Chapter1) 운영체제의 개요

(1) 운영체제 소개

1. 일상생활 속의 운영체제

1) 운영체제(OS, Operating System)

• 운영체제는 일반 컴퓨터나 노트북의 전원을 켜면 가장 먼저 만나게 되는 소프트웨어

ex) 윈도우, Mac OS – 작은 컴퓨터용 / 유닉스, 리눅스 – 큰 컴퓨터용

모바일 운영체제(iOS – 애플사는 폐쇄정책을 사용, 안드로이드 – 범용적인 모바일 운영체제 등)

2) 임베디드 운영체제(embedded system)

• CPU의 성능이 낮고 메모리 크기도 작은 시스템에 내장하도록 만든 운영체제

-> 특수 목적에 있는 장치에 컴퓨팅 능력을 부여함.

-> 컴퓨터 기능이 내장되어 있다는 것은 CPU역할을 하는 프로세서와 메모리가 있음

• 임베디드 운영체제는 CPU의 성능이 낮고 메모리 크기가 작은 기계에 설치되기 때문에 일반 운영체제에 비해 몇 가지 기능이 빠져 있다.

• 장점

① 임베디드 운영체제가 있는 기계는 기능을 계속 향상 할 수 있다는 장점이 있다.

② mp3 플레이어 같은 경우에는 재생하는 기능 외에도 음질을 향상하거나 MP3 파일 외에도 FLAC, WMA, Ogg 등 다양한 종류의 음악 파일을 지원할 수 있으며 무선 통신을 이용해 MP3 파일을 공유 가능하다.

③ 전화기 같은 경우에는 운영체제가 없기 전에는 통화 기능 외에 특별한 기능이 없었지만 운영체제를 사용하게 되면서 응용 프로그램 앱이 설치가 하여 게임이나 쇼핑 등 다양한 용도로 사용할 수 있다.

2. 운영체제의 필요성

1) 컴퓨터는 운영체제 없이도 작동하는가?

• 운영체제 없이도 잘 동작하지만 기능에 제약이 있다.

• 예를 들면 에니악은 운영체제 없고 키보드, 모니터도 없지만 정상적으로 작동했지만 크기에 비해 성능이

매우 낮아 공학 전자계산기 수준에 불과했다. 하지만 컴퓨터로 부르는 이유는 전공관과 진공관을 전선

으로 연결하여 프로그래밍이 가능하기 때문이다. 그러나 새로운 기능을 구현하려면 매번 회로를 변경해야

하는 번거로움이 있다.

2) 운영체제가 있는 기계와 없는 기계는 어떤 차이가 있는가?

• 운영체제가 있는 기기는 다양한 응용 프로그램을 설치하여 사용할 수 있고 성능 향상을 위한 새로운

기능을 추가 할 수 있다.

• 운영체제가 없는 기기는 기계가 만들어진 당시에 구현한 기능 외에도 다른 기능을 추가하거나 성능을

향상할 수 없기 때문에 처음에 설계한 대로만 사용가능

• 운영체제가 있는 기계는 새로운 기능이 추가 되거나 성능의 변경이 가능하여 성능 및 효율성의 향상을

꾀할 수 있다.

3) 운영체제는 성능을 향상하는 데만 필요한가?

• 운영체제는 컴퓨터 성능 향상할 뿐만 아니라 자원을 관리하고 사용자에게 편리한 인터페이스 환경 제공

• 운영체제는 CPU를 최대로 사용하여 하드웨어를 효율적으로 사용하고 하드웨어의 중요한 부분을

건드리지 못하게 안전하게 사용할 수 있게 만들어 준다.

• 많은 응용 프로그램이 컴퓨터를 구성하는 장치를 서로 독차지하려고 한다. 이 때, 운영체제가 강력한

중재자 역할을 하여, 컴퓨터 자원을 나누어 준다.

4) 운영체제는 자원을 어떻게 관리하는가?

• 운영체제는 사용자가 직접 자원에 접근하는 것을 막음으로써 컴퓨터 자원을 보호한다.

• 악의적인 사용자가 데이터를 지우거나 덮어쓰는 일을 방지하기 위해 응용 프로그램과 사용자에게

모든 자원을 숨긴다.

5) 사용자는 숨어 있는 자원을 어떻게 이용할 수 있는가?

• 운영체제가 제공하는 사용자 인터페이스와 하드웨어 인터페이스를 이용하여 자원에 접근한다.

