OperatingSystem Note

1장. Introduction

운영체제는 컴퓨터 사용자와 컴퓨터 하드웨어 사이에서 중개자(intermediary) 역할

사용자가 프로그램을 편리하고 효율적으로 수행할 수 있는 환경을 제공하는 데 있다.

컴퓨터 하드웨어를 관리하는 소프트웨어, 운영체제를 자원 할당자

응용 프로그램을 위한 기반을 제공

하드웨어, 운영체제, 응용 프로그램 및 사용자로 구분

- 하드웨어: 하드웨어는 중앙 처리 장치(CPU), 메모리 및 입출력(I/O) 장치로 구성되어, 기본 계산용 자원을 제공한다
- 응용 프로그램: 응용 프로그램이란 워드 프로세서, 스프레드시트, 컴파일러, 웹 브라우저 등을 말하며 사용자의 계산 문제를 해결하기 위해 이들 자원이 어떻게 사용될지를 정의한다.
- 운영체제: 위 구성도에서 운영체제는 다양한 사용자를 위해 다양한 응용 프로그램 간의 하드웨어 사용을 제어하고 조정하는 역할을 한다.
- 사용자: 컴퓨터에 대한 사용자의 관점은 사용되는 인터페이스에 따라 달라진다.

인터럽트: 하드웨어는 어느 순간이든 시스템 버스를 통해 CPU에 신호를 보내 인터럽트를 발생시킬 수 있다.

인터럽트는 다른 많은 목적으로도 사용되며 운영체제와 하드웨어의 상호 작용 방식의 핵심 부분이다.

CPU가 인터럽트 되면,

- CPU는 하던 일을 중단하고 즉시 고정된 위치로 실행을 옮긴다. 이러한 고정된 위치는 일반적으로 인터럽트를 위한 서비스 루틴이 위치한 시작 주소를 가지고 있다.
- 그리고 인터럽트 서비스 루틴이 실행된다.
- 인터럽트 서비스 루틴의 실행이 완료되면, CPU는 인터럽트 되었던 연산을 재개한다.

인터럽트는 최신 운영체제에서 비동기 이벤트를 처리하기 위해 사용된다.

저장장치 구조: CPU는 메모리에서만 명령을 적재할 수 있으므로 실행하려면 프로그램을 먼저 메모리에 적재해야 한다.

범용 컴퓨터는 프로그램 대부분을 메인 메모리(RAM)라는 재기록 가능한 메모리에서 가져온다.

컴퓨터는 다른 형태의 메모리도 사용하는데, 예를 들어 부트스트랩 프로그램은 운영체제를 적재하고 있으며 컴퓨터 전원을 켤 때 가장 먼저 실행되는 프로그램이다. 이때 RAM은 휘발성(전원이 꺼질 때 내용이 손실되는) 메모리이므로 부트스트랩 프로그램을 유지하는 용도로 사용할 수 없다.

대신 컴퓨터는 전기적으로 소거 가능한 프로그램 가능 읽기 전용 메모리(EEPROM) 및 기타 형태의 펌웨어를 사용한다.

CPU - 명령을 실행하는 하드웨어
프로세서(processor) - 하나 이상의 CPU를 포함하는 물리적 칩

코어(core) - CPU의 기본 계산 단위
다중 코어(multicore) - 동일한 CPU에 여러 컴퓨팅 코어를 포함함
다중 처리기(multiprocessor) - 여러 프로세서를 포함함

멀티프로그래밍과 멀티태스킹

운영체제의 가장 중요한 측면 중 하나는 하나의 프로그램은 항상 CPU나 I/O 장치를 바쁘게 유지할 수 없으므로 여러 프로그램을 실행할 수 있다는 것이다. (한 프로그램이 모든 장치를 사용해 동작하지 않는다는 의미)

또한 사용자는 일반적으로 한 번에 둘 이상의 프로그램을 실행하려 한다.

메인 메모리의 특징

메인 메모리는 CPU와 입출력 장치에 의해 공유되는, 빠른 접근이 가능한 데이터의 저장소이다. 일반적으로 CPU가 직접 주소를 지정할 수 있고, 직접 접근할 수 있는 유일한 대량 메모리이다

2장. Operating-System Structures

운영체제는 프로그램과 그 프로그램의 사용자에게 특정 서비스를 제공

- 사용자 인터페이스
- 프로그램 수행
- 입출력 연산
- 파일 시스템 조작
- 통신
- 오류 탐지
- 자원 할당
- 기록 작성(logging)

시스템 콜은 운영체제에 의해 사용 가능하게 된 서비스에 대한 인터페이스를 제공

시스템 콜은 다섯 가지의 중요한 범주로 묶을 수 있다.

- 프로세스 제어
- 파일 조작
- 장치 조작
- 정보 유지 보수
- 통신과 보호

3장. Processes

프로세스

프로세스란 실행 중인 프로그램이다.

프로세스의 현재 활동의 상태는 프로그램 카운터(pc) 값과 프로세서 레지스터의 내용으로 나타낸다.

- text : 실행 코드
- data : 전역 변수
- heap : 프로그램 실행 중에 동적으로 할당되는 메모리
- stack : 함수를 호출할 때 임시 데이터 저장장소(ex : 함수 매개변수, 복귀 주소 및 지역 변수)

프로세스 상태 (Process State)

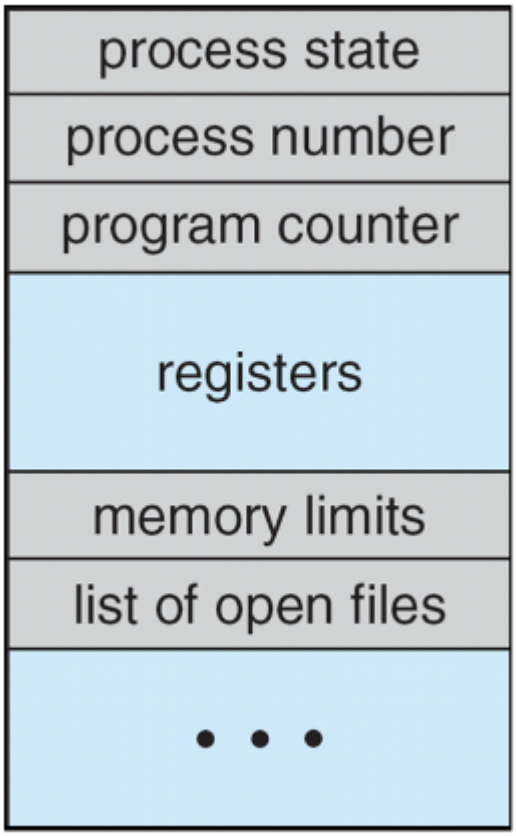
new : 프로세스가 생성 중이다. running : 명령어들이 실행되고 있다. waiting : 프로세스가 어떤 이벤트(입출력 완료 또는 신호의 수신)가 일어나기를 기다린다. ready : 프로세스가 처리기에 할당되기를 기다린다. terminated : 프로세스의 실행이 종료되었다.

중요한 것은 어느 한순간에 한 프로세서 코어에서는 오직 하나의 프로세스만이 실행된다는 것이다.

하지만 많은 프로세스가 waiting, ready 상태에 있을 수는 있다.

프로세스 제어 블록 (Process Control Block)

각 프로세스는 운영체제에서 프로세스 제어 블록(Process Control Block, PCB)에 의해 표현된다



스레드 (Threads)

위에서 설명한 프로세스 모델은 한 프로세스가 단일의 실행 스레드를 실행하는 프로그램이라 가정하였다.

하지만 현대 운영체제는 한 프로세스가 다수의 실행 스레드를 가질 수 있도록 허용한다. 이러한 특성은 여러 스레드가 병렬로 실행될 수 있다는 것을 의미한다.

프로세스 스케줄링

다중 프로그래밍의 목적은 CPU 이용을 최대화하기 위하여 항상 어떤 프로세스가 실행되도록 하는 데 있다.

