 거쳐 운영체제에 전달됨



- 시스템 콜은 하나의 추상화 계층

  장점

  - 이를 통해 네트워크 통신이나 데이터베이스와 같은 낮은 단계의 영역 처리에 대한 부분을 많이 신경 쓰지 않고 프로그램을 구현할 수 있음

modebit

- 시스템 콜 작동시 modebit을 참고하여 유저모드와 커널모드를 구분
- modebit : 1 또는 0의 값을 가지는 플래그 변수
- 카메라, 키보드 등 I/O 디바이스는 운영체제를 통해서만 작동해야 함
  - 유저 모드를 기반으로 카메라가 켜지면 공격자가 나쁜 짓을 할 수 있음
  - 커널 모드를 거쳐 운영체제를 통해 작동해도 100% 막을 수는 없음
  - 그러나 운영체제를 통해 작동해야 막기 쉽다
  - 이를 위한 장치가 바로 modebit임
  - 0은 커널모드 1은 유저모드
- 예 : 카메라 이용시 → 시스템 콜 호출 → modebit 1에서 0으로 전환 → 커널모드 변경 → 카메라 자원을 이용한 로직 수행 → modebit 0에서 1로 전환 → 유저모드 변경 → 로직 수행

💡 **유저모드**

- 유저가 접근할 수 있는 영역을 제한적으로 두며 컴퓨터 자원에 함부로 침범하지 못하는 모드

커널모드

- 모든 컴퓨터 자원에 접근할 수 있는 모드

커널

- 운영 체제의 핵심 부분이자 시스템콜 인터페이스를 제공
- 보안, 메모리, 프로세스, 파일 시스템, I/O 디바이스, I/O 요청 관리 등 운영체제의 중추적 역할

3.1.2 컴퓨터의 요소

- CPU, DMA 컨트롤러, 메모리, 타이머, 디바이스 컨트롤러

CPU(Central Processing Unit)

- 산술논리 연산장치, 제어장치, 레지스터로 구성되어 있는 컴퓨터 장치
- 인터럽트에 의해 단순히 메모리에 존재하는 명령어를 해석해서 실행
- 관리자 역할을 하는 커널이 프로그램을 메모리에 올려 프로세스로 만들면 일꾼인 CPU가 이를 처리

제어장치(CU, Control Unit)

- 프로세스 조작을 지시하는 CPU의 한 부품
  - 입출력장치 간 통신을 제어
  - 명령어를 읽고 해석
  - 데이터 처리를 위한 순서를 결정

레지스터

- CPU 안에 있는 매우 빠른 임시기억장치
- CPU와 직접 연결되어 있음
  - 따라서 연산 속도가 메모리보다 수십 배 ~ 수백 배 빠름
- CPU는 자체적으로 데이터를 저장할 수 없음

  → 레지스터를 거쳐 데이터를 전달

산술논리연산장치(ALU, Arithmetic Logic Unit)

- 산술연산(덧셈, 뺄셈..)과 논리 연산(배타적 논리합, 논리곱)을 계산하는 디지털 회로

CPU에서 연산 처리

1. 제어장치가 메모리에 계산할 값을 로드하면서 레지스터에도 로드
2. 제어장치가 레지스터에 있는 값을 계산하라고 산술논리연산장치에 명령
3. 제어장치가 '레지스터에서 메모리로' 계산한 값을 다시 저장

인터럽트

- 어떤 신호가 들어왔을 때 CPU를 잠시 정지시킴
  - 키보드, 마우스 등 IO 디바이스로 인한 인터럽트
  - 0으로 숫자를 나누는 산술 연산에서의 인터럽트
  - 프로세스 오류 등으로 발생
- 인터럽트 발생시
  - 인터럽트 핸들러 함수가 모여있는 인터럽트 벡터로 가서 인터럽트 핸들러 함수 실행
  - 인터럽트는 우선순위에 따라 실행
  - 하드웨어 인터럽트/소프트웨어 인터럽트로 나뉨

💡 **인터럽트 핸들러 함수**

- 인터럽트가 발생했을 때 이를 핸들링하기 위한 함수
- 커널 내부의 IRQ를 통해 호출
- request_irq()를 통해 인터럽트 핸들러 함수를 등록할 수 있음

하드웨어 인터럽트

- 키보드 연결이나 마우스 연결 등 IO 디바이스에서 발생하는 인터럽트
  - 인터럽트 라인 설계 후 순차적인 인터럽트 실행 중지 → 운영체제에 시스템 콜 오청, 원하는 디바이스로 향해 디바이스에 있는 작은 로컬 버퍼에 접근하여 일을 수행

💡 **하드웨어 인터럽트란?**
컴퓨터가 일을 하고 있는데, 키보드나 마우스 같은 외부 장치가 "잠깐! 내가 할 일이 있어!"라고 알려주는 신호예요.

과정을 일상 예시로 설명하면:

1. **인터럽트 발생**:
    - 당신이 책을 읽고 있는데 갑자기 전화벨이 울림
    - 키보드를 누르거나 마우스를 클릭하는 것과 같음
2. **현재 작업 중단**:
    - 책 읽기를 멈추고 어디까지 읽었는지 책갈피로 표시
    - CPU도 현재 하던 일을 멈추고 상태를 저장
3. **운영체제에 알림**:
    - "전화가 왔어요!"라고 누군가 알려줌
    - 하드웨어가 운영체제에게 "키보드 입력이 있어요!" 신호를 보냄
4. **해당 장치 처리**:
    - 전화를 받으러 감
    - 운영체제가 키보드나 마우스의 작은 저장공간(버퍼)에서 데이터를 가져와서 처리
5. **원래 작업 복귀**:
    - 전화를 끝내고 책갈피가 있던 곳부터 다시 읽기 시작
    - CPU도 멈췄던 작업을 이어서 계속함

왜 이런 방식을 쓸까요?

- CPU가 계속 "혹시 키보드 눌렀나? 마우스 움직였나?" 확인할 필요 없이
- 정말 필요할 때만 알려주니까 효율적이에요!

소프트웨어 인터럽트

- 트랩(trap)이라고도 함
- 프로세스 오류 등으로 프로세스가 시스템 콜을 호출할 때 발동

💡 **소프트웨어 인터럽트(트랩)란?**
프로그램이 실행되다가 "도움이 필요해요!" 하고 운영체제에게 도움을 요청하거나, 뭔가 잘못되었을 때 발생하는 신호예요.

일상 예시로 설명하면:

1. 시스템 콜 상황 (도움 요청)

- 당신이 요리하다가 "냉장고 문 좀 열어줘!"라고 부모님께 부탁
- 프로그램이 "파일을 읽고 싶어요!", "화면에 글자 출력해주세요!" 하고 운영체제에게 부탁

2. 프로세스 오류 상황 (문제 발생)

- 요리하다가 실수로 뜨거운 팬을 만져서 "아야!" 하고 소리지름
- 프로그램이 잘못된 메모리에 접근하거나, 0으로 나누기를 시도할 때 "에러!" 신호 발생

처리 과정:

1. **문제 발생/도움 요청**: 프로그램에서 시스템 콜 호출 또는 에러 발생
2. **현재 작업 중단**: CPU가 프로그램 실행을 멈춤
3. **운영체제 개입**: "무슨 일이야?" 하고 운영체제가 나서서 상황 파악
4. **문제 해결**:
    - 시스템 콜이면 → 요청한 작업 수행 (파일 읽기, 화면 출력 등)
    - 에러면 → 프로그램 종료하거나 에러 처리
5. **결과 전달**: 작업 완료되면 프로그램에게 결과 알려줌

하드웨어 인터럽트와 차이점:

- **하드웨어**: 외부에서 "똑똑!" (키보드, 마우스가 신호)
- **소프트웨어**: 내부에서 "도와주세요!" 또는 "문제 생겼어요!" (프로그램이 신호)

DMA 컨트롤러 (Direct Memory Access)

- I/O 디바이스가 메모리에 직접 접근할 수 있도록 하는 하드웨어 장치
- CPU 부하를 막아주며 CPU의 일을 부담하는 보조일꾼

    - CPU에만 너무 많은 인터럽트 요청이 들어오니까..
- 하나의 작업을 CPU와 DMA 컨트롤러가 동시에 하는 것을 방지 (동시에 메모리 접근하면 충돌)
    - Cycle Stealing : DMA가 CPU가 안 쓰는 틈틈이 메모리 접근
    - Burst Mode : DMA가 한 번에 쭉 처리하고 CPU에게 메모리 사용권 넘김