• 운영체제는 사용자가 컴퓨터를 보다 쉽게 사용할 수 있도록 다양한 인터페이스를 제공함으로써 컴퓨터

자원을 보호함과 동시에 사용자의 편의를 도모한다.

• 사용자한테 편리성을 제공하여 사용자가 하드웨어에 대해 자세히 몰라도 자원을 이용할 수 있게끔

도와준다. 자원을 사용하는 방법은 초창기는 cmd창이었는데 지금은 그래픽으로 발전하여 편리성이

점점 증가해진다.

3. 운영체제 정의

1) 운영체제는 사용자에게 편리한 인터페이스 환경을 제공하고 컴퓨터 시스템의 자원을 효율적으로

관리하는 소프트웨어이다.

2) 운영체제는 응용 프로그램이나 사용에게 컴퓨터 자원을 사용할 수 있는 인터페이스를 제공하고

그 결과를 돌려준다.

3) 응용 프로그램이나 사용자에게 모든 컴퓨터 자원을 숨기고 정해진 방법으로만 컴퓨터 자원을 사용할

수 있도록 제한한다. 그래서 운영체제가 허락한 곳에만 자원을 사용할 수 있도록 해준다.

4) 사용자가 하드디스크의 특정 위치에 저장이 가능하게 되면 기존 데이터와 섞여서 엉망이 되므로,

운영체제는 자신을 거치지 않고는 하드웨어를 접근하지 못하게 해준다.

5) 운영체제는 사용자에게 인터페이스를 제공하고 컴퓨터 자원한테는 자원을 보호하고 효율적으로 사용할

수 있게끔 해준다.

4. 운영체제의 역할

1) 자원 관리

• 돌아가는 프로그래밍이 있으면 적절하게 메모리를 분배해주고 CPU를 최대한 쉬게 하여 여러 프로그램을

돌릴 수 있게 한다.

• 자원을 요청한 프로그램이 여러 개라면 적당한 순서로 자원을 배분하고 적절한 시점에 자원을 회수하여

다른 응용 프로그램에 나누어 준다.

2) 자원 보호

• 커널 영역에 사용자가 함부로 접근하지 못하게 한다.

• 비정상적인 작업으로부터 컴퓨터 자원을 지켜야 한다.

3) 하드웨어 인터페이스 제공

• 운영체제는 복잡한 과정 없이 다양한 장치를 사용할 수 있도록 해주는 하드웨어 인터페이스를 제공하여

하드웨어의 종류에 상관없이 사용할 수 있게 해준다.

• 운영체제는 CPU, 메모리, 키보드, 마우스와 같은 다양한 하드웨어를 일관된 방법으로 사용할 수 있도록

지원해준다.

4) 소프트웨어 인터페이스 제공

• 사용자 인터페이스는 사용자가 운영체제를 편리하게 사용하도록 지원한 것이다.

• 과거에는 cmd창으로만 제공하여 불편했지만 그래픽 사용자 인터페이스를 제공하게 되면서 사용자의

편의성이 높아졌으며, 더 나아가 스마트폰에 터치스크린 적용이 되어 보다 편리한 인터페이스를 제공

받게 된다.

5. 운영체제의 목표

1) 효율성

• 운영체제는 자원을 효율적으로 관리하는 것을 목표로 한다.

• 같은 자원을 사용하여 더 많은 작업량을 처리하거나, 같은 작업량을 처리하는 데 보다 적은 자원을

사용하는 것이다.

• 효율적으로 자원을 관리하는 방법을 다양하지만 일반적으로는 운영체제의 크기를 최소화하고 운영체제가

사용하는 코드를 최적화하는 것이다.

2) 안정성

• 작업을 안정적으로 처리하려면 사용자와 응용 프로그램의 안전 문제와 하드웨어적인 보안 문제를 처리할

수 있어야 한다.

• 시스템에 문제가 발생했을 대 이전으로 복구하는 결함 포용 기능(fault tolerant)을 수행해야 한다.

3) 확장성

• 운영체제는 다양한 시스템 자원을 추가하거나 제거하기가 편리해야 한다. 즉 확장성이 좋아야 한다.

• 컴퓨터로 장치에 연결 했을 때 따로 설치 하는 게 없어야 한다.

4) 편리성

• 사용자가 편리하게 작업을 할 수 있는 환경을 제공하는 것이 운영체제의 목표이다.