시분할의 목적은 각 프로그램이 실행되는 동안 사용자가 상호 작용할 수 있도록 프로세스들 사이에서 CPU 코어를 빈번하게 교체하는 것이다.

이 목적을 달성하기 위해 프로세스 스케줄러는 코어에서 실행 가능한 여러 프로세스 중에서 하나의 프로세스를 선택한다.

CPU 스케줄링

프로세스는 수명주기 동안 준비 큐와 다양한 대기 큐를 이동한다.

CPU 스케줄러의 역할은 준비 큐에 있는 프로세스 중에서 선택된 하나의 프로세스에 CPU 코어를 할당하는 것이다.

문맥 교환 (Context Switch)

인터럽트는 운영체제가 CPU 코어를 현재 작업에서 뺏어 커널 루틴을 실행할 수 있게 한다.

인터럽트가 발생하면 시스템은 인터럽트 처리가 끝난 후에 문맥을 복구할 수 있도록 현재 실행 중인 프로세스의 문맥을 저장할 필요가 있다.

이 문맥은 프로세스의 PCB에 표현된다.

4장. Threads & Concurrency

스레드는 CPU 이용의 기본 단위이다. 스레드는 스레드 ID, 프로그램 카운터(PC), 레지스터 집합, 그리고 스택으로 구성된다.

스레드의 장점

1. 응답성
2. 자원 공유
3. 경제성
4. 규모 적응성

컴퓨팅 시스템이 발전되면서, 단일 컴퓨팅 칩에 여러 컴퓨팅 코어를 배치할 수 있게 되었다.

각 코어는 운영체제에 별도의 CPU로 보이게 되는데, 이러한 시스템을 다중 코어라고 한다.

스레드들이 병렬적으로 실행된다.

5장. Process Scheduling

프로세스 실행은 CPU 버스트와 I/O 버스트의 반복이다.

CPU 스케줄러

CPU가 유휴 상태가 될 때마다, 운영체제는 준비 큐에 있는 프로세스 중 하나를 선택해 실행해야 한다.

선택 절차는 CPU 스케줄러에 의해 수행된다. (주의점은 선입선출 방식의 큐가 아니어도 된다는 점이다.)

선점 및 비선점 스케줄링

CPU 스케줄링 결정은 아래 4가지 상황에서 발생할 수 있다.

- 한 프로세스가 실행 상태에서 대기 상태로 전환될 때 (ex. I/O 요청이나 자식 프로세스가 종료되기를 기다리기 위해 wait()를 호출할 때)
- 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때 (ex. 인터럽트가 발생할 때)

- 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때 (ex. I/O 종료 시)
- 프로세스가 종료할 때

스케줄링 기준

- CPU 이용률
- 처리량
- 총 처리 시간
- 대기 시간
- 응답 시간

9장. Main Memory

포토샵은 굉장히 큰 프로그램이기 때문에 모든 프로그램을 메모리에 올리면 비효율적이다.

따라서, 본 프로그램은 메모리에 올리고, 나머지는 "사용자가 필요할 때"마다 가져오는것이 효율적이다.

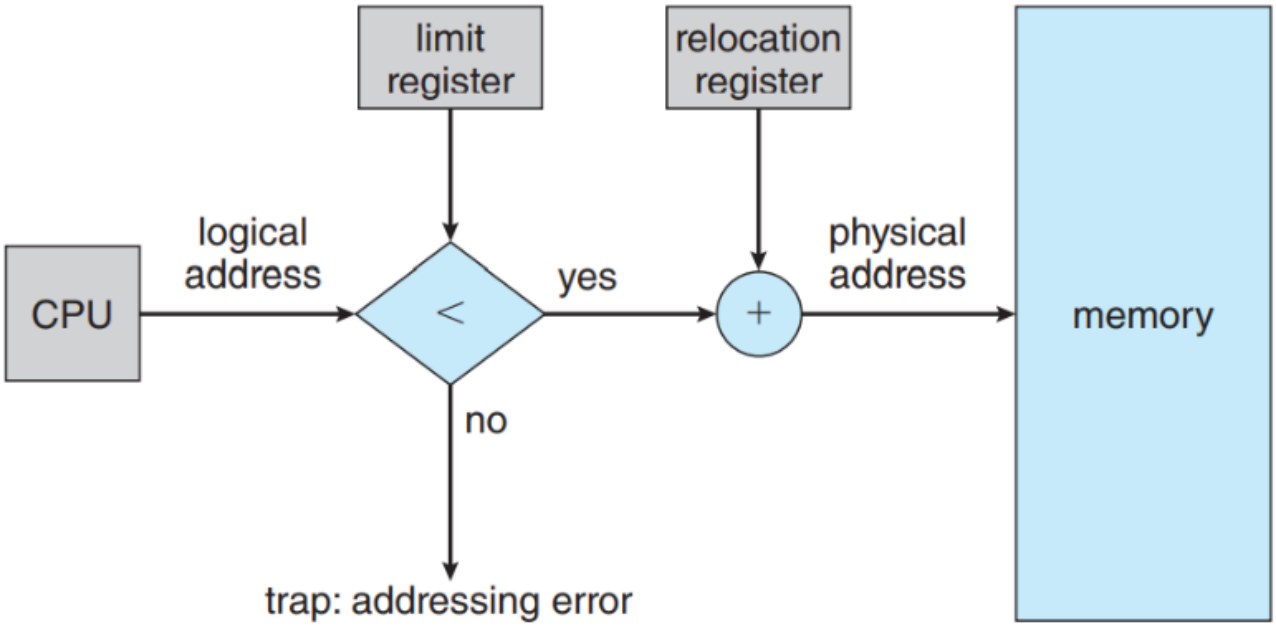
(요구 페이징)

Base 레지스터 Limit 레지스터로 해당 프로세스의 메모리 영역을 지정한다.

if (base <= logical_address < base + limit) then ok

else trap

trap인터럽트를 발생하여 운영체제에 알림



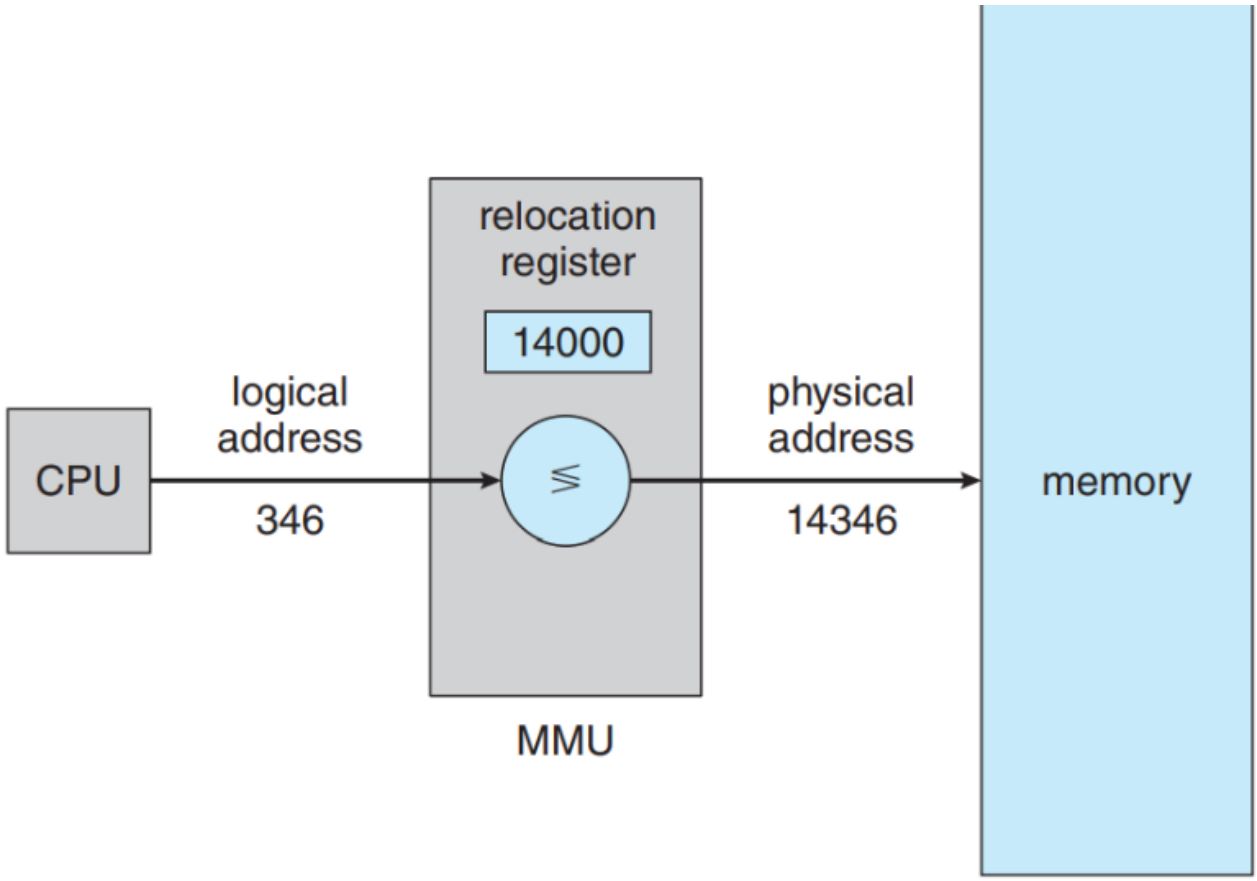
CPU에서 생성한 주소를 논리 주소 logical address라 부름. 반면에 메모리 장치에서 인식하는 주소, 즉 메모리의 메모리 주소 레지스터 memory-address register에 저장하는 주소를 물리 주소 physical address라 부름.