장점

1. CPU 부담 줄임
    a. CPU가 데이터 이동 같은 단순 작업에서 해방
    b. 더 중요한 연산에 집중 가능
2. 성능 향상
    a. 인터럽트 횟수 대폭 감소
    b. 전체 시스템 효율성 증가
3. 병렬 처리
    a. CPU는 계산, DMA는 데이터 이동을 동시에

메모리

- 전자회로에서 데이터나 상태, 명령어 등을 기록하는 장치
- 보통 RAM을 일컬어 메모리라고 함 (Random Access Memory)
- CPU는 계산을 담당, 메모리는 기억을 담당
    - 예 : 공장 비유
        - CPU는 일꾼, 메모리는 작업장

타이머

- 특정 프로그램에 시간 제한을 다는 역할
- 시간이 많이 걸리는 프로그램이 작동할 때 제한을 걸기 위해 존재

디바이스 컨트롤러

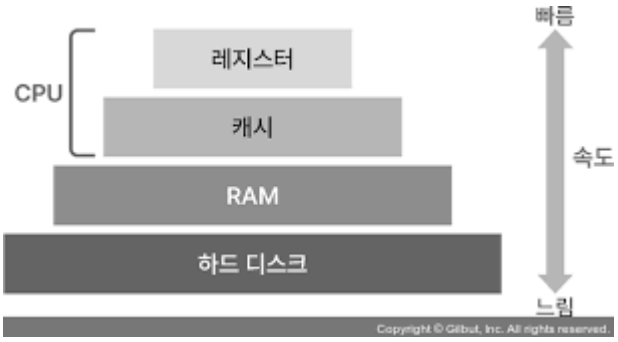
- 컴퓨터와 연결되어 있는 IO 디바이스들의 작은 CPU를 말함
- 옆에 붙어 있는 로컬 버퍼 : 각 디바이스에 데이터를 임시로 저장하기 위한 작은 메모리

3.2 메모리

- CPU는 그저 '메모리'에 올라와 있는 프로그램의 명령어를 실행할 뿐

3.2.1 메모리 계층

- 메모리 계층은 레지스터, 캐시, 메모리, 저장장치로 구성
  - 레지스터 : CPU 안에 있는 작은 메모리, 휘발성, 속도 가장 빠름, 기억 용량 가장 적음
  - 캐시 : L1, L2 캐시 지칭, 휘발성, 속도 빠름, 기억 용량 적음, L3 캐시도 있긴 함
  - 주기억장치 : RAM, 휘발성, 속도 보통, 기억 용량 보통
  - 보조기억장치 : HDD, SSD, 비휘발성, 속도 낮음, 기억 용량 많음



- 계층이 존재하는 이유 - 경제성과 캐시 때문

캐시(cache)

- 데이터를 미리 복사해놓는 임시 저장소
- 빠른 장치와 느린 장치에서 속도 차이에 따른 병목 현상을 줄이기 위한 메모리

→ 데이터를 접근하는 시간이 오래 걸리는 경우를 해결, 무언가를 다시 계산하는 시간을 절약

- 실제로 메모리와 CPU 사이의 속도 차이가 매우 큼
  - 중간에 레지스터 계층을 두어 속도 차이 해결

- 이처럼 속도 차이를 해결하기 위해 계층과 계층 사이에 있는 계층을 캐싱 계층이라고 함
- 예 : 캐시 메모리와 보조기억장치 사이에 있는 주기억장치를 보조기억장치의 캐싱 계층이라고 할 수 있음

지역성의 원리

- 캐싱 계층을 두는 것 말고 캐시를 직접 설정해야 할때는?
  - 자주 사용하는 데이터를 기반으로 설정해야 함
  - 이에 대한 근거가 바로 지역성
  - 시간 지역성과 공간 지역성으로 나뉨

시간지역성(temporal locality)

- 최근 사용한 데이터에 다시 접근하려는 특성
- 예 : for문 안에 있는 i

공간지역성(spatial locality)

- 최근 접근한 데이터를 이루고 있는 공간이나 그 가까운 공간에 접근하는 특성
- 예 : 배열의 요소에 연속적으로 접근

캐시히트와 캐시미스

- 캐시히트 : 캐시에서 원하는 데이터를 찾았을 때
  - 해당 데이터를 제어장치를 거쳐 가져오게 됨
  - 위치도 가깝고 CPU 내부 버스를 기반으로 작동하므로 빠름
- 캐시미스 : 해당 데이터가 캐시에 없다면 주메모리로 가서 데이터를 찾아오는 것
  - 데이터를 메모리에서 가져옴
  - 시스템 버스를 기반으로 작동하므로 느림

캐시매핑

- 캐시가 히트되기 전에 매핑하는 방법
- CPU의 레지스터와 주메모리(RAM) 간에 데이터를 주고받을 때를 기반으로 설명

- 레지스터는 작고 주메모리는 크기 때문에 작은 레지스터가 캐싱 계층으로서 역할을 잘 하려면 매핑을 어떻게 하느냐가 중요함

이름	설명
직접 매핑 (directed mapping)	- 메모리가 1~100이고 캐시가 1~10일 때 1:1~10, 2:1~20 이런 식으로 매핑 어쩌구.. - 처리는 빠르나 충돌 발생이 잦음
연관 매핑 (associated mapping)	- 순서를 일치시키지 않고 관련 있는 캐시와 메모리를 매핑 - 충돌이 적으나 모든 블록을 탐색해야 하므로 느림
집합 연관 매핑 (set associated mapping)	- 직접 매핑 + 연관 매핑 - 순서는 일치시키나 집합을 두어 저장 - 블록화되어 검색이 좀 더 효율적 - 메모리 1~100, 캐시 1~10 일 경우 캐시 1~5에는 1~50의 데이터를 무작위로 저장

웹 브라우저의 캐시

- 소프트웨어적인 대표적인 캐시 : 웹브라우저의 작은 저장소 쿠키, 로컬 스토리지, 세션 스토리지
- 보통 사용자의 커스텀한 정보나 인증 모듈 관련 사항들을 웹브라우저에 저장
    - 추후 서버에 요청할 때 자신을 나타내는 아이덴티티나 중복 요청 방지를 위해 쓰임
    - 오리진에 종속

쿠키

- 만료기한이 있는 키-값 저장소
- same site 옵션을 strict로 설정하지 않았을 경우
    - 다른 도메인에서 요청했을 때 자동 전송
    - 4KB까지 데이터를 저장
    - 만료기한을 정할 수 있음
- 쿠키 설정시
    - httponly옵션 설정 → document.cookie로 쿠키를 볼 수 없음
    - 서버에서 보통 만료기한을 정함

로컬 스토리지

- 만료기한이 없는 키-값 저장소
- 5MB까지 저장 가능
- 웹 브라우저를 닫아도 유지
- HTML5를 지원하지 않는 웹 브라우저에서는 사용 불가
- 클라이언트에서만 수정 가능

세션 스토리지

- 만료기한이 없는 키-값 저장소
- 탭 단위로 세션 스토리지 생성
- 탭 닫을 때 해당 데이터 삭제
- 5MB까지 저장 가능
- HTML5를 지원하지 않는 웹 브라우저에서 사용 불가
- 클라이언트에서만 수정 가능

데이터베이스의 캐싱 계층

- 데이터베이스 시스템 구축시 메인 데이터베이스 위에 레디스(redis) 데이터베이스 계층을 '캐싱 계층'으로 두어 성능을 향상시키기도 함

3.2.2 메모리 관리

- 컴퓨터 내 한정된 메모리를 극한으로 사용해야 함

가상 메모리(virtual memory)

- 메모리 관리 기법 중 하나
- 컴퓨터가 실제로 이용 가능한 메모리 자원을 추상화

  → 이를 사용하는 사용자에게 매우 큰 메모리로 보이게 만듦
- 이 때 가상으로 주어진 주소 : 가상 주소(logical address)
- 실제 메모리상 있는 주소 : 실제 주소(physical address)
- 가상 주소는 메모리관리장치(MMU)에 의해 실제 주소로 변환

→ 사용자는 실제 주소를 의식할 필요 없이 프로그램을 구축

- 가상 메모리
  - 가상 주소와 실제 주소가 매핑되어 있음
  - 프로세스의 주소가 들어있는 '페이지 테이블'로 관리
  - 이 때 속도 향상을 위해 TLB 사용

💡 **TLB**

- 메모리와 CPU 사이에 있는 주소 변환을 위한 캐시
- 페이지 테이블에 있는 리스트를 보관
- CPU가 페이지 테이블까지 가지 않도록 해 속도를 향상시킬 수 있는 캐싱 계층

