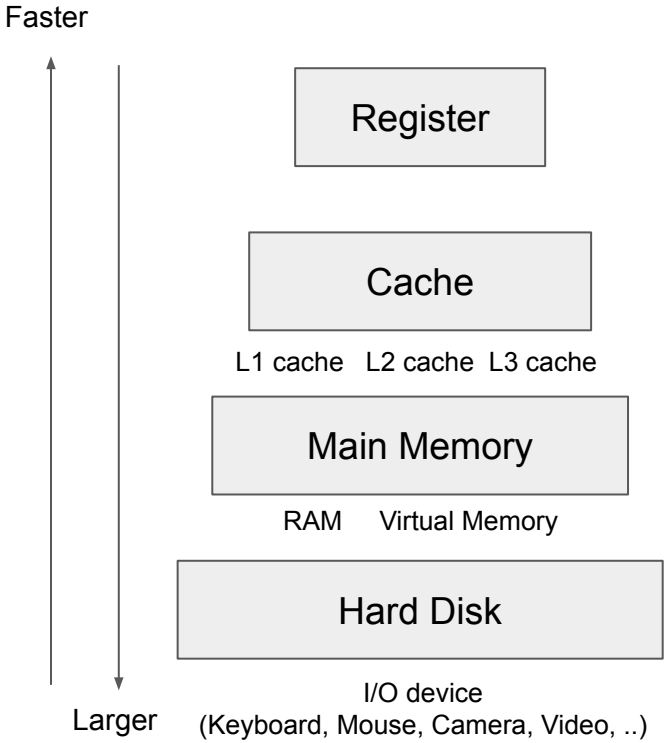
# Memory Hierarchy

**자주 쓰는 데이터를 CPU에서 더 빨리 접근하기 위함 → 전체적인 기억장치**

**액세스 속도 개선**




CPU 내부에 존재, CPU 연산 수행시 직접 참조, 데이터와 명령어 저장 (연산에 필요한)

CPU 내부에 존재, 데이터를 읽을 때 먼저 캐시에 있는지 살펴보고 있으면 캐시에서 직접 읽음, 없으면 main memory로 직접 접근
SRAM(메모리를 보존하기 위해 주기적으로 데이터 갱신하는 refresh X)으로 구성

CPU에서 직접 접근(CPU 외부에 존재), DRAM(메모리를 보존하기 위해 주기적으로 데이터 갱신하는 refresh O)으로 구성

CPU에서 직접 접근X → 접근 시, 데이터를 주기억장치로 옮겨 주기억장치에서 접근

# Memory Hierarchy - Cache

**CPU와 메모리의 속도 차이로 인한 병목현상 완화**

- DRAM 보다 빠른 SRAM 으로 구성

- CPU와 RAM 사이 중간 저장소 역할

- Main Memory에서 가장 자주 사용되는 데이터들의 집합

- 캐싱 : 프로세서에서 접근 속도가 빠른 계층의 메모리에 데이터를 미리 가져다 놓는 작업 속도

- L1 cache > L2 cache

- 연산에 필요한 데이터 찾을 시) L1 → L2 → Main Memory → Hard Disk

- 지역성 : 실행중인 프로세스가 주기억장치 참조할 때 일부 페이지만 집중적으로 참조하는 것

  시간 지역성 : CPU가 한 번 참조한 데이터는 또 참조할 가능성 높음

# Memory Hierarchy - Main Memory(1)

**DRAM 계열 메모리**

- RAM : 전원 꺼지면 기억된 내용이 사라지는 휘발성 메모리

- ROM : 전원이 꺼져도 기억된 내용이 사라지지 않는 비휘발성 메모리

- 주기억장치 관리 전략 : 반입 / 배치 / 교체

***반입*** : 보조기억장치에 있는 프로그램, 데이터를 언제 주기억장치로 적재할 것인지 결정

***배치*** : 새로 반입되는 프로그램, 데이터를 주기억장치의 어디에 위치시킬 것인지

***교체*** : 주기억장치의 모든 영역이 이미 사용중인 상태에서 가상기억장치의 필요한 페이지를

주기억장치에 배치하려고 할 때 이미 사용중인 영역에서 어느 영역을 교체할 것인지

단편화 : 분할된 주기억장치에 프로그램을 할당하고 반납하는 과정 반복하면 사용되지

# Memory Hierarchy - Main Memory(2)

**DRAM 계열 메모리**

- Virtual Memory : 보조기억장치의 일부를 주기억장치처럼 사용(현재 os에서 흔히 사용)

- 가상 기억장치의 프로그램 실행시 가상기억장치 주소 --mapping--> 주기억장치의 주소 필요

- 구현 기법 : 페이징 기법 / 세그먼테이션 기법

- ***페이징 기법*** : 가상기억장치에 저장된 프로그램과 주기억장치 영역을 동일한 크기로 나눈 후,

  나눠진 페이지를 동일하게 나눠진 주기억장치 영역(페이지 프레임)에 적재 후

  실행 → 페이지 맵 테이블 필요

- ***세그먼테이션 기법*** : 가상기억장치에 저장된 프로그램을 다양한 크기의 논리적인 단위

# Critical Section(임계구역)

- 여러 프로세스가 공유하는 자원에 하나의 프로세스만 자원을 사용하도록 지정된 영역

  → 임계 구역에는 하나의 프로세스만 접근(특정 프로세스 독점 X)

- WhyThreadMutex, FailedBank 클래스)

  FailedBank 내의 money라는 전역변수는 데이터 무결성 보장이 되지 않음

  (데이터 종속성이 없음)

```
a.start()
```

```
b.start();
```

두 메소드 호출시 각 스레드가 경쟁을 통해 run() 메소드를 실행하지만,
같은 변수임에도 데이터 무결성이 보장되지 않은 결과값이 나옴
-> 임계구역 설정 필요(Lock Mechanism)
-> 동기화(synchronized) 필요

# Synchronized(동기화)

- 두 개 이상의 스레드가 하나의 자원을 공유할 때 자원을 보호하기 위해 사용

- 공유 자원 접근시 다른 thread가 접근하지 못하도록 해당 자원을 lock 함

- 자원 사용 끝나면 unlock 을 하여 다른 thread의 접근을 허용함

- synchronized 키워드를 통해 강제로 Thread 간 순서 조정 → race condition 무마 O

- synchronized 메소드 이용 → 자신이 포함된 객체에 lock을 걺

```
public synchronized void plusMoney(int plus) {

    int m = getMoney();

public synchronized void minusMoney(int minus) {

    int m = getMoney();
```

synchronized 메소드를 통해 경쟁하는 다른 thread가 접근하지 못함으로써 m이라는 변수는 데이터의 무결성을 보장받게 된다 → thread 순서 조정을 통해 처음 메소드 실행 후 나온 결과값으로 다음 메소드 진행

# Synchronized(동기화)

- synchronized 블록을 이용 → 메서드 전체가 아닌 필요한 부분만 lock 처리

(메서드보다 thread 대기 시간 적음 → 성능 good)

```
synchronized (PerfSyncBankTest.psb) {

    PerfSyncBankTest.psb.plusMoney(3000);

}
```

문제가 될 부분만 블록처리로 임계구역 설정
→ 해당 블록을 제외한 다른 메소드와 자원들은 실행 중, 이 객체에만 다른 thread들이 접근 불가