컴파일 시나 로드 시에 생성한 논리 주소는 물리 주소와 동일하지만, 실행 시에 생성한 논리 주소는 물리 주소와 다름. 이 경우 논리 주소를 가상 주소 virtual address라 부름. 앞으로 논리 주소와 가상 주소를 사실상 동일하

게 부를 것임. 프로그램이 생성한 가상 주소 집합을 가상 주소 공간 logical address space라 부름. 반대로 물리 주소 집합을 물리 주소 공간 physical address space라 부름.

시행 시 가상 주소를 물리 주소로 매핑해주는 과정은 메모리 관리 장치 memory-management unit (MMU)라는 하드웨어 장치에서 담당

재배치 레지스터 relocation register라 부를 것임. 재배치 레지스터의 값은 사용자 프로세스의 주소가 메모리에 보내질 때 사용자 프로세스가 생성한 모든 주소에 더해짐



동적 적재 좀 더 메모리 공간을 효율적으로 사용하기 위해 동적 적재 dynamic loading를 사용할 수 있음

모든 루틴은 디스크에 재배치 가능한 형태로 저장되어있음. 주프로그램이 메모리에 적재되어 실행이 됨. 이때 한 루틴이 다른 루틴을 호출하게 되면 호출자가 호출한 루틴들이 적재되어있는지 우선 확인함. 적재 안 되어있음 재배치 가능한 링킹 로더를 불러와 호출한 루틴들을 메모리에 적재하여 프로그램의 주소 테이플을 갱신하여 변화를 줌. 이후 제어권을 새롭게 적재한 루틴에 넘겨줌.

메모리 할당

메모리 할당하는 가장 간단한 법은 프로세스를 메모리에 딱 넣을 수 있을 만큼의 크기로 분할하여 넣어주는 것임. 이런 가변 분할 variable partition 스킴에서는 운영체제가 메모리 중 어느 부분이 현재 사용 중인지를 기록해야함. 처음엔 전부 사용 가능한 상태일 것이고, 하나의 거대한 메모리 덩어리, 어떻게 보면 하나의 거대한 구멍 hole이라고 할 수 있음. 나중에 보면 알겠지만 메모리는 구멍이 송송 난 상태가 될 것임.

동적 저장 공간 할당 문제 dynamic storage-allocation problem의 한 형태임. 동적 저장 공간 할당 문제란 주어진 구멍들 목록에서 크기 n만큼의 메모리를 요구하는 요청을 들어주는 방법을 찾는 문제임. 여러 해결책이 있음. 최초 적합 first-fit, 최적 적합 best-fit, 최악 적합 worst-fit 등이 가장 일반적으로 사용하는 방법임.

- 최초 적합. 메모리를 탐색해서 공간이 충분한 처음 구멍에 할당. 탐색 방법은 보통 앞에서부터 하거나 마지막으로 탐색했던 곳에서부터 시작함. 암튼 충분히 큰 구멍 찾으면 탐색 끝임.
- 최적 적합. 공간이 충분한 구멍들 중 가장 크기가 작은 구멍에 할당. 구멍들의 목록이 사전에 정렬되어있는거 아니면 목록 전부 탐색해야함. 잉여 구멍은 제일 작음.
- 최악 적합. 공간이 충분한 구멍들 중 가장 크기가 큰 구멍에 할당. 잉여 구멍은 제일 큰데, 이게 오히려 최적 적합보다 더 좋을 수도 있음.

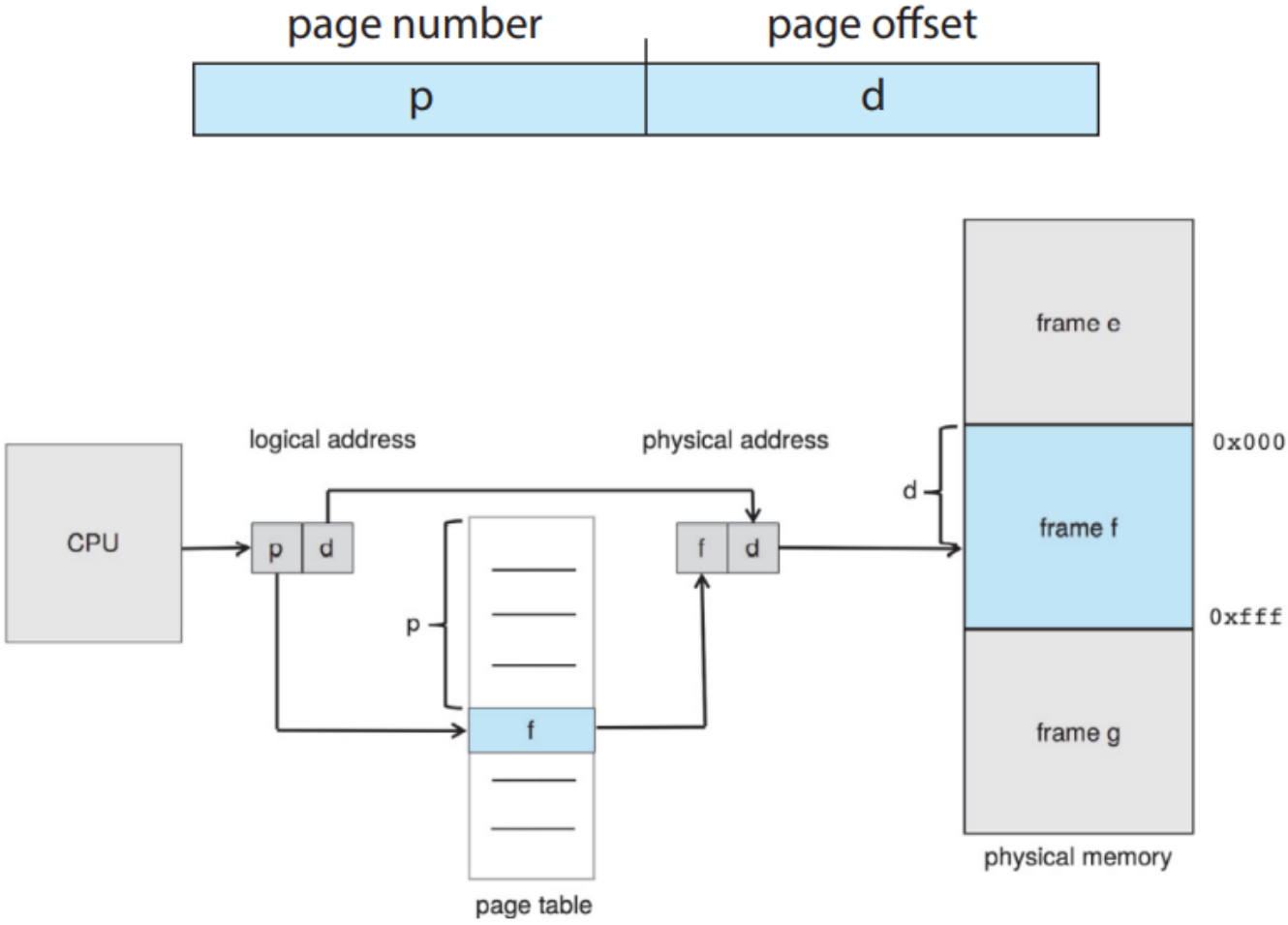
최초 적합과 최적 적합 둘 다 외부 파편화 external fragmentation라는 문제가 생김