스와핑

- 가상 메모리에는 존재하지만 실제 메모리인 RAM에는 현재 없는 데이터나 코드에 접근할 경우 페이지 폴트 발생
- 이 때 메모리에서는 당장 사용하지 않는 영역을 하드디스크로 옮기고 하드디스크의 일부분을 마치 메모리처럼 불러와 쓰는 것을 스와핑이라고 함
- 이를 통해 마치 페이지 폴트가 일어나지 않은 것처럼 만듦

페이지 폴트(page fault)

- 프로세스의 주소 공간에는 존재하나 지금 이 컴퓨터의 RAM에는 없는 데이터에 접근했을 때 발생
- 페이지 폴트와 스와핑의 과정
  1. 어떤 명령어가 유효한 가상 주소에 접근, 해당 페이지가 없다면 트랩 발생하여 운영체제에 알림
  2. 운영체제는 실제 디스크에서 사용하지 않는 프레임을 찾음

3. 해당 프레임을 실제 메모리에 가져와서 페이지교체알고리즘을 기반으로 특정 페이지와 교체

   (이 때 스와핑이 일어남)

4. 페이지 테이블 갱신 후 해당 명령어를 다시 시작

💡 **페이지**

- 가상 메모리를 사용하는 최소 크기 단위

프레임

- 실제 메모리를 사용하는 최소 크기 단위

스레싱(thrashing)

- 메모리의 페이지 폴트율이 높은 것
- 컴퓨터의 심각한 성능 저하 초래
- 스래싱은 메모리에 너무 많은 프로세스가 동시에 올라가면 스와핑이 많이 일어나서 발생
- 페이지 폴트가 일어나면 CPU 이용률이 낮아짐
- CPU 이용률이 낮아지면 운영체제가 “CPU가 한가한가?” 하고 가용성을 높이기 위해 더 많은 프로세스를 메모리에 올림
- 이런 악순환이 반복되어 스레싱이 일어남

- 해결법
  - 메모리를 늘림
  - HDD를 사용한다면 SSD로 바꿈
  - 작업세트
  - PFF

작업 세트(working set)

- 프로세스의 과거 사용 이력인 지역성을 통해 결정된 페이지 집합을 만들어서 미리 메모리에 로드하는 것
- 미리 메모리에 로드하면?
  - 탐색에 드는 비용 줄임
  - 스와핑도 줄임

PFF(Page Fault Frequency)

- 페이지 폴트 빈도를 조절
- 상한선과 하한선 만듦
- 상한선 도달시 : 프레임 늘림
- 하한선 도달시 : 프레임 줄임

메모리 할당

- 메모리에 프로그램 할당시 시작 메모리 위치, 메모리 할당 크기를 기반으로 할당
- 연속 할당/불연속 할당

연속 할당

- 메모리에 '연속적으로' 공간 할당
- 고정 분할 방식 : 메모리를 미리 나누어 관리
- 가변 분할 방식 : 매 시점 프로그램의 크기에 맞게 메모리를 분할하여 사용

고정 분할 방식(fixed partition allocation)

- 메모리를 미리 나누어 관리
- 융통성 없음
- 내부 단편화 발생

가변 분할 방식(variable partition allocation)

- 매 시점 프로그램의 크기에 맞게 메모리를 분할하여 사용
- 내부 단편화 발생 X, 외부 단편화 발생 O
- 최초적합, 최적적합, 최악적합이 있음

이름	설명
최초적합 (first fit)	위쪽이나 아래쪽부터 시작해서 홀을 찾으면 바로 할당
최적적합 (best fit)	프로세스의 크기 이상인 공간 중 가장 작은 홀부터 할당
최악적합 (worst fit)	프로세스의 크기와 가장 많이 차이가 나는 홀에 할당

💡 **내부 단편화(internal fragmentation)**

- 메모리를 나눈 크기보다 프로그램이 작아서 들어가지 못하는 공간이 많이 발생하는 현상

외부 단편화(external fragmentation)

- 메모리를 나눈 크기보다 프로그램이 커서 들어가지 못하는 공간이 많이 발생하는 현상

홀(hole)

- 할당할 수 있는 비어 있는 메모리 공간

불연속 할당

- 현대 운영체제가 쓰는 방법
- 페이징 기법
  - 메모리를 동일한 크기의 페이지(보통 4KB)로 나눔
  - 프로그램마다 페이지 테이블을 두어 이를 통해 메모리에 프로그램을 할당

- 세그멘테이션, 페이지드 세그멘테이션이 있음

페이징(paging)

- 동일한 크기의 페이지로 나누어 메모리의 서로 다른 위치에 프로세스 할당
- 홀의 크기가 균일하지 않은 문제가 없어짐
- 그러나 주소 변환이 복잡

세그멘테이션(segmentation)

- 페이지 단위가 아닌 의미 단위인 세그먼트로 나눔
- 프로세스의 메모리는 코드 영역, 데이터 영역, 스택 영역, 힙 영역으로 이루어짐
- 코드와 데이터로 나누거나 코드 내 작은 함수를 세그먼트로 놓고 나누기도 함
- 장점 : 공유나 보안
- 단점 : 홀 크기가 균일하지 않음

페이지드 세그멘테이션(paged segmentation)

- 의미단위인 세그먼트로 나눠 공유나 보안 측면에 강점
- 임의의 길이가 아닌 동일한 크기의 페이지 단위로 나눔

페이지 교체 알고리즘

- 메모리는 한정되어 있어 스와핑이 많이 일어남
- 스와핑이 많이 일어나지 않도록 설계해야 함
- 페이지 교체 알고리즘으로 스와핑이 일어남

오프라인 알고리즘

- 먼 미래에 참조되는 페이지와 현재 할당하는 페이지를 바꾸는 알고리즘
- 가장 좋은 방법
- 그러나 우리는 미래에 사용되는 프로세스를 알 수 없음

- 따라서 사용할 수 없는 알고리즘이지만 가장 좋은 알고리즘이기 때문에 다른 알고리즘과의 성능 비교에 대한 상한 기준을 제공

FIFO(First In First Out)

- 가장 먼저 온 페이지를 교체 영역에 가장 먼저 놓는 방법

LRU(Least Recently Used)

- 참조가 가장 오래된 페이지를 바꿈
- '오래된' 것 파악을 위해 각 페이지마다 계수기, 스택을 두어야 하는 문제점
- LRU 구현을 프로그램으로 할 때는 보통 두 개의 자료 구조로 구현
  - 해시 테이블과 이중 연결 리스트
  - 해시 테이블 : 이중 연결 리스트에서 빠르게 찾을 수 있도록 사용
  - 이중 연결 리스트 : 한정된 메모리를 나타냄

NUR(Not Used Recently)

- LRU에서 발전
- 일명 clock 알고리즘
  - 먼저 0과 1을 가진 비트를 둠
  - 1 : 최근에 참조
  - 0 : 참조되지 않음
  - 시계 방향으로 돌면서 0을 찾고 0을 찾은 순간 해당 프로세스를 교체, 해당 부분을 1로 바꿈

LFU(Least Frequency Used)

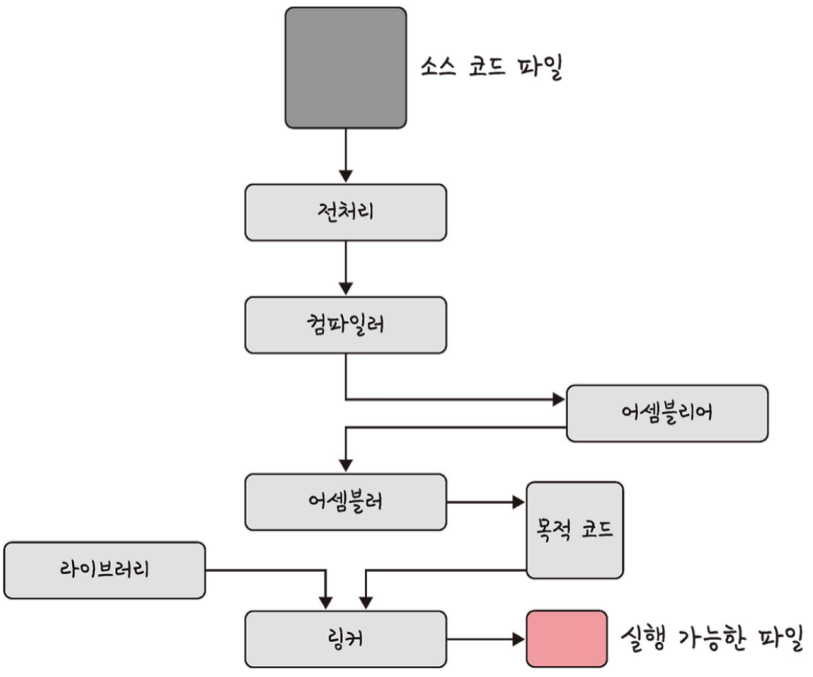
- 가장 참조 횟수가 적은 페이지를 교체
- 많이 사용되지 않은 것을 교체한다는 것