• 효율성만 높이려고 하면 응용 프로그램에게는 자원 사용을 제약을 걸게 하고 사용자에게는 텍스트만

입력하도록 하면 된다. 하지만 이와 같이 사용하면 사용자는 텍스트로만 사용해야 한다는 불편함과 다양한

응용 프로그램이 개발이 되지 않는다. 따라서 응용 프로그램과 사용자에게 다양한 편리성을 제공하면서도

자원의 낭비 요소를 막아야 한다.

(2) 운영체제의 역사

1. 운영체제 역사 개요

일괄 처리 시스템 -> 다중 프로그래밍 시스템 / 온라인 시스템 -> 시분할 시스템 -> 실시간 처리 시스템

-> 다중 모드 처리 시스템 -> 분산 처리 시스템 -> 병렬 처리 시스템

2. 초창기 컴퓨터 시대(1940년대)

• 애니악

1) 초창기의 컴퓨터에는 운영체제가 없다.

2) 진공관이라는 소자를 사용하여 진공관이 켜지면 1, 꺼지면 0으로 판단하여 2진법을 사용

3) 전선을 연결하여 논리회로를 구성하는 방식은 하드와이어링(hard wiring)방식이며, 전선으로 논리회로를

구성하는 방식이므로, 다른 계산이나 수식을 사용하려면 전선을 다시 연결해야 한다.

3. 일괄 처리 시스템(1950년대)

• 천공 카드 시스템

- 천공 카드 리더는 구멍을 뚫어 문자나 숫자를 표현하여 프로그래밍을 작성하는 입력 장치로 사용하였고,

라인 프린터는 문자만 출력하는 프린터로 프로그래밍의 실행 결과를 한 번에 한 줄씩 출력한다.

또한, 제어카드를 통해 내가 짠 프로그래밍 언어가 C인지 등을 알려주었으며 요청한 일이 무엇인지도

알려주었다.

- 천공 카드 시스템은 실시간으로 결과를 주는 것이 아니기 때문에 여러가지 맡긴 일을 효율적으로 처리

위한 작업 시스템이 발달하게 되었다.

• 일괄 처리(bath processing) 시스템

- 일괄 처리 시스템은 모든 작업을 한꺼번에 처리해야 하고 프로그램 실행 중간에 사용자가 데이터를

입력하거나 수정하는 것이 불가능해야 한다. 즉, 사용자의 interaction 없이 한꺼번에 처리가 가능해야 한다.

• 상주 모니터(Resident Monitor)

- 자동 작업 순서에서 하나의 작업에서 다른 작업으로의 진행을 자동을 제어하는 프로그램

- 메모리가 시작하는 부분에 제어용 프로그램을 넣어서 사용자 프로그램 실행 시 자동으로 실행시키도록

하여 일괄 처리가 가능하도록 만들어준다.

4. 대화형 시스템(1960년대)

• 키보드와 모니터의 등장으로 프로그램이 진행되는 도중에 사용자로부터 입력을 받을 수 있어 입력 값에

따라 작업의 흐름을 바꾸는 것도 가능하게 되었으며 이와 같은 시스템을 대화형 시스템이라고 한다.

• 대화형 시스템의 등장으로 문서 편집기, 게임과 같은 다양한 종류의 응용 프로그램을 만들 수 있게

되었다. 즉, 사용자 편의를 위한 응용 프로그램이 많이 등장했다.(ex 문서편집기, 게임 등)

• 키보드, 모니터를 합쳐서 터미널이라고 불렸으며 큰 컴퓨터 시스템의 말단에 붙여 있었다.

5. 시분할 시스템(1960년대)

1) 다중 프로그래밍 시스템

• 배경: 한 번에 하나의 작업만 수행하는 것이 낭비라서 효율적으로 사용하기 위해 다중 프로그래밍

기술이 발달되었다.

• 다중 프로그래밍은 하나의 CPU로 여러 작업을 동시에 실행하는 기술로 CPU가 항상 수행할 작업을

가지도록 하여 프로세서 이용률을 극대화하려는 방법이다.

• 폰 노이만 구조를 보면 실행될 프로그램이 메인 메모리에 적재되어 있는 구조로, A 프로그램을

실행하려고 하면 A 프로그램이 main memory에 들어가 있어야 한다. 그러나 문제점이 존재한다.

프로그램을 실행하다가 입출력해야 하는 상황이 왔을 때 CPU는 유휴 상태가 되어 버린다.

그래서 해결책으로는 여러 프로그램을 기억 장치에 적재하여, CPU가 다른 프로그램 작업을 수행

할 수 있도록 해주어야 한다.