이걸 해결하기 위해서 물리 메모리를 고정 크기 블록으로 나눈 다음, 메모리를 할당할 땐 이 크기 단위로 할당을 하는 것임. 이러면 프로세스에 할당한 메모리가 본래 메모리보다 더 클 수도 있음. 이렇듯 크기 단위에 따라 할당한 메모리와 요구한 메모리 간의 차가 바로 내부 파편화 internal fragmentation임. (페이징)

다른 해결책으로는 가상 주소 공간이 비연속적일 수 있게 해주어 물리 메모리에 공간이 있는 곳마다 할당해주는 것임. 이 전략이 바로 페이징 paging으로, 컴퓨터 시스템에서 가장 일반적인 메모리 관리 기법임.

지금까지는 프로세스의 물리 주소 공간이 연속적이어야한다고 했지만, 페이징 paging 기법을 사용하면 비연속적이어도 됨. 페이징을 사용하면 메모리를 감염시키는 주범인 외부 파편화를 해결할 수 있으니 또다른 주범 압축도 필요 없음. 장점이 워낙 많다보니 여러 운영체제에서 여러 방법으로 페이징을 사용함. 페이징은 운영체제와 하드웨어가 좀 서로 협력해야 구현이 가능함.

페이징 구현의 기본은 물리 메모리를 고정된 크기의 블록, 즉 프레임 frame으로 나누고, 가상 메모리를 이와 동일한 크기의 블록, 즉 페이지 page로 나누는 것임. 프로세스를 실행하려면 프로세스의 페이지를 프로세스가 있는 곳(파일 시스템이나 보조 기억 장치)에서 사용 가능한 메모리 프레임에 적재함. 보조 기억 장치는 메모리 프레임과 동일한 고정된 크기의 블록 혹은 여러 프레임의 군집으로 나뉘어짐. 이게 아이디어는 간단한데 이게 기능성이 매우 좋고, 다양하게 활용할 수 있음



MMU가 CPU가 생성한 가상 주소를 물리 주소로 변환할 때 다음 단계를 거침:

- 페이지 수 p 구하여 페이지 테이블의 색인으로 삼음
- 페이지 테이블에서 이 색인에 대응하는 프레임 수 f 구함 = 가상 주소의 페이지 수 p를 프레임 수 f로 교체

페이징을 하면 외부 파편화가 없어짐. 빈 프레임은 결국 필요한 프로세스에게 할당이 되니까. 하지만 내부 파편화는 발생함.

10장. Virtual-Memory Management

가상 메모리를 통해 프로세스가 전부 다 메모리에 안 올라가있어도 실행이 가능하게 해줄 수 있음. 이러면 프로그램이 물리 메모리보다 클 수 있다는 장점

또한 프로세스가 파일과 라이브러리를 서로 공유하게 할 수 있으며, 공유 메모리도 구현 가능함. 추가적으로 효율적인 프로세스 생성 메커니즘도 제공함. 가상 메모리가 구현하기가 쉬운 건 아니지만 대충 사용할 경우 성능이 좀 많이 떨어짐.

- 프로그램을 작성할 때 오류 상황 처리할 코드를 작성해야할 때가 있음. 물론 실무에서는 거의 발생 안할 수도 있어 아예 실행이 안 될 수도 있음.
- 배열, 리스트, 테이블은 보통 실제 필요한 것보다 더 많이 메모리를 잡곤 함. 배열 안에 원소가 겨우 열 몇 개 밖에 없어도 100 개 씩 잡고 그럼.
- 프로그램의 특정 옵션이나 기능이 거의 사용이 안 될 수도 있음.

프로그램 일부만 올려도 되면:

- 프로그램이 더 이상 실제 물리 메모리에 의해 제약을 받지 않아도 됨. 사실상 매우 거대한 가상 주소 공간에 넣을 프로그램도 작성할 수 있게 되어 프로그래밍이 간단해짐.
- 물리 메모리를 실제로 덜 먹을테니까 동시에 프로그램을 더 많이 실행할 수 있으므로 CPU 효용과 처리율도 높아지면서도 응답 시간이나 턴어라운드 시간이 늘어나지도 않음. = 프로그램의 일부를 메모리에 적재하거나 스왑하는데 필요한 입출력이 적어지므로 프로그램이 더 빠르게 돌 것임.

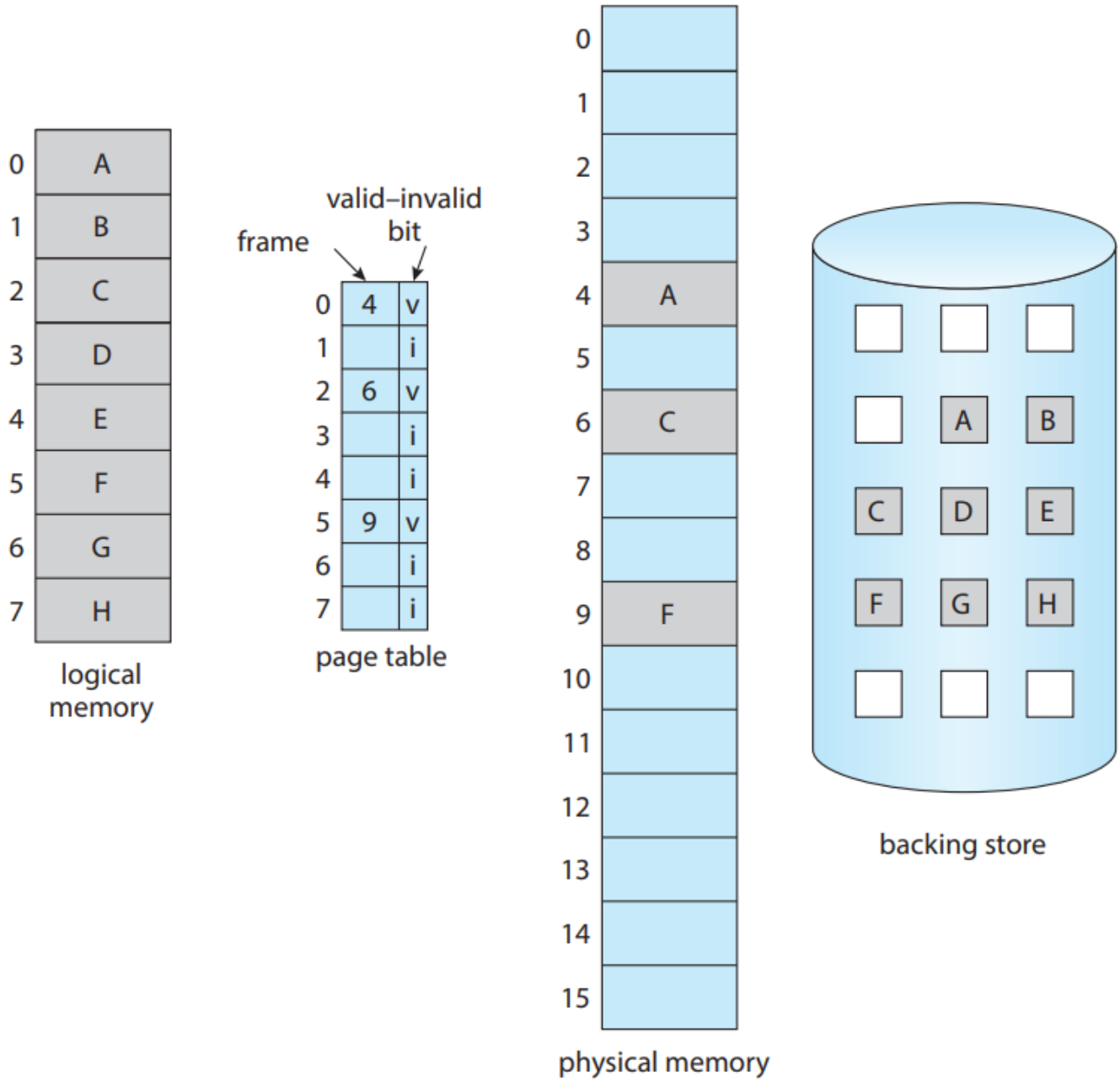
즉, 프로그램을 일부만 올릴 수 있게 되면 시스템 뿐만 아니라 사용자에게도 좋음.