3.3 프로세스와 스레드

- 프로세스 : 컴퓨터에서 실행되고 있는 프로그램
  - CPU 스케줄링의 대상이 되는 '작업'과 같은 의미로 쓰임
- 스레드 : 프로세스 내 작업의 흐름
- 프로그램이 메모리에 올라가면 프로세스가 되는 인스턴스화가 일어남
- 이후 운영체제의 CPU 스케줄러에 따라 CPU가 프로세스를 실행

3.3.1 프로세스와 컴파일 과정

- 프로세스 : 프로그램이 메모리에 올라가 인스턴스화된 것
- 예 : 프로그램 - chrome.exe 같은 실행 파일
  - 이를 두 번 클릭 → 구글 크롬 프로세스르 변환
- 프로그램의 컴파일 과정(C언어 기반 프로그램으로 설명)



전처리

- 소스 코드의 주석을 제거하고 헤더 파일을 병합하여 매크로를 치환

컴파일러

- 오류 처리, 코드 최적화 작업을 하며 어셈블리어로 변환

어셈블러

- 어셈블리어는 목적 코드로 변환
- 이 때 확장자는 운영체제마다 다름
    - 예 : 리눅스 - .o
    - 가영.c 파일 → 가영.o 파일로 만들어짐

링커

- 프로그램 내에 있는 라이브러리 함수 또는 다른 파일들과 목적 코드를 결합하여 실행 파일 만듦
- 실행파일의 확장자 : .exe 또는 .out

정적 라이브러리와 동적 라이브러리

- 정적 라이브러리
    - 프로그램 빌드 시 라이브러리가 제공하는 모든 코드를 실행 파일에 넣는 방식으로 라이브러리 사용
    - 장점 : 시스템 환경 등 외부 의존도 낮음
    - 단점 : 코드 중복 등 메모리 효율성 떨어짐
- 동적 라이브러리
    - 프로그램 실행 시 필요할 때만 DLL이라는 함수 정보를 통해 참조하여 라이브러리 사용
    - 장점 : 메모리 효율성
    - 단점 : 외부 의존도 높음

3.3.2 프로세스의 상태

생성 상태(created)

- 프로그램이 생성된 상태
- fork()나 exe()를 통해 생성

- 이 때 PCB 할당

fork()

- 부모 프로세스의 주소 공간을 그대로 복사, 새로운 자식 프로세스 생성
- 주소 공간만 복사함
- 부모 프로세스의 비동기 작업 등을 상속하지 않음

exec()

- 새롭게 프로세스 생성

대기 상태(ready)

- 메모리 공간이 충분하면 메모리를 할당받고 아니면 아닌 상태로 대기
- CPU 스케줄러로부터 CPU 소유권이 넘어오기를 기다리는 상태

대기 중단 상태(ready suspended)

- 메모리 부족으로 일시 중단된 상태

실행 상태(running)

- CPU 소유권과 메모리를 할당받고 인스트럭션을 수행 중인 상태
- CPU burst가 일어났다고도 함

중단 상태(blocked)

- 어떤 이벤트가 발생한 이후 기다리며 프로세스가 차단된 상태
- I/O 디바이스에 의한 인터럽트로 이런 현상이 많이 발생하기도 함
- 예 : 프린트 인쇄 버튼을 눌렀을 때 프로세스가 잠깐 멈춘 듯할 때

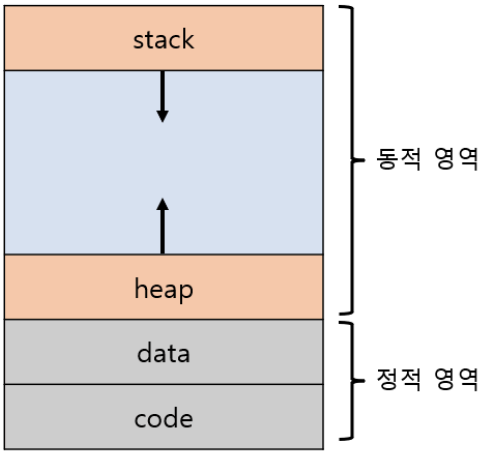
일시 중단 상태(blocked suspended)

- 대기 중단과 유사
- 중단된 상태에서 프로세스가 실행되려고 했으나 메모리 부족으로 일시 중단된 상태

종료 상태(terminated)

- 메모리와 CPU 소유권을 모두 놓고 가는 상태
- 종료 : 자연적으로 종료되거나 / 부모 프로세스가 자식 프로세스를 강제시키는 비자발적 종료
    - 자식 프로세스에 할당된 자원의 한계치를 넘어서거나
    - 부모 프로세스가 종료되거나
    - 사용자가 process.kill 등 여러 명령어로 프로세스를 종료할 때 발생

3.3.3 프로세스의 메모리 구조



- 스택은 위 주소에서부터 할당
- 힙은 아래 주소부터 할당

스택과 힙

- 스택과 힙은 동적 할당
- 동적 할당 : 런타임 단계에서 메모리를 할당받는 것
- 스택 : 지역 변수, 매개변수, 실행되는 함수에 의해 늘어나거나 줄어드는 메모리 영역

    - 함수가 호출될 때마다 호출될 때의 환경 등 특정 정보가 스택에 계속해서 저장됨
    - 또한 재귀 함수가 호출된다고 했을 때 새로운 스택 프레임이 매번 사용되기 때문에 함수 내의 변수 집합이 해당 변수의 다른 인스턴수 변수를 방해하지 않음
- 힙 : 동적으로 할당되는 변수를 담음
    - malloc(), free() 함수를 통해 관리
    - 동적으로 관리되는 자료 구조의 경우 힙 영역 사용
    - 예 : 벡터는 내부적으로 힙영역 사용

데이터 영역과 코드 영역

- 정적 할당되는 영역
- 정적 할당 : 컴파일 단계에서 메모리를 할당하는 것
- 데이터 영역은 BSS segment와 Data segment, code/text segment로 나눠서 저장
    - BSS segment : 전역 변수, static, const로 선언되어 있고 0으로 초기화 또는 초기화가 어떤 값도 되지 않은 변수들이 할당
    - Data segment : 전역 변수, static, const로 선언되어 있고 0이 아닌 값으로 초기화된 변수들이 할당
    - code segment : 프로그램의 코드

3.3.4 PCB(Process Control Block)

- 운영체제에서 프로세스에 대한 메타데이터를 저장한 '데이터'
- 프로세스 제어 블록이라고도 함
- 프로세스가 생성되면 운영체제는 해당 PCB를 생성

- 프로그램 실행 → 프로세스 생성 → 프로세스 주소 값들에 메모리 할당 → 프로세스의 메타데이터들이 PCB에 저장되어 관리됨
- 프로세스의 중요한 정보를 가지고 있어서 일반 사용자들이 접근하지 못하도록 커널 스택의 가장 앞부분에서 관리됨

💡 **메타데이터**

- 데이터에 관한 구조화된 데이터
- 데이터를 설명하는 작은 데이터
- 대량의 정보 가운데에서 찾고 있는 정보를 효율적으로 찾아내서 이용하기 위해 일정한 규칙에 따라 콘텐츠에 대해 부여되는 데이터

PCB의 구조

- 프로세스 스케줄링 상태 : '준비', '일시중단' 등 프로세스가 CPU에 대한 소유권을 얻은 이후의 상태
- 프로세스 ID : 프로세스 ID, 해당 프로세스의 자식 프로세스 ID
- 프로세스 권한 : 컴퓨터 자원 또는 I/O 디바에스에 대한 권한 정보
- 프로그램 카운터 : 프로세스에서 실행해야 할 다음 명령어의 주소에 대한 포인터
- CPU 레지스터 : 프로세스를 실행하기 위해 저장해야 할 레지스터에 관한 정보
- CPU 스케줄링 정보 : CPU 스케줄러에 의해 중단된 시간 등에 대한 정보
- 계정 정보 : 프로세스 실행에 사용된 CPU 사용량, 실행한 유저의 정보
- I/O 상태 정보 : 프로세스에 할당된 I/O 디바이스 목록