• 유휴 상태란? 컴퓨터 시스템이 정상적으로 사용 가능한 상태이지만 계산 처리 시간과 입출력 처리

시간 등의 차이로 실제적으로 작업이 없는 상태이다.

• 이를 구현하기 위해서 몇 개의 프로그램까지 적재할 것인지를 관리해주는 기억 장치 관리 기법과

CPU를 어떤 순서로 사용할 것인지를 결정하는 CPU 스케줄링 기법이 필요하다

• 다중 프로그래밍은 어떤 하나의 프로그램의 실행 성능을 향상시키지는 못하지만 전반적인 시스템

성능을 향상한다.

2) 시분할 시스템(time-sharing)

• 배경: 사용자 여러 명이 터미널을 통해 사용하게 되면서 사용자들이 컴퓨터 시스템으로부터 서비스를

제대로 받고 있다는 것이 중요하게 여기게 되면서 나온 개념이다.

• CPU 사용 시간을 잘게 쪼개어 작업들에 나누어 줌으로써 작업이 동시에 이루어지는 것처럼 보이게 하는

것이다.

• 잘게 나뉜 시간 한 조각을 타임 슬라이스(time slice) 또는 타임 퀀텀(time quantum)이라고 한다.

• 이 시스템의 장점은 번갈아 가면서 프로그램을 실행하게 되면서 사용자는 응답 시간이 빠르다고

느낄 것이며, 자신만이 컴퓨터를 독점하여 사용하는 것처럼 느낄 것이다.

• 이 시스템의 단점은 여러 작업을 동시에 처리하기 위한 추가 작업이 필요하다는 점과 시스템 내에

많은 양의 작업이 공존할 경우, 중요한 작업이 일정시간 내에 끝낼 수 없다는 점이다. 이로 인해

새로 생긴 시스템은 실시간 시스템(real-time system)이다.

• 시분할 시스템은 여러 개의 프로그램이 메모리에 적재되어 있다는 전재로 나온 개념이라서 1960년대에

여러 개의 시스템을 사용하는 의미에서 다중모드 시스템이라는 용어가 나옴

3) 실시간 시스템(Real-time system)

• 처리를 요구하는 자료가 발생할 때마 즉시 처리하여 제한 시간안에 응답하는 방식이다.

• 특정 시스템을 일정 시간 안에 작업을 처리할 수 있는 장점이 있다.

• 은행, 기차와 비행기의 좌석 예약 등 특수목적 뿐만 아니라 로봇 제어 및 가상 현실 등의 응용 분야에

제어 장치로 사용

• 특징

- 자료가 발생한 지점에서 바로 입출력이 되는 형태

- 자원을 효율적으로 사용하는 것보다 신속하게 대답하는 것이 중요

- 자료가 무작위로 도착하기 때문에 일시 저장과 대기가 필요

- 실시간 시스템은 특정한 상황(비행기 관재 상태나 공장 제어 상태)에서 동작하기 때문에 특정한

상황이 지나면 재현이 불가능하다.

- 시스템 장애 시 단순한 재실행 불가

6. 분산 시스템(1970년대)

• 개인용 컴퓨터와 인터넷이 보급되면서 값이 싸고 크기가 작은 컴퓨터들을 하나로 묶어 대형 컴퓨터에

버금가는 시스템

• 분산 시스템은 네트워크상에 분산되어 있는 여러 컴퓨터로 작업을 처리하고 그 결과를 상호 교환하도록

구성한 시스템

• 70년대 후반까지 터미널을 통해 사용자들이 메인 프레임을 사용했다면 70년대 후반에는 PC가 등장하여

완전히 다른 운영체제 개념이 등장

7. 클라이언트/서버 시스템(1990년대~현재)

• 작업을 요청하는 클라이언트와 거기에 응답하여 요청 받은 작업을 처리하는 서버의 이중 구조로 나뉜다.

• 서비스를 이용 받은 쪽은 클라이언트이고 서비스를 제공하는 쪽이 서버이며, 컴퓨터 상에서는 클라이언트

/서버 관계가 유동적으로 일어날 수 있다.

• 클라이언트/서버의 문제점은 서버의 과부화가 생길 수 있다는 점이다. 그 이유는 모든 요청이 서버로

집중되기 때문에 수많은 클라이언트를 처리하기 위해서는 많은 서버와 큰 용량의 네트워크가 필요하다.

8. P2P 시스템(2000년대 초반~현재)

• 배경: 서버의 부하를 줄일 수 있도록 나온 시스템이 P2P 시스템이다.