가상 메모리 virtual memory는 개발자가 인식하는 논리 메모리를 실제 물리 메모리와 구분하는 것으로 시작

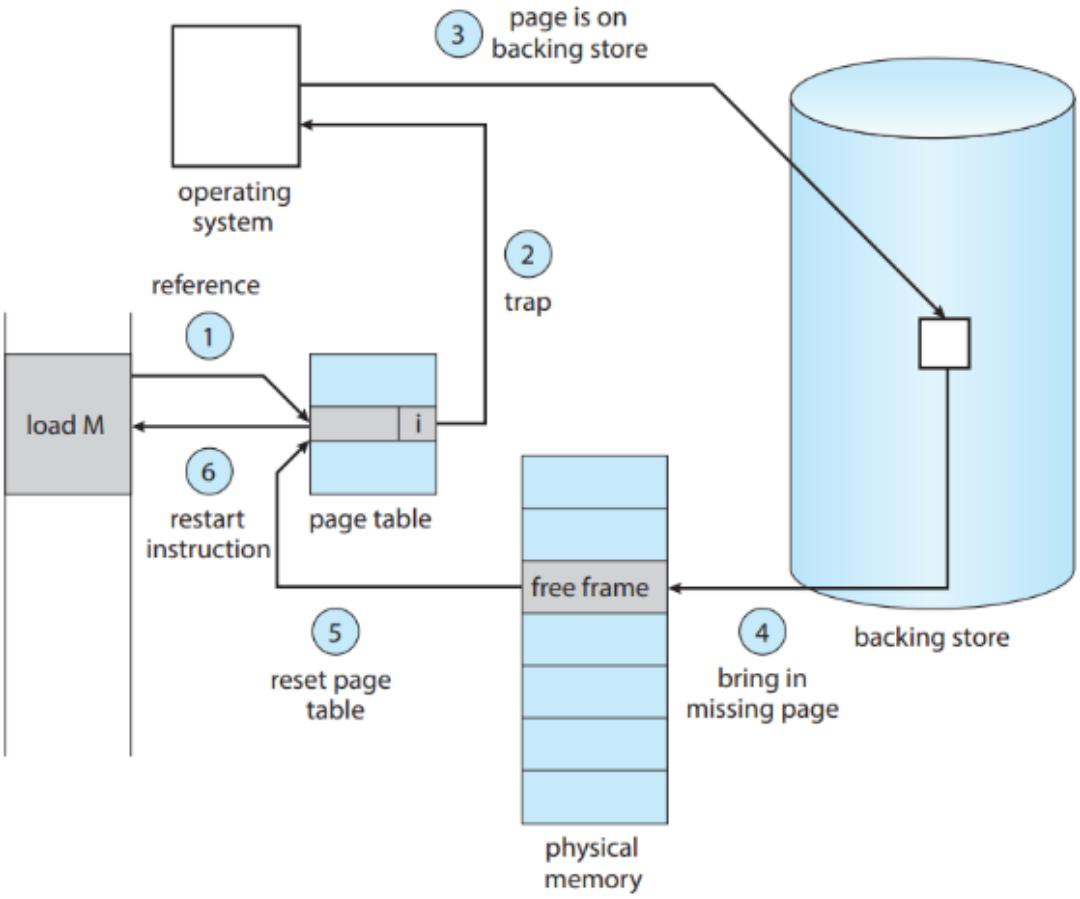
실행할 프로그램을 보조 기억 장치에서 메모리로 올리는 방법 중 하나는 프로그램 전체를 프로그램 실행 시에 물리 메모리에 올리는 건데, 위에서도 언급했듯 프로그램 전체가 필요하지 않을 수도 있음. 사용자가 선택할 수 있는 부분이 많은 프로그램의 경우 사용자가 선택하지 않은 옵션도 포함한모든 옵션을 메모리에 올리게 되버림...

다른 방법은 필요한 페이지만 올리는 것으로, 이걸 요구 페이징 demand paging이라 부르고, 가상 메모리 시스템에서 일반적으로 사용하는 방법임. 이러면 프로그램 실행 중에 추가적인 페이지를 요구할 때만 페이지를 적재해줌

요구 페이징의 기본 개념은 위에서도 언급했듯 필요할 때만 페이지를 메모리에 올리는 것임. 즉, 실행 중인 부분이 메모리에 올라가있다면, 나머지 부분은 보조 기억 장치에 있다는 뜻이므로, 이 둘을 구분할 수 있게 하드웨어단에서 지원해줘야함. 9.3.3 절에서 언급했던 유부호 비트 스킴을 여기에 적용하면 됨. 근데 이번엔 비트가 설정된 상태, 즉 유효한 상태는 연관된 페이지가 합법인 페이지이면서 메모리 위에 있다는 의미로 사용하고, 비트가 설정되지 않았을 때, 즉 무효 상태는 페이지가 유효하지 않거나(즉, 프로세스의 가상 주소 공간에 속하지 않음) 유효하긴 한데, 현재 보조 기억 장치 안에 있다는 의미로 사용함. 메모리에 올린 페이지 테이블 엔트리는 기존과 같지만, 현지 메모리에 없는 페이지는 엔트리에 단순히 무효하다고 표기만 해주면 됨.



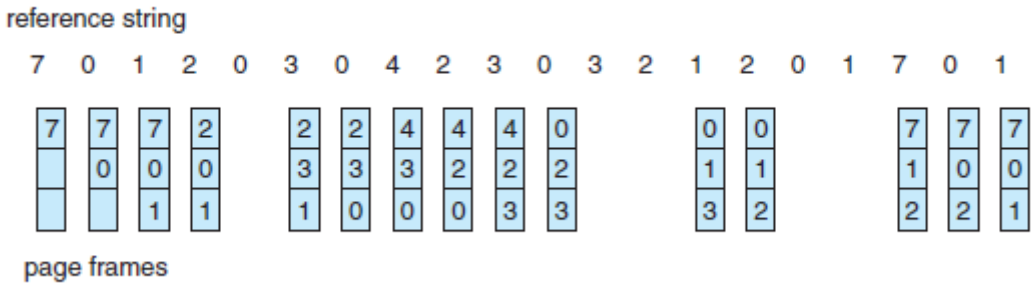
프로세스가 메모리에 없는 페이지에 접근하려고 하면 페이지 부재 page fault가 발생하게 됨. 페이징 하드웨어 입장에서는 주소를 변환할 때 페이지 테이블을 참고해서 보는데, 무효 비트가 설정되어있으니 운영체제에 트랩을 보낼 것임. 운영체제 입장에서는 페이지를 메모리에서 불러와야하는데 실패했으니까 트랩이 발생한 것