컨텍스트 스위칭(context switching)

- PCB를 기반으로 프로세스의 상태를 저장하고 로드시키는 과정
- 한 프로세스에 할당된 시간이 끝나거나 인터럽트에 의해 발생
- 많은 프로세스가 동시에 구동되는 것 같아보이는 건 컨텍스트 스위칭이 아주 빠른 속도로 실행되기 때문
- 싱글 코어 기준 컨텍스트 스위칭
  1. 프로세스 A 실행 중 멈춤
  2. 프로세스 A PCB 저장, 프로세스 B 로드 후 실행
  3. 프로세스 B PCB 저장, 프로세스 A 로드
- 컨텍스트 스위칭이 일어날 때 유휴 시간이 발생
- 컨텍스트 스위칭 시 캐시미스 발생

비용 : 캐시미스

- 컨텍스트 스위칭이 일어날 때 프로세스가 가지고 있는 메모리 주소가 그대로 있으면 잘못된 주소 변환이 생기므로 캐시클리어 과정을 겪게 되고 이로 인해 캐시미스 발생

스레드에서의 컨텍스트 스위칭

- 스레드에서도 컨텍스트 스위칭 일어남
- 스레드는 스택 영역을 제외한 모든 메모리를 공유하므로 스레드 컨텍스트 스위칭의 경우 비용이 더 적고 시간도 더 적게 걸림

3.3.5 멀티 프로세싱

- 멀티 프로세싱 : 여러 개의 프로세스를 통해 동시에 두 가지 이상의 일을 수행할 수 있는 것
  - 하나 이상의 일을 병렬로 처리
  - 프로세스 중 일부에 문제가 생겨도 다른 프로세스에서 처리할 수 있으므로 신뢰성이 높음
  - 하드웨어 관점에서 여러 개의 프로세서로 작업을 처리하는 것을 의미

웹 브라우저

- 웹 브라우저는 멀티프로세스 구조를 가짐
- 브라우저 프로세스 : 주소 표시줄, 북마크 막대, 뒤로 가기 버튼, 앞으로 가기 버튼 등을 담당, 네트워크 요청이나 파일 접근 같은 권한을 담당
- 렌더러 프로세스 : 웹 사이트가 '보이는' 부분의 모든 것을 제어
- 플러그인 프로세스 : 웹 사이트에서 사용하는 플러그인을 제어
- GPU 프로세스 : GPU를 이용해서 화면을 그리는 부분을 제어

IPC(Inter Process Communication)

- 멀티프로세스는 IPC가 가능
- IPC : 프로세스끼리 데이터를 주고받고 공유 데이터를 관리하는 매커니즘
- 예 : 클라이언트와 서버

    - 클라이언트는 데이터를 요청
    - 서버는 클라이언트 요청에 응답
- IPC의 종류 : 공유 메모리, 파일, 소켓, 익명 파이프, 명명 파이프, 메시지 큐
    - 모두 메모리가 완전히 공유되는 스레드보다는 속도가 떨어짐

공유 메모리(shared memory)

- 여러 프로세스에 동일한 메모리 블록에 대한 접근 권한이 부여되어 프로세스가 서로 통신할 수 있도록 공유 메모리를 생성해서 통신하는 것
- 기본적으로 각 프로세스의 메모리를 다른 프로세스가 접근할 수 없지만 공유 메모리를 통해 여러 프로세스가 하나의 메모리를 공유할 수 있음
- IPC 방식 중 어떠한 매개체를 통해 데이터를 주고받는 것이 아닌 메모리 자체를 공유
    - 불필요한 데이터 복사의 오버헤드가 발생하지 않아 가장 빠름
    - 같은 메모리 영역을 여러 프로세스가 공유하므로 동기화 필요
- 하드웨어 관점에서 공유 메모리는 CPU가 접근할 수 있는 큰 랜덤 접근 메모리인 RAM을 가리키기도 함

파일

- 디스크에 저장된 데이터 또는 파일 서버에서 제공한 데이터
- 이를 기반으로 프로세스 간 통신을 함

소켓

- 동일한 컴퓨터의 다른 프로세스나 네트워크의 다른 컴퓨터로 네트워크 인터페이스를 통해 전송하는 데이터를 의미
- TCP, UDP가 있음

익명 파이프(unnamed pipe)

- 파이프 : 프로세스 간 FIFO 방식으로 읽히는 임시공간
- 파이프를 기반으로 데이터를 주고받으며 단방향 방식의 읽기 전용, 쓰기 전용 파이프를 만들어 작동

명명된 파이프(named pipe)

- 파이프 서버와 하나 이상의 파이프 클라이언트 간의 통신을 위한 명명된 단방향 또는 양방향 파이프
- 클라이언트/서버 통신을 위한 별도의 파이프 제공
- 여러 파이프를 동시에 사용할 수 있음
- 컴퓨터의 프로세스끼리 또는 다른 네트워크상의 컴퓨터와도 통신할 수 있음
- 보통 서버용 파이프와 클라이언트용 파이프로 구분해서 작동
- 하나의 인스턴스를 열거나 여러 개의 인스턴스를 기반으로 통신

메시지 큐

- 메시지를 큐 데이터 구조 형태로 관리하는 것
- 커널의 전역변수 형태 등 커널에서 전역적으로 관리
- 장점
  - 다른 IPC 방식에 비해 사용 방법이 매우 직관적이고 간단하며
  - 다른 코드의 수정 없이 단지 몇 줄의 코드를 추가시켜 간단하게 메시지 큐에 접근할 수 있음
- 공유 메모리를 통해 IPC를 구현할 때 쓰기 및 읽기 빈도가 높으면 동기화 때문에 기능 구현이 매우 복잡해짐

  → 대안으로 메시지 큐 사용

3.3.5 스레드와 멀티 스레딩

스레드

- 프로세스의 실행 가능한 가장 작은 단위
- 프로세스는 여러 스레드를 가질 수 있음
- 프로세스와 달리 코드, 데이터, 힙은 스레드끼리 서로 공유
- 그 외의 영역은 각각 생성

멀티스레딩

- 멀티 스레딩 : 프로세스 내 작업을 여러 개의 스레드로 처리하는 기법
- 스레드끼리 서로 자원을 공유하므로 효율성이 높음
- 예 : 웹 요청 처리시
    - 새 프로세스 생성 대신 스레드를 사용하는 웹서버의 경우 리소스 소비가 적음
    - 한 스레드가 중단되어도 다른 스레드는 실행 상태일 때 중단되지 않은 빠른 처리 가능
- 장점 : 리소스 소비 적음, 동시성
- 단점 : 한 스레드에 문제 발생 → 다른 스레드에도 영향 ⇒ 프로세스에 영향

동시성

- 서로 독립적인 작업들을 작은 단위로 나누고 동시에 실행되는 것처럼 보여주는 것

3.3.7 공유 자원과 임계 영역

공유자원(shared resource)

- 시스템 안에서 각 프로세스, 스레드가 함께 접근할 수 있는 모니터, 프린터, 메모리, 파일 데이터 등의 자원이나 변수 의미
- 경쟁 상태 : 공유자원을 두 개 이상의 프로세스가 동시에 읽거나 쓰는 상황
    - 동시에 접근을 시도할 때 접근의 타이밍이나 순서 등이 결괏값에 영향을 줄 수 있는 상태

임계 영역(critical section)

- 둘 이상의 프로세스, 스레드가 공유 자원에 접근할 때 순서 등의 이유로 결과가 달라지는 코드 영역
- 임계 영역 해결 방법 : 뮤텍스, 세마포어, 모니터
    - 이 방법 모두 상호 배제, 한정 대기, 융통성 조건 만족

- 이 방법의 토대가 되는 매커니즘 : 잠금
  - 화장실

> 💡 **상호 배제(mutual exclusion)**
>
> - 한 프로세스가 임계 영역에 들어갔을 때 다른 프로세스는 들어갈 수 없다
>
> **한정 대기(bounded waiting)**
>
> - 특정 프로세스가 영원히 임계 영역에 들어가지 못하면 안된다
>
> **융통성(progress)**
>
> - 만약 어떠한 프로세스도 임계영역을 사용하지 않는다면
>   - 임계 영역 외부의 어떠한 프로세스도 들어갈 수 있으며
>   - 이 때 프로세스끼리 서로 방해하지 않는다

뮤텍스(mutex)

- 프로세스나 스레드가 공유자원을 lock()을 통해 잠금 설정하고 사용한 후에 unlock()을 통해 잠금 해제하는 객체
- 잠금 설정시 다른 프로세스나 스레드는 잠긴 코드영역에 접근할 수 없고 해제는 반대임
- 뮤텍스는 잠금/잠금 해제만 가짐