• P2P 시스템은 사용자와 사용자를 직접 연결하여 서비스가 일어난다.

• 1999년에는 냅스터(mp3 공유 시스템)을 시작으로 하여 메신저나 토렌트(영화파일 조각으로 주어서 다른

사람이 영화를 볼 수 있도록 해줌) 시스템에서 사용했지만 저작권 문제 및 불법 소프트웨어 기술 규제로

인해 발전하지 못하게 된다.

• 그러나 메신저 프로그램(내 글을 P2P 시스템으로 옮기는 것)에 P2P 시스템을 적용하게 되면서 큰 발전을 이룬다.

9. 그리드 컴퓨팅(grid computing)

• 필요한 기간만큼만 컴퓨터를 사용하고 사용한 금액만큼만 돈을 지불할 수 있는 컴퓨팅 환경으로 그리드 형태로 연결하여 필요한 만큼만 사용할 수 있도록 해준다.

• 그리드 컴퓨팅 기술이 발전해서 클라우드 컴퓨팅으로 발전하게 된다. 또한 그리드 컴퓨팅은 클라우드

컴퓨팅의 하드웨어적인 버전이다.

10. 클라우드 컴퓨팅(cloud computing)

• 클라우드 컴퓨팅은 언제 어디서나 응용 프로그램과 데이터를 자유롭게 사용할 수 있는 컴퓨팅 환경으로

그리드 컴퓨팅과 SaaS를 합쳐 놓은 형태이다.

• 하드웨어를 포함한 시스템이 구름에 가려진 것처럼 사용자에게 보이지 않는 컴퓨팅 환경이라는 의미이다.

• 클라우드 컴퓨팅은 PC, 휴대전화, 스마트 기기 등을 통해 인터넷에 접속하고, 다양한 작업을 수행하며,

기기 간의 데이터 이동이 자유로운 컴퓨팅 환경이다.

• 클라우드 컴퓨팅에서는 Iaas(infrastructure as a service), Paas(platform as a service),

Saas(software as a service) 서비스가 있다. Iaas는 ‘서비스로서의 인프라’라는 뜻으로 서비스 제공자가

컴퓨팅 리소스를 네트워크를 토해 서비스르 제공하는 모델로 클라우드 컴퓨팅에서 많이 사용하는 형태이다.(ex 스토리지 – 데이터 저장장치) Paas는 개발환경, 개발 언어 등을 제공하는 서비스로 제공하는 모델입니다. Saas는 서비스 제공자가 사용자에게 소프트웨어 서비스를 제공하는 모델입니다. 대표적으로 G-mail은 설치해서 사용하는 것이 아니라 구글 뒤에 가려진 소프트웨어 사용하고 있다.

11. 사물인터넷(Internet of Things)

• 사물에 센서와 통신 기능을 내장하여 인터넷에 연결하는 기술

• 사물인터넷은 사람이 사용하는 컴퓨터들이 연결된 망에서 사물까지 확장된 개념이다.

• 사물이 인터넷에 연결되면서 스마트폰으로 전자제품들을 제어하는 것이 가능하며, 사물에서도 막대한

데이터를 모으고 분석하는 등 유용한 정보로 제공이 가능하게 되었다.

(3) 운영체제의 구조

1. 커널과 인터페이스

1) 커널

- 프로세스 관리, 메모리 관리, 저장장치 관리와 같은 운영체제의 핵심적인 기능을 모아놓은 것이다.

2) 인터페이스

- 사용자와 응용 프로그램에 인접하여 커널에 명령을 전달하고 실행 결과를 사용자와 응용 프로그램에

돌려주는 역할을 한다.

2. 시스템 호출과 드라이버 디바이스

1) 시스템 호출(system call)

- 커널이 자신을 보호하기 위해 만든 인터페이스로 응용프로그램이 커널에 접근할 수 있도록 해준다.

즉, 커널에 접근할 수 있도록 함수를 제공해준다. (ex) printf(), write(), read() )

- 운영체제는 사용자나 응용 프로그램이 하드웨어에 직접 접근하지 못하도록 막음으로써 컴퓨터 자원을

보호한다. 그리고 대신 하드웨어와 같은 시스템 자원을 사용할 수 있도록 인터페이스를 제공하는 데

그것이 바로 시스템 호출이다. 그래서 커널은 시스템 호출을 통해서 들어오는 것만 접근을 허용한다.