1. 이 프로세스의 내부 테이블(프로세스 제어 블록이랑 같이 저장되어있곤 함)을 확인해서 해당 참조의 메모리 접근의 유무효 여부를 판단.
2. 만약 무효한 참조라면 프로세스 종료. 유효하지만 페이지를 아직 불러온게 아니었다면 이제 이걸 메모리에 올리면 됨.
3. 올릴 수 있는 프레임을 찾음(사용 가능한 프레임 목록에서 하나 가져오는 식으로).
4. 올려야할 페이지를 새롭게 할당한 프레임에 읽어올 보조 기억 장치 연산을 스케줄링해줌.
5. 기억 장치에서 다 읽어오면 이제 페이지가 메모리에 올라갔다는 걸 알려줘야하니까 프로세스랑 같이 저장해뒀던 내부 테이블이랑 페이지 테이블을 수정해줌.
6. 트랩에 의해 인터럽트됐던 명령어 다시 재시작해줌. 마치 페이지가 처음부터 메모리에 존재했던 것 마냥 돌 것임.

FIFO Page Replacement

메모리에 가장 오래 올라와 있던 page를 바꾸는 방법이다.

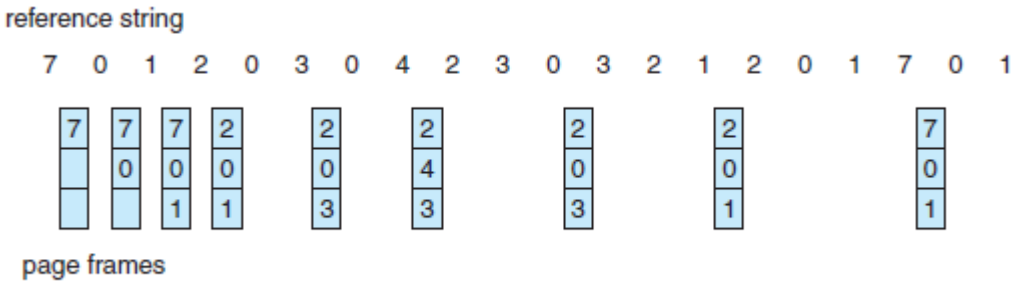


예시 page reference string에 대해 15개의 page fault를 일으킨다.

Belady의 모순은 프로세스에게 frame을 더 주었음에도, page fault가 더 증가하는 것을 의미한다.ㄴ

Optimal Page Replacement

앞으로 가장 오랫동안 사용되지 않을 페이지를 찾아 교체하는 방법이다.

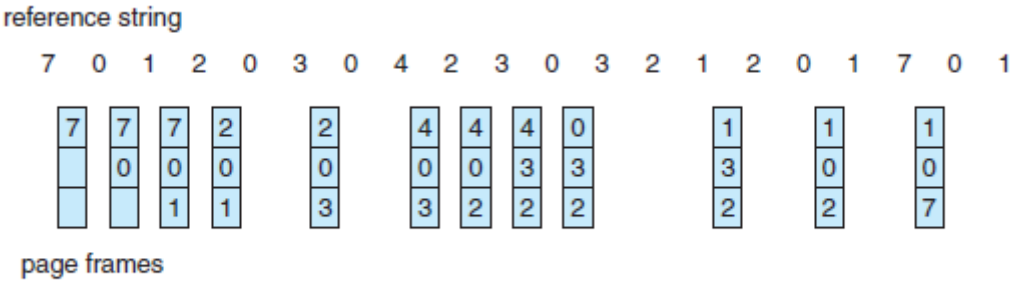


예시 page reference string에 대해 9개의 page fault를 일으킨다.

미래를 예측해야하기 때문에 실제로 구현하기 힘들다.

LRU Page Replacement

LRU는 각 페이지마다 마지막 사용 시간을 유지하고, 페이지 교체 시에 가장 오랫동안 사용되지 않은 페이지를 교체한다.



예시 page reference string에 대해 12개의 page fault를 일으킨다.

LRU 알고리즘은 페이지 교체 알고리즘으로 자주 사용되며, 좋은 알고리즘으로 인정받고 있다.

Mass-Storage Systems

Mass-Storage

- 비휘발성 (non-voliatie, Secondary Storage
- 보통 하드디스크나 NVM이라 칭함, 제 2 저장 장치 혹은 보조 저장장치라고 함
- 때로는 마그네틱 테입이나, 광학 디스크, 클라우드 저장소를 사용하는 경우가 있음

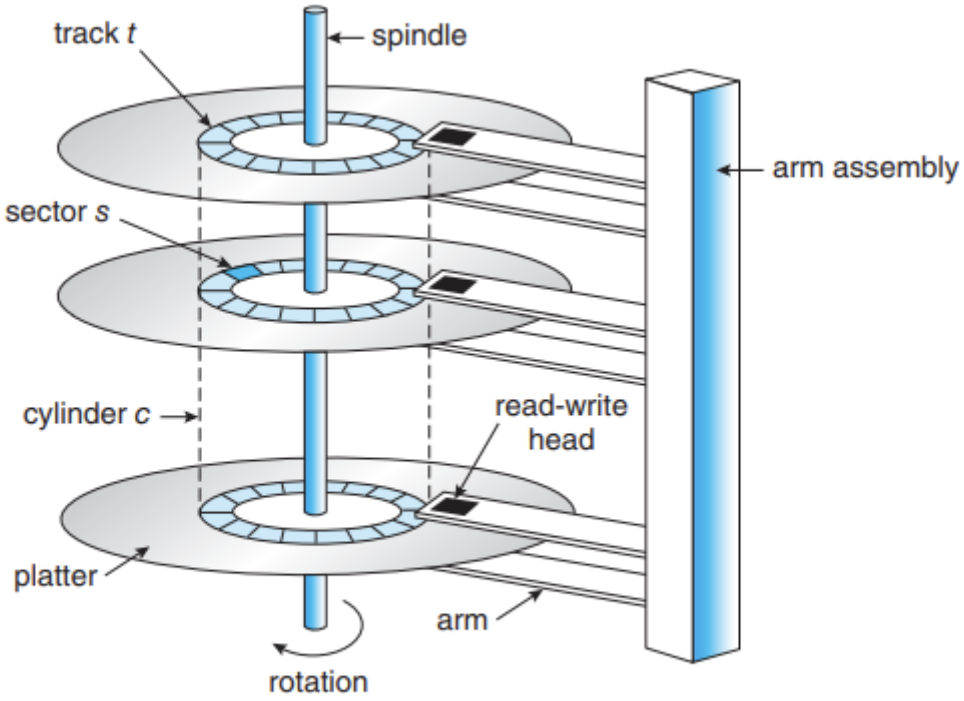


Figure 11.1 HDD moving-head disk mechanism.

- FIFO Scheduling
  - 먼저 온 요청을 먼저 처리. head movement가 효율적이지는 않음

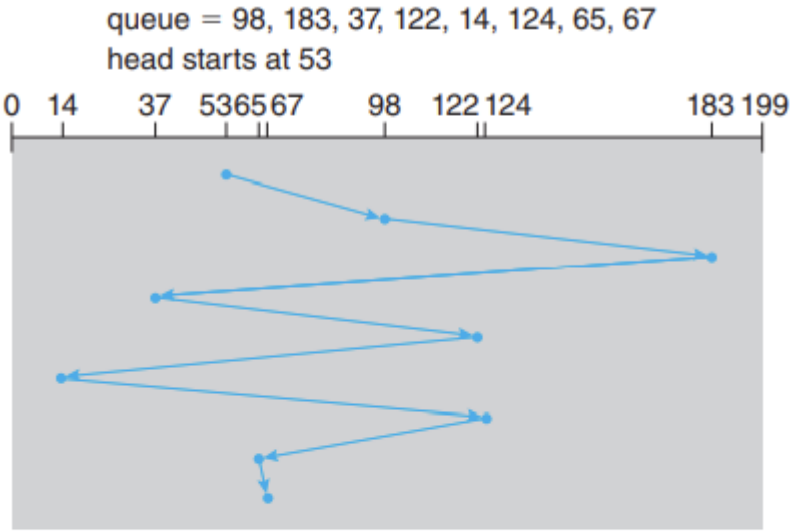


Figure 11.6 FCFS disk scheduling.