세마포어(semaphore)

- 일반화된 뮤텍스
- 간단한 정수값과 두 가지 함수인 wait()과 signal() 함수로 공유 자원에 대한 접근을 처리
  - wait() : P 함수라고도 함, 자신의 차례가 올 때까지 기다리는 함수
  - signal() : 다음 프로세스로 순서를 넘겨주는 함수
- 프로세스나 스레드가 공유 자원에 접근시 세마포어에서 wait() 작업 수행
- 프로세스나 스레드가 공유 자원을 해제하면 세마포어에서 signal() 작업 수행

- 세마포어에는 조건 변수가 없음
    - 프로세스나 스레드가 세마포어 값을 수정할 때 다른 프로세스나 스레드는 동시에 세마포어 값을 수정할 수 없음

바이너리 세마포어

- 0과 1만 가질 수 있는 세마포어
- 뮤텍스와 바이너리 세마포어은 구현이 유사함, 그러나
    - 뮤텍스 : 잠금을 기반으로 상호 배제가 일어나는 '잠금 메커니즘'
    - 세마포어 : 신호를 기반으로 상호 배제가 일어나는 '신호 메커니즘'

카운팅 세마포어

- 여러 개의 값을 가질 수 있는 세마포어
- 여러 자원에 대한 접근을 제어하는 데 사용

모니터(monitor)

- 둘 이상의 스레드나 프로세스가 공유 자원에 안전하게 접근할 수 있도록 공유자원을 숨기고 해당 접근에 대해 인터페이스만 제공
- 모니터는 모니터 큐를 통해 공유 자원에 대한 작업들을 순차적으로 처리
- 모니터는 세마포어보다 구현하기 쉬움
    - 모니터에서 상호 배제는 자동
    - 세마포어는 상호 배제를 명시적으로 구현해야 함

3.3.8 교착상태

- 교착 상태(deadlock) : 두 개 이상의 프로세스들이 서로가 가진 자원을 기다리며 중단된 상태
    - 프로세스 A가 프로세스 B의 어떤 자원을 요청
    - 프로세스 B도 프로세스 A의 점유 자원을 오청

교착상태의 원인

- 상호 배제 : 한 프로세스가 자원을 독점하고 있으며 다른 프로세스들은 접근이 불가능
- 점유 대기 : 특정 프로세스가 점유한 자원을 다른 프로세스가 요청하는 상태
- 비선점 : 다른 프로세스의 자원을 강제적으로 가져올 수 없음
- 환형 대기 : 프로세스 A는 프로세스 B의 자원을 요구하고, 프로세스 B는 프로세스 A의 자원을 요구하는 등 서로가 서로의 자원을 요구하는 상황

교착 상태의 해결 방법

1. 자원을 할당할 때 애초에 조건이 성립되지 않도록 설계
2. '은행원 알고리즘' 사용
    a. 교착 상태 가능성이 없을 때만 자원할당됨
    b. 프로세스당 요청할 자원들의 최대치를 통해 자원 할당 기능 여부를 파악
3. 교착 상태 발생시 사이클이 있는지 찾아보고 이에 관련된 프로세스를 한 개씩 지움
4. 교착 상태는 매우 드물게 일어나므로 이를 처리하는 비용이 더 커서 교착 상태가 발생하면 사용자가 작업을 종료
    a. 현대 운영체제는 이 방법 채택

은행원 알고리즘

- 총 자원의 양과 현재 할당한 자원의 양을 기준으로 안정 또는 불안정 상태로 나눔
- 안정 상태로 가도록 자원을 할당하는 알고리즘

3.4 CPU 스케줄링 알고리즘

- CPU 스케줄러는 CPU 스케줄링 알고리즘에 따라 프로세스에서 해야 하는 일을 스레드 단위로 CPU에 할당
- 프로그램 실행시 CPU 스케줄링 알고리즘이 어떤 프로그램에 CPU 소유권을 줄 지 결정
- CPU 스케줄링 알고리즘의 목표
    - CPU 이용률 높게

- 주어진 시간에 많은 일을 하게
- 준비 큐에 있는 프로세스는 적게
- 응답 시간을 짧게 설정하는 것

3.4.1 비선점형 방식(non-preemptive scheduling)

- 프로세스가 스스로 CPU 소유권을 포기하는 방식
- 강제로 프로세스를 중지하지 않음
- 컨텍스트 스위칭으로 인한 부하가 적다

FCFS(First Come, First Served)

- 가장 먼저 온 것을 가장 먼저 처리
- 단점
  - 프로세스가 길게 수행되므로 ‘convoy effect’ 발생
  - convoy effect : 준비 큐에서 오래 기다리는 현상

SJF(Shortest Job First)

- 실행 시간이 가장 짧은 프로세스를 가장 먼저 실행
- starvation 일어남 : 긴 시간을 가진 프로세스가 실행되지 않는 현상
- 평균 대기시간이 가장 짧음
- 그러나 실제로는 실행 시간을 알 수 없으므로 과거의 실행했던 시간을 토대로 추측해서 용

우선순위

- 기존 SJF 스케줄링 : 긴 시간을 가진 프로세스가 실행되지 않는 현상
- 우선순위 : 오래된 작업일수록 우선순위를 높이는 방법을 사용해 보완
  - 우선순위 : SJF와 우선순위 뿐 아니라 FCFS 활용해서 만들기도 함
  - 선점형, 비선점형적 우선순위 스케줄링 알고리즘을 말하기도 함

13.4.2 선점형 방식(preemptive scheduling)

- 현대 운영체제가 쓰는 방식
- 지금 사용하는 프로세스를 알고리즘을 통해 중단시키고 강제로 다른 프로세스에 CPU 소유권을 할당하는 방식

라운드 로빈(RR, Round Robin)

- 현대 컴퓨터가 쓰는 선점형 알고리즘 스케줄링 방법
- 각 프로세스는 동일한 할당 시간을 주고 그 시간 안에 끝나지 않으면 다시 준비 큐의 뒤로 가는 알고리즘
- 할당 시간이 너무 크면 FCFS가 되고 짧으면 컨텍스트 스위칭이 잦아져 오버헤드, 비용이 커짐
- 일반적으로 전체 작업 시간은 길어지지만 평균 응답 시간은 짧아진다는 특징
- 로드 밸런서에서 트래픽 분산 알고리즘으로도 쓰임

SRF(Shortest Remaining Time First)

- SJF : 중간에 실행 시간이 더 짧은 작어비 들어와도 기존 짧은 작업을 모두 수행하고 그 다음 짧은 작업을 함
- SRF : 중간에 더 짧은 작업이 들어오면 수행하던 프로세스를 중지하고 해당 프로세스를 수행

다단계 큐

- 우선순위에 따른 준비 큐를 여러 개 사용
- 큐마다 라운드 로빈이나 FCFS 등 다른 스케줄링 알고리즘을 적용
- 큐 간 프로세스 이동이 안됨
    - 스케줄링 부담 적음
    - 유연성 떨어짐

데이터베이스

4.1 데이터베이스의 기본

4.1.1 엔터티

- 엔터티 : 여러 개의 속성을 지닌 명사

약한 엔터티와 강한 엔터티

- 예 : A가 혼자서는 존재하지 못하고 B의 존재에 따라 종속적일 때
    - A는 약한 엔터티
    - B는 강한 엔터티
- 예 : 방은 건물 안에서만 존재하므로 약한 엔터티, 건물은 강한 엔터티

4.1.2 릴레이션

- 데이터베이스에서 정보를 구분하여 저장하는 기본 단위
- DB에서는 테이블, NoSQL 데이터베이스에서는 컬렉션

테이블과 컬렉션

- MySQL 구조 : 레코드-테이블-데이터베이스
- MongoDB 구조 : 도큐먼트 - 컬렉션 - 데이터베이스

4.1.3 속성

- 속성 : 릴레이션에서 관리하는 구체적이며 고유한 이름을 갖는 정보

4.1.4 도메인

- 도메인 : 릴레이션에 포함된 각각의 속성들이 가질 수 있는 값의 집합

4.1.5 필드와 레코드

- 레코드 : 테이블에 쌓이는 행 단위의 데이터, 튜플이라고도 함

필드 타입

숫자타입

- TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT 등

날짜타입

- DATE, DATETIME, TIMESTAMP 등

DATE

- 날짜 부분 있고 시간 부분 없고, 3바이트

DATETIME

- 1000년 어쩌구 ~ 9999년
- 날짜 및 시간 부분 모두 포함, 8바이트

TIMESTAMP

- 1970년 어쩌구 ~ 2038년
- 날짜 및 시간 부분 모두 포함 4바이트

문자타입

CHAR와 VARCHAR

- CHAR : 고정 길이 문자열, 0 ~ 255
- VARCHAR : 가변 길이 문자열 : 0 ~ 65535

TEXT와 BLOB

- TEXT : 큰 문자열 저장
- BLOB : 이미지, 동영상 등 큰 데이터 저장

ENUM과 SET

- ENUM : ENUM 형태. 단일 선택만 가능, 잘못된 값 삽입시 빈 문자열
- SET : ENUM과 비슷하나 여러 개 데이터 선택 가능, 비트 단위 연산 가능
- ENUM과 SET 사용시 공간적 이점 있으나, 애플리케이션 수정시 DB도 수정해야 함