2) 드라이버(driver)

- 드라이버란? 커널과 하드웨어의 인터페이스

- 하드웨어의 특성을 반영한 소프트웨어를 디바이스 드라이버라고 부르며, 운영체제가 많은 하드웨어를

사용할 수 있는 환경을 제공하는 프로그램이다.

- 마우스와 같이 간단한 제품은 드라이버를 커널이 가지고 있으나, 그래픽카드와 같이 복잡한 하드웨어의

경우 제작자가 드라이버를 제공함.

-> 드라이버는 커널 전체를 감싸고 있지 않으며 커널이 제공하는 드라이버도 있고 하드웨어 제작자가

제공하는 드라이버도 있다는 뜻이다.

- Plug&Play: 마이크를 연결해도 자동으로 인식되어서 사용할 수 있도록 해주는 등등 사용자가 컴퓨터에 새로운 장치를 연결/제거 하였을 때 컴퓨터를 재부팅 하면서 플러그 앤 플레이 바이오스가 자동으로 새로운 장치를 감지하여 필요한 환경 값을 설정해 주는 기능입니다. 때문에 사용자가 장치를 변경하는데 보다 손쉽게 컴퓨터를 작동하게 해줍니다.

3. 커널의 구성

1) 단일형 구조 커널(Monolithic architecture)

- 단일형 구조 커널은 초창기의 운영체제 구조 -> 하나에 모든 커널 기능을 넣음

- 커널의 핵심 기능을 구현하는 모듈들이 구분 없이 하나로 구성

- 장점

• 각 기능들을 하는 모듈들이 특별히 통신할 필요가 없어서 효율적인 운영이 가능

- 단점

• 버그나 오류를 처리하기 어려움

• 운영체제의 여러 기능이 서로 연결되어 있어 상호의존성이 높다. 그래서 기능상의 작은 결함이

시스템 전체로 확산될 수 있다.

• 다양한 환경의 시스템에 적용하기 어려움

• 현대의 운영체제는 매우 크고 복잡하기 때문에 완전 단일형 구조의 운영체제를 구현하기가 어렵다.

2) 계층형 구조 커널(layered architecture)

- 비슷한 기능을 가진 모듈을 묶어서 하나의 계층으로 만들고 계층 간의 통신을 통해 운영체제를

구현하는 방식

- 비슷한 기능을 모아 모듈화했기 때문에 단일형 구조보다 버그나 오류를 쉽게 처리할 수 있다. 또한,

오류 발생시 해당 계층만 따로 수정하면 되기 때문에 디버깅(debugging)하기도 쉽다.

- 기능 별로 모듈화해서 계층 간의 통신이 필요하기 때문에 계층 구분이 어렵다.

- 한 모듈에 나온 오류가 다른 곳에 영향을 주지 않으며 구분해서 구현하기 때문에 구현이 단순하다.

3) 마이크로 구조 커널(micro architecture)

- 마이크로 구조 커널의 운영체제는 프로세스 관리, 메모리 관리, 프로세스 간 통신 관리 등

가장 기본적인 기능만 제공하며 다른 구조와 달리 운영체제의 많은 부분이 사용자 영역 구현

- 각 모듈을 세분화되어 존재하고 모듈 간의 정보 교환은 프로세스 간 통신을 통해 이루어진다.

- 마이크로 구조 커널의 장점은 각 모듈이 독립적으로 작동하기 때문에 한 모듈에서 오류나도

전체 운영체제가 멈추지 않는다. 커널 자체가 작으므로 지연이 적고 예측이 가능하여 실시간

시스템에 활용된다. 그러나 단점으로는 모듈이 많기 때문에 빈번하게 통신이 발생해 성능이

저하된다는 문제가 있다.

4) 가상머신

- 배경: 각각 운영체제마다 호환이 되는 언어가 다르며, 유닉스의 버전이 다양해지면서 유닉스

운영체제마다 호환되지 않는 경우들이 생기다 보니, 개발자들은 응용프로그램 만들 때

각각 운영체제의 호환에 맞게끔 해주어야 한다. 그래서 이와 같은 문제점을 해결하고자

운영체제 위에 가상머신을 만들고 그 위에 응용프로그램 작동하게 만들어 호환성의 문제를

해결을 하였다.

- 가상머신은 운영체제와 응용 프로그램 사이에서 작동하는 프로그램으로, 가상머신을 설치하면

응용 프로그램이 모두 동일한 환경에서 작동하는 것처럼 보인다.