98, 183, 37, 122, 14, 124, 65, 67,

- SCAN Scheduling
  - 한쪽 종단에서 종단으로 계속 움직이면서 들어온 Request들을 처리함
  - 하지만 양방향으로 움직이는 것 보다는 한 방향으로 움직이는 게 구현이 편하기 때문에 S-SCAN이라는 게 나옴

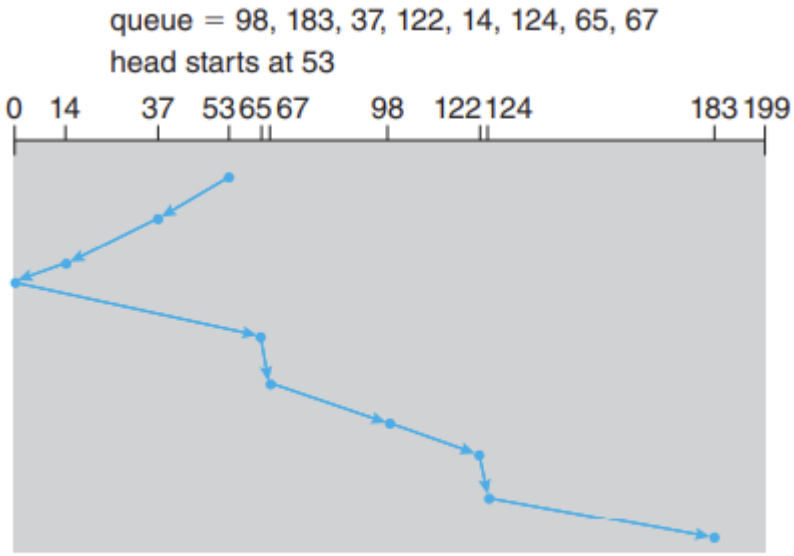


Figure 11.7 SCAN disk scheduling.

- C-SCAN (Circular-SCAN) Scheduling
  - 한 방향으로만 헤드를 움직이고, 시작점에서 끝점까지 읽어들인 다음에 되돌아올때에는 아무것도 읽지 않고 다시 첫 시작점으로 돌아온다 (단, 되돌아올 때 움직임은 무시함)

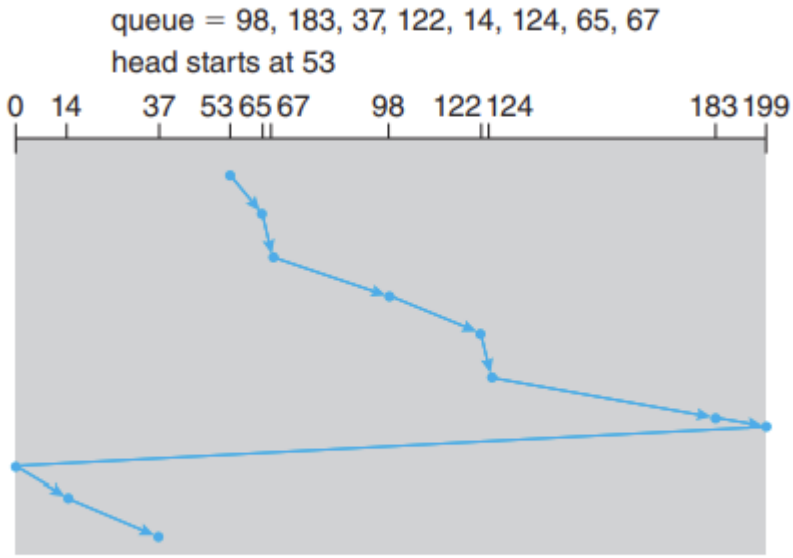


Figure 11.8 C-SCAN disk scheduling.

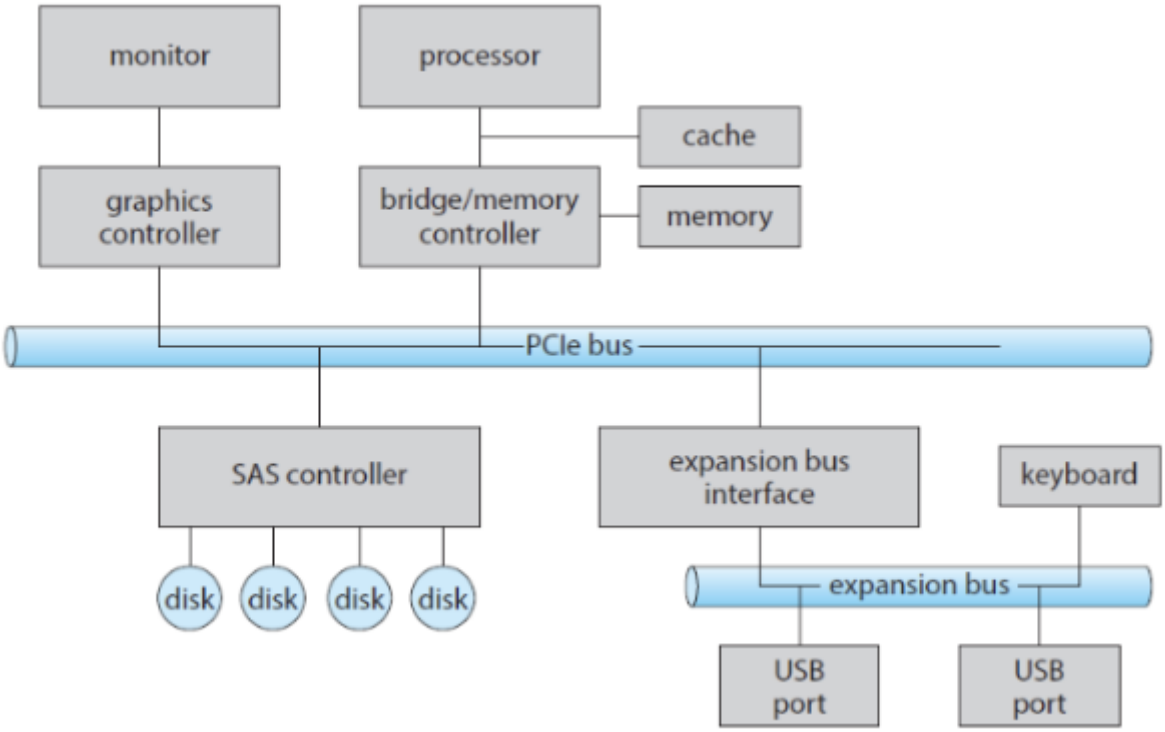
- Boot Block
  - 전원이 꺼졌을 때 컴퓨터를 구동시키기 위한 첫번째 프로그램을 bootstrap이라 했지! Boot Block을 flash memory(ROM)에 저장함

12장. I/O Systems

대부분 컴퓨팅에서 I/O는 주된 작업임

I/O 명령어와 I/O 장치들을 제어하는 것은 OS의 할 일

PC bus (CPU가 bus를 통해 각 영역의 컨트롤러에게 명령을 내림!)