4.1.6 관계

1:1 관계

1:N 관계

N:M 관계

- 직접적으로 연결하지 않고 1:N과 1:M 테이블 두개로 나눠서 설정

4.1.7 키

기본키

- 유일성, 최소성 만족
- 자연키, 인조키 중 골라서 설정

자연키

- 속성 중 중복된 값 제외하면서 '자연스럽게' 뽑다가 나오는 키
- 자연키는 언젠가는 변한다

인조키

- auto increment 등 인위적으로 생성한 키
- 변하지 않음
- 따라서 보통 기본키는 인조키

외래키

- 다른 테이블의 기본 키를 그대로 참조하는 값
- 개체와의 관계를 식별하는 데 사용
- 중복 가능

후보키

- 기본키가 될 수 있는 후보들
- 유일성과 최소성을 동시에 만족

대체키

- 후보키가 두 개 이상일 경우 하나를 기본키로 지정하고 남은 후보키들

슈퍼키

- 각 레코드를 유일하게 식별할 수 있는 유일성을 갖춘 키

4.2 ERD와 정규화 과정

4.2.1 ERD의 중요성

- ERD는 시스템의 요구 사항을 기반으로 작성

- 이 ERD를 기반으로 데이터베이스 구축
- 데이터베이스 구축 이후에도 디버깅 또는 비즈니스 프로세스 재설계가 필요한 경우에 설계도 역할을 담당
- ERD는 관계형 구조로 표현할 수 있는 데이터를 구성하는데 유용
- 비정형 데이터를 충분히 표현할 수 없음

4.2.3 정규화 과정

- 정규화 과정 : 릴레이션 간의 잘못된 종속 관계로 인해 데이터베이스 이상 현상이 일어나서 이를 해결하거나, 저장 공간을 효율적으로 사용하기 위해 릴레이션을 여러 개로 분리하는 과정

정규형 원칙

- 정규형 원칙 : 같은 의미를 표현하는 릴레이션이지만
  - 좀 더 좋은 구조로 만들어야 하고
  - 자료의 중복성은 감소해야 하고
  - 독립적인 관계는 별도의 릴레이션으로 표현해야 하며
  - 각각의 릴레이션은 독립적인 표현이 가능해야 하는 것

제1정규형

- 릴레이션의 모든 도메인이 더 이상 분해될 수 없는 원자 값만으로 구성되어야 함
- 릴레이션의 속성 값 중 한 개의 기본키에 대해 두 개 이상의 값을 가지는 반복 집합이 있어서는 안됨
  - 반복 집합이 있다면 제거해야 함

제2정규형

- 릴레이션이 제 1정규형이며 부분 함수 종속성을 제거한 형태
- 부분 함수의 종속성 제거 : 기본키가 아닌 모든 속성이 기본키에 완전 함수 종속적인 것
  - 릴레이션을 분해할 때 동등한 릴레이션으로 분해해야 함
  - 정보 손실이 발생하지 않는 무손실 분해로 분해되어야 함

제3정규형

- 제 2 정규형이고 기본 키가 아닌 모든 속성이 이행적 함수 종속을 만족하지 않는 상태

이행적 함수 종속

- A → B와 B → C가 존재할 때 논리적으로 A → C 성립
    - 이 때 집합 C가 집합 A에 이행적으로 함수 종속이 되었다고 함

보이스/코드 정규형

- 제 3정규형이고 결정자가 후보키가 아닌 함수 종속 관계를 제거하여 릴레이션의 함수 종속 관계에서 모든 결정자가 후보키인 상태

4.3 트랜잭션과 무결성

- 트랜잭션 : 데이터베이스에서 하나의 논리적 기능을 수행하기 위한 작업의 단위
    - 여러 개의 쿼리들을 하나로 묶는 단위
    - 원자성, 일관성, 독립성, 지속성 ⇒ ACID

4.3.1 트랜잭션

원자성

- “ALL OR NOTHING”
- 트랜잭션과 관련된 일이 모두 수행되었거나 되지 않았거나를 보장하는 특징
- 트랜잭션 단위로 여러 로직들을 묶을 때 외부 API를 호출하는 것이 있으면 안됨
    - 만약 있다면 롤백이 일어났을 때 어떻게 해야 할 것인지에 대한 해결방법이 있어야 하고
    - 트랜잭션 전파를 신경써서 관리해야 함

커밋과 롤백

- 커밋 : 여러 쿼리가 성공적으로 처리되었다고 확정하는 명령어
    - 트랜잭션 단위로 수행되며 변경된 내용이 모두 영구적으로 저장되는 것
- 롤백 : 트랜잭션으로 처리한 하나의 묶음 과정을 일어나기 전으로 돌리는 일
- 커밋과 롤백으로 데이터 무결성이 보장됨

- 데이터 변경 전에 변경 사항을 쉽게 확인할 수 있음
- 작업을 그룹화할 수 있음

트랜잭션 전파

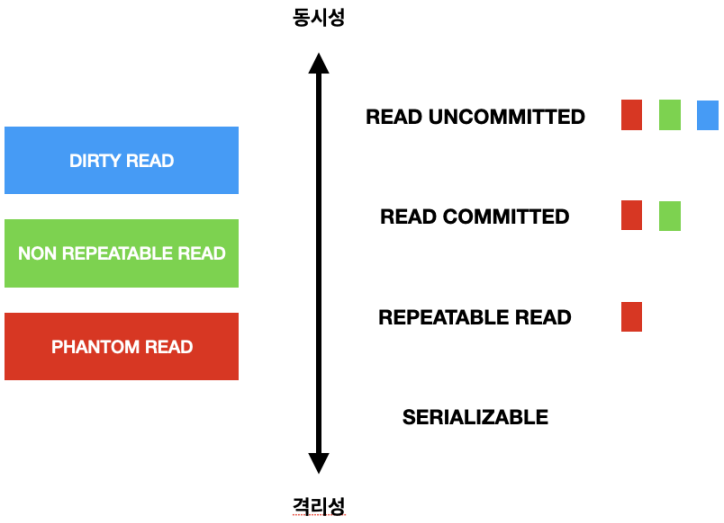
- 트랜잭션을 수행시 커넥션 단위로 수행하기 때문에 커넥션 객체를 넘겨서 수행해야 함
- 그러나 이를 매번 넘겨주기가 어렵고 귀찮기도 함
- 트랜잭션 전파 : 커넥션 객체를 넘겨서 수행하지 않고 여러 트랜잭션 관련 메서드의 호출을 하나의 트랜잭션에 묶이도록 하는 것

일관성

- '허용된 방식'으로만 데이터를 변경해야 하는 것

격리성

- 트랜잭션 수행 시 서로 끼어들지 못하는 것
- 격리성은 여러 개의 격리 수준으로 나뉘어 격리성 보장
- 격리성 수준 : SERIALIZABLE, REPEATABLE_READ, READ_COMMITED, READ_UNCOMMITED



- 위로 갈수록 동시성 강해짐, 격리성 약해짐
- 아래로 갈수록 동시성 약해짐, 격리성 강해짐

격리 수준에 따라 발생하는 현상

- 격리 수준에 따라 팬텀리드, 반복 가능하지 않은 조회, 더티리드가 발생할 수 있음

팬텀 리드

- 한 트랜잭션 내에서 동일한 쿼리를 보냈을 때 해당 조회 결과가 다른 경우

반복 가능하지 않은 조회

- 한 트랜잭션 내의 같은 행에 두 번 이상 조회가 발생했을 때 그 값이 서로 다른 경우
- 팬텀리드와 다른 점
    - 반복 가능하지 않은 조회는 행 값이 달라질 수 있음
    - 팬텀 리드는 다른 행이 선택될 수 있음

더티 리드

- 반복 가능하지 않은 조회와 유사
- 한 트랜잭션이 실행 중일 때 다른 트랜잭션에 의해 수정되었지만 아직 '커밋되지 않은' 행의 데이터를 읽을 수 있을 때 발생

격리 수준

SERIALIZABLE

- 트랜잭션을 순차적으로 진행시키는 것
- 여러 트랜잭션이 동시에 같은 행에 접근할 수 없음
- 매우 엄격한 수준
- 해당 행을 격리, 이후 트랜잭션이 이 행에 대해 일어나면 기다려야 함
- 교착 상태가 일어날 확률 많음
- 가장 성능 떨어지는 격리 수준

REPEATABLE_READ

- 하나의 트랜잭션이 수정한 행을 다른 트랜잭션이 수정할 수 없도록 막아주나 새로운 행을 추가하는 것은 막지 않음
- 따라서 이후에 추가된 행이 발견될 수도 있음

READ_COMMITED

- 가장 많이 사용되는 격리수준
- postgreSQL, SQL server, 오라클에서 기본 값
- 트랜잭션이 커밋하지 않은 정보는 읽을 수 없음 → 커밋 완료된 데이터에 대해서만 조회를 허용
- 하지만 어떤 트랜잭션이 접근한 행을 다른 트랜잭션이 수정할 수 있음

READ_UNCOMMITED

- 가장 낮은 격리수준
- 하나의 트랜잭션이 커밋되기 이전에 다른 트랜잭션에 노출되는 문제가 있음
- 가장 빠름
- 데이터 무결성을 위해 되도록 사용하지 않는 것이 이상적
- 몇몇 행이 제대로 조회되지 않더라도 괜찮은 거대한 양의 데이터를 '어림잡아' 집계하는 데 사용하면 좋다

지속성

- 성공적으로 수행된 트랜잭션은 영원히 반영되어야 하는 것
- 데이터베이스에 시스템 장애가 발생해도 원래 상태로 복구하는 회복 기능이 있어야 함
    - 체크섬, 저널링, 롤백 등을 제공

4.3.2 무결성

- 데이터의 정확성, 일관성, 유효성을 유지하는 것을 뜻함
- 무결성이 유지되어야 데이터베이스에 저장된 데이터 값과 그 값에 해당하는 현실 세계의 실제 값이 일치하는지에 대한 신뢰가 생김
- 무결성 종류

이름	설명

개체 무결성	기본기로 선택된 필드는 빈 값을 허용하지 않음
참조 무결성	서로 참조 관계에 있는 두 테이블의 데이터는 항상 일관된 값을 유지해야 함
고유 무결성	특정 속성에 대해 고유한 값을 가지도록 조건이 주어진 경우 그 속성 값은 모두 고유한 값을 가진다
NULL 무결성	특정 속성 값에 NULL이 올 수 없다는 조건이 주어진 경우 그 속성 값은 NULL이 될 수 없다

4.4 데이터베이스의 종류

4.4.1 관계형 데이터베이스

- 행과 열을 가지는 표 형식 데이터를 저장하는 형태의 데이터베이스
- SQL을 사용하여 조작

MySQL

postgreSQL

- VACUUM이 특징 : 디스크 조각이 차지하는 영역을 회수할 수 있음
- JSON을 사용해서 데이터에 접근할 수 있음 등

4.4.2 NoSQL 데이터베이스

- SQL을 사용하지 않는 데이터베이스

MongoDB

- JSON을 통해 데이터에 접근
- Binary JSON 형태(BSON)로 데이터 저장 등등
- 도큐먼트를 생성할 때마다 다른 컬렉션에서 중복된 값을 지니기 힘든 유니크한 값인 ObjectID가 생성됨

redis

- 인메모리 데이터베이스
- 키-값 데이터 모델 기반의 데이터 베이스

- 캐싱 계층, 세션 정보 관리, 채팅 시스템, 실시간 순위표 서비스에 사용

4.5 인덱스

4.5.1 인덱스의 필요성

- 데이터를 빠르게 찾을 수 있는 장치

4.5.2 B-트리

- 인덱스는 보통 B-트리라는 자료 구조로 이루어짐
- 루트노드, 리프노드, 브랜치 노드로 나뉨

인덱스가 효율적인 이유와 대수확장성

- 인덱스가 효율적인 이유 : 효율적인 단계를 거쳐 모든 요소에 접근할 수 있는 균형 잡힌 트리 구조와 트리 깊이의 대수 확장성 때문
- 대수확장성 : 트리 깊이가 리프 노드 수에 비해 매우 느리게 성장하는 것
    - 인덱스가 한 깊이씩 증가할 때마다 최대 인덱스 항목의 수는 4배씩 증가

4.5.3 인덱스 만드는 방법

MySQL

- 클러스터형 인덱스와 세컨더리 인덱스가 있음
- 클러스터형 인덱스 : 한 테이블당 하나
    - primary key 옵션으로 기본키 만들면 클러스터형 인덱스를 만들 수 있음
    - 기본키로 만들지 않고 unique not null 옵션을 붙이면 클러스터형 인덱스를 만들 수 있음
    - 하나의 인덱스만 생성할 것이라면 클러스터형 인덱스를 만드는 것이 세컨더리 인덱스를 만드는 것보다 성능이 좋음
- 세컨더리 인덱스 : 보조 인덱스
    - 여러 개의 필드 값을 기반으로 쿼리를 많이 보낼 때 생성해야 하는 인덱스
    - created index... 명령어 기반으로 세컨더리 인덱스를 만들 수 있음

MongoDB

- 도큐먼트를 만들면 자동으로 ObjectID가 형성됨
- 해당 키가 기본 키로 설정
- 세컨더리 키도 부가적으로 설정해서 기본키와 세컨더리키를 같이 쓰는 복합 인덱스를 설정할 수 있음

4.5.4 인덱스 최적화 기법

1. 인덱스는 비용이다

2. 항상 테스팅하라

3. 복합 인덱스는 같음, 정렬, 다중값, 카디널리티 순이다

4.6 조인의 종류

- MySQL에서는 JOIN, MongoDB에서는 lookup
- 몽고db에서 lookup은 되도록 사용하지 말 것
    - 조인 연산이 관계형 데이터베이스보다 성능이 떨어짐

4.6.1 내부 조인

- 두 테이블 간의 교집합

4.6.2 왼쪽 조인

- 왼쪽 A테이블 오른쪽 B테이블일 경우
- A테이블은 완전한 레코드 집합, B는 A테이블과 일치하는 부분의 집합
- B에 일치하는 부분 없으면 null

4.6.3 오른쪽 조인

- 왼쪽 조인과 반대임

4.6.4 합집합 조인

- 완전 외부 조인
- A테이블과 B테이블의 모든 레코드 집합 생성

- 일치하는 부분 없으면 null

4.7 조인의 원리

4.7.1 중첩 루프 조인

- 중첩 for문과 같은 원리로 조건에 맞는 조인을 하는 방법
- 랜덤 접근에 대한 비용이 많이 증가하므로 대용량의 테이블에서는 사용하지 않음
- 발전된 방식의 블록 중첩 루프 조인도 있음

4.7.2 정렬 병합 조인

- 각각의 테이블을 조인할 필드 기준으로 정렬하고 정렬이 끝난 이후에 조인 작업을 수행하는 조인
- 조인할 때 쓸 적절한 인덱스가 없고 대용량 테이블들을 조인하고 조인조건으로 <, > 등 범위 비교 연산자가 있을 때 씀

4.7.3 해시 조인

- 해시 테이블을 기반으로 조인하는 방법
- 두 개의 테이블을 조인할 때 하나의 테이블이 메모리에 온전히 들어간다면 보통 중첩 루프 조인보다 더 효율적
- 동등 조인에서만 사용할 수 있음
- MySQL의 해시 조인 단계는 빌드 단계, 프로브 단계로 나뉨

빌드 단계

- 입력 테이블 중 하나를 기반으로 메모리 내 해시 테이블을 빌드하는 단계
- 두 테이블 조인 시 바이트가 더 작은 테이블을 기반으로 해시 테이블을 빌드
- 또한 조인에 사용되는 필드가 해시 테이블의 키로 사용됨

프로브 단계

- 프로브 단계 동안 레코드 읽기를 시작하며, 각 레코드에서 일치하는 레코드를 찾아 결괏값으로 반환
- 이를 통해 각 테이블은 한 번씩만 읽게 되어 중첩해서 두 개의 테이블을 읽는 중첩 루프 조인보다 보통 성능은 더 좋음