- 가상머신은 플랫폼은 다르지만, 응용프로그램은 똑같이 돌 수 있게끔 해주어 운영체제 종류에

상관없이 서비스를 제공받을 수 있게 되었다.

- 서버 가상화: 물리적 서버 하나에 가상 서버를 여러 개 구성하는 방법

-> 가상머신이라는 개념을 이용해 한 개의 하드웨어서 여러 개로 나누어 사용할 수 있도록 해준다.

- 가상 머신을 사용하면 호환성이 높아지지만 응용 프로그램이 가상머신을 통해서 작동하기 때문에 느려진다는 단점이 있다.

(4). 심화학습

1. 유닉스

- 개인용 PC 나오기 전 mult-frame에서 동작하는 운영체제로 사용자가 여러 명 연결되어 있는 형태이다.

- 대부분 C언어로 되어 있으면 개발 후 소스코드를 공개되어 기능이 추가되었고 이식하기 쉬었던 탓에

인기를 얻게 된다.

2. 리눅스

- 유닉스를 작은 컴퓨터에서 사용할 수 있도록 만든 시스템이다.

- 유닉스의 많은 기능을 PC용으로 만들었다. / 유닉스 계열의 운영체제 / 초기 설치가 어려움

3. 안드로이드

- GNU의 리눅스 커널을 사용하여 제작

- 공짜로 사용할 수 있으며 누구나 수정,배포할 수 있다. 다만 이득을 얻기 위해 사용할 수는 없다.

Chapter2) 컴퓨터의 구조와 성능 향상

(1) 컴퓨터의 기본 구성

1. 하드웨어의 구성

1) 컴퓨터는 중앙처리장치(CPU), 메인메모리, 입력장치, 출력장치, 저장장치로 구성

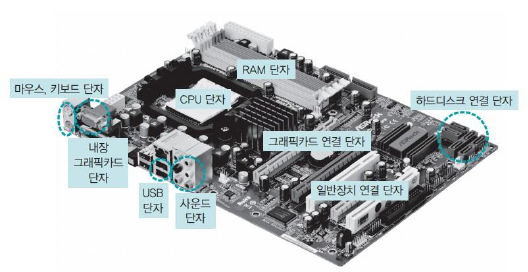
2) 컴퓨터로 하는 작업은 대부분은 중앙처리장치와 메인메모리이기 때문에 필수장치로 분류되며,

입력장치, 출력장치, 저장장치는 주변장치로 분류된다.

3) 메인보드

- 메인보드는 CPU와 메모리 등 다양한 부품을 연결하는 커다란 판

- 메인보드는 버스가 복잡하게 얽혀 있으며 전력이 공급되면 버스로 연결된 부품이 작동한다.

 - 버스는 각 장치를 연결하는 선의 집합이자 데이터가 지나다니는 통로이다.

-> 연결단자로 되어 있는 것이 주변 장치이다.

2. 하드웨어 사양 관련 용어

1) 클록

- CPU내에 있는 시계(속도를 정해줄 시계)로 CPU의 속도와 관련된 단위

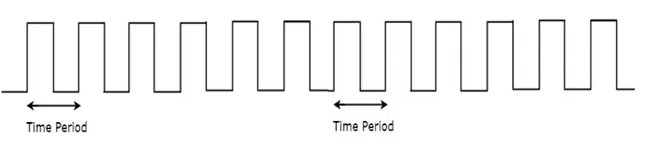
- 클록이 일정간격으로 틱(tick 혹은 펄스(pulse), 클록틱(clock tick)이라고 불린다.)을 만들면 거기에 맞추어 CPU 안의 모든 구성 부품이 작업을 한다.

2) 헤르츠

- 클록틱이 발생하는 속도를 나타내는 단위

- 1초 클록틱이 한 번이면 1Hz, 1000번이면 1kHz

- ‘인텔 코어 i7’을 보면 3.4GHz라고 나와 있는 데 이때 1초에 클록틱이 번으로 약 34억 번의 연산작업을 할 수 있다.

 - ‘메인보드’을 보면 1333MHz(약 1.3GHz)의 클록을 사용하는 데 초당 1.3억 번의 데이터 이동이 가능하다.

- 1초 10개 클록틱이 있으면 10Hz일 때 하나의 펄스 주기는 1/10초이다

- time period는 하나의 펄스 주기이며 프로세서가 일하는 단위시간이다.

3) 시스템 버스와 CPU 내부 버스

- 시스템 버스(System Bus)는 메모리와 주변장치를 연결하는 버스로 FSB(Front Side Bus, 전면버스)라고

한다.

- CPU 내부 버스는 CPU 내부에 있는 장치를 연결하는 버스로 BSB(Back Side Bus, 후면버스)라고 한다.

- CPU는 BSB(CPU 내부 버스)의 속도로 작동하고, 메모리는 FSB(시스템 버스)의 속도로 작동한다.

 - 두 버스의 속도로 차이(후면버스의 속도 > 전면버스의 속도)로 인한 문제는 캐쉬(cache)로 해결

- CPU부분을 보면 4코어로, 4개의 프로세서가 있다.

3.4GHz를 클럭 속도(CPU 내부 버스 속력과 CPU의 클록 속도가 같음)이며, 4MB는 캐시 메모리이다.

- 메모리를 보면 1,333MHz의 속도를 가지고 있어 FSB의 속도와 동일하다. (그 이유는 FSB 속도보다 낮으면 메인보드가 그 해당 속도에 맞추어서 작동해야 하기 때문에 컴퓨터 성능이 안 좋아지는 결과가 나온다.)

- 요새 속도를 더블로 하는 DDR3를 사용한다. 4GB는 메모리 용량이다.

- 그래픽 카드를 보면 그래픽을 처리를 위한 메모리가 1GB가 있다.

- 하드디스크를 보면 1TB는 용량이며, 7200rpm(1분에 7200을 돌게 된다)은 회전속도이다.

32MB는 하드디스크를 접근을 향상시키기 위해 사용하는 캐시 메모리이다.

- 광학디스크를 보면 DVD의 22X, CD-R의 48X는 속도를 나타낸 것이며, CD-R은 Recording을 할 수 있는 것

으로, recording 작업을 한 번만 할 수 있다. 반면 CD-RW은 여러 번 사용이 가능하다.

(2) CPU와 메모리

1. CPU의 구성과 동작

1) 산술논리 연산장치(Arithmetic and Logic Unit, ALU)

- 산술논리 연산장치는 데이터의 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 산술 연산과 AND, OR과 같은 논리 연산 수행

2) 제어장치(Control Unit, CU)

- CU에서 작업을 지시하는 부분으로 관재센터 역할을 한다. 즉 작업을 지시하고 명령어를 읽어와서 해독한다.

- 컴퓨터 내에 일어나는 일을 총괄하는 곳

3) 레지스터(Register)

- CPU 내에 데이터를 임시로 보관하는 곳

-ALU 동작이나 CU 동작에 필요한 자료를 임시 보관하는 곳

2. CPU의 명령어 처리 과정

- C언어로 작성한 덧셈 프로그램 어셈블리어로 변환된 덧셈 프로그램

int D2 = 2, D3 = 3, sum; -> LOAD mem(100), register 2;

메모리 100번에 있는 데이터를 레지스터 2에 옮겨라!

sum = D2 + D3; LOAD mem(120), register 3;

메모리 120번에 있는 데이터를 레지스터 3에 옮겨라!

ADD register 5, register 2, register3;

레지스터 2와 레지스터 3에 있는 데이터를 가지고 ADD 연산하고 결과값은 레지스터 5에 저장

MOVE register 5, mem(160);

레지스터 5에 있는 값은 메모리 160번에 저장

-> LOAD랑 MOVE는 버스를 통해 데이터를 이동하며, LOAD는 하위레벨에서 상위레벨로 데이터를 옯긴다.

-> CPU는 어셈블리어의 명령 한 줄을 처리하기 위해서는 FI(Fetch Instruction / 명령어 인출 작업) -> DA (명령어를 decode하고 유효 Address를 계산) -> FO(Fetch Operand / Operand(피연산자 함수)를 메모리로부터 Fetch(가지고 옴)함) -> EX (실행) 이와 같은 과정을 가진다.

-> ADD register 5, register 2, register3은 ALU에서 일어나는 일

3. 레지스터의 종류

1) 사용자 가시 레지스터

- 데이터 레지스터(DR, Data Register)

• 데이터 레지스터는 CPU가 명령어를 처리하는 데 필요한 일반 데이터를 메모리에서 가져와 임시로 보관

• CPU에 있는 대부분의 레지스터가 데이터 레지스터이기 때문에 범용 레지스터이다.

- 주소 레지스터(AR, Address Register)

• 데이터 또는 명령어가 저장된 메모리 주소를 저장

2) 사용자 불가시 레지스터

- 프로그램 카운터(PC, Program