- Memory-Mapped I/O--

    - I/O 장치에 내리는 명령을 어떻게 전달할 것인가?
        - data-in register
        - data-out register
        - status register
        - control register
    - Memory-mapped I/O : I/O Address에 어떤 interrupt controller가 매핑되어 있는 지 메모리에 매핑 시켜둠
    - 그래서 디바이스에 직접 명령어를 전달하지 않고, 메모리에 I/O 명령을 주어서 control register의 역할을 할 수 있게 됨

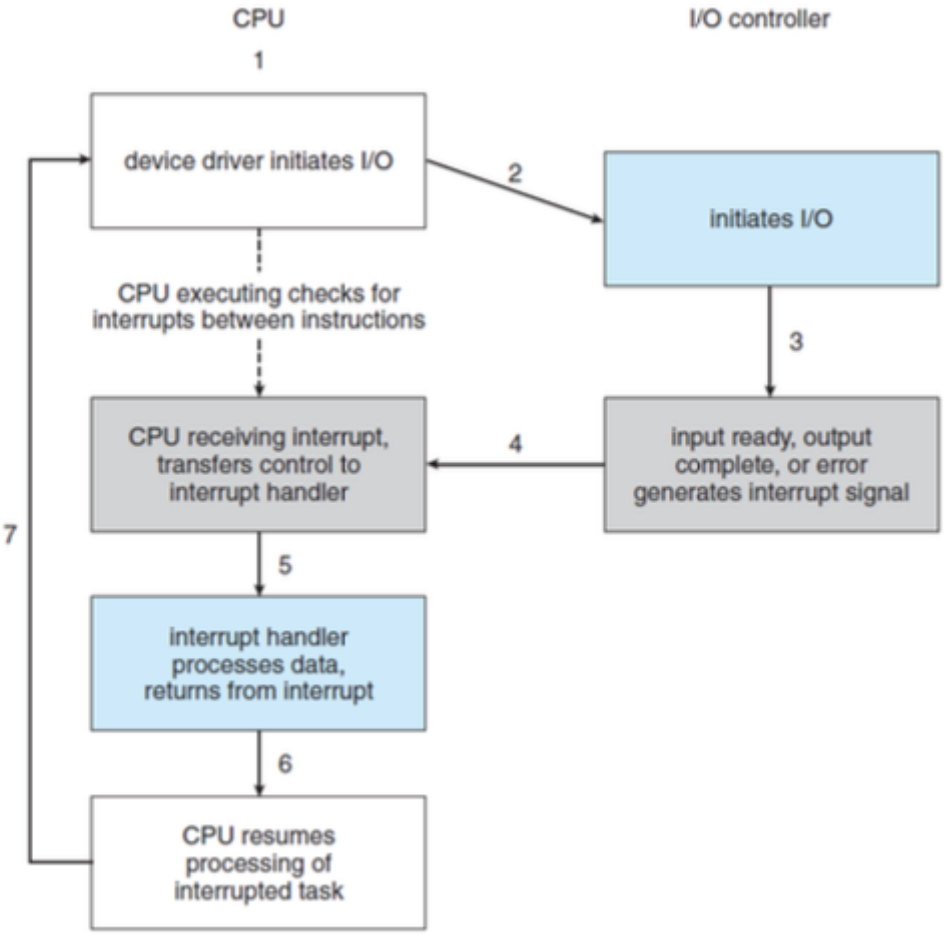
세 가지 유형의 I/O

- polling : or busy-waiting

상태 레지스터의 상태를 계속 읽으면서 원하는 값이 올때까지 기다림

- interrupt

interrupt driven I/O : wait 과 signal (이미지 첨부)



- DMA : Direct Memory Access

아주 대용량 전송의 경우 다이렉트로 메모리에 접근함

13장. File System Interface

- 파일은 운영체제의 의하여 정의되고 구현되는 추상적인 자료형이다.

- 파일은 논리 레코드의 연속으로서, 바이트, 행 또는 좀 더 복잡한 자료 항목들이다.
- 운영체제는 다양한 레코드형을 사용자에게 제공하거나 아니면 사용자가 프로그램상으로 정의하도록 해준다.
- 운영체제의 가장 중요한 문제는 논리적인 파일을 실제 저장장치(하드)에 어떻게 매핑시키는 것인지
    - 보통 물리 레코드 크기는 논리 레코드와 일치하지 않기 때문에, 논리 레코드를 물리 레코드에 연관시켜야한다.
    - 이 작업은 운영체제에 의하여 제공되거나 사용자의 응용 프로그램에서 할 수 있다.
- 파일 시스템 내에서 파일을 구조화하는 디렉터리를 만드는 것이 유용하다.
    - 다중 사용자 시스템에서 1단계 디렉터리는 모든 파일이 고유의 이름을 가져야 하므로 파일 명칭 부여 문제를 유발한다.
    - 2단게 디렉터리는 각 사용자에 대하여 독자적인 디렉터리를 할당함으로써 이 문제를 해결한다.
    - 디렉터리는 이름으로 파일을 기록하고, 디스크 상에서의 파일의 위치, 길이, 형태, 소유자, 생성시간, 마지막 사용 시간등 과 같은 정보를 포함한다.
- 파일 또한 디스크의 공간을 회수하기 위해 가비지 컬랙션이 필요
    - 2단계 디렉터리를 일반화하여 확장하면 트리 구조 디렉터리가 되는데 사용자만의 서브디렉터리를 구성할 수 있게 해준다.
    - 일반적인 그래프 디렉터리는 디렉터리와 파일의 공유에 완전한 융통성을 주는 대신 사용되지 않는 디스크 공간을 회수하기 위해 가비지 컬랙션을 필요로 한다.

Contiguous Allocation (연속 할당)

하나의 파일이 디스크 상에 연속해서 저장되는 방식

- 단점
    - 외부 조각이 발생한다.
    - 파일의 크기를 키우는 데 제약이 있다. 파일을 수정하면서 파일의 크기가 커질 수 있는데, 그것에 제약이 있다는 것이다. 그래서 미리 여유 공간을 할당할 수는 있지만, 이 경우 내부 조각이 발생할 수 있다.
- 장점
    - 빠른 입출력이 가능하다.
    - 한번의 seek/rotation으로 많은 바이트를 전송할 수 있다.
    - realtime file용으로, 또는 이미 run 중이던 프로세스의 swapping용으로 사용 가능하다.
    - 직접 접근(임의 접근)이 가능하다.

Linked Allocation (연결 할당)

파일의 데이터를 디스크에 연속적으로 배치하는 대신, 빈 위치 아무데나 들어갈 수 있게 배치하는 방식

- 장점
    - 외부 조각이 발생하지 않는다.
- 단점
    - 직접 접근(임의 접근)이 불가능하다.
    - reliability 문제
    - 하나의 sector가 bad sector가 되어 pointer가 유실되면 이후 부분을 모두 잃어버리게 된다.
    - 포인터를 위한 공간이 block의 일부가 되어 공간 효율성을 떨어뜨린다.
    - 디스크에서 하나의 sector는 512byte를 차지하는데, 포인터가 4byte를 차지하므로 비효율적이다.

- 변형
    - File-allocation table (FAT) 파일 시스템
    - 포인터를 별도의 위치에 보관하여 reliability와 공간 효율성의 문제를 해결한다. (아래에서 자세히 다룸)

Indexed Allocation (인덱스 할당)

블록 하나에 위치 정보를 저장하는데, 이 블록을 인덱스 블록이라고 한다. 디렉토리는 파일의 위치 정보 대신 인덱스 블록 값을 저장한다.

- 장점
    - 외부 조각이 발생하지 않는다.
    - 직접 접근(임의 접근)이 가능하다.
    - (연속 할당 / 연결 할당의 단점을 모두 극복)
- 단점
    - 작은 파일인 경우 공간이 낭비된다. (실제로 많은 파일들의 크기가 작다.)
    - 너무 큰 파일인 경우 하나의 블록으로 인덱스를 모두 저장할 수 없다.
    - 해결 방안
        - linked scheme
            - 인덱스 블록의 마지막 위치가 또 다른 인덱스 블록을 가리키게 한다.
        - multi-level index
            - 인덱스 블록 내의 값들이 각각 인덱스 블록을 가리킨다.
            - (2단계 페이징 테이블과 유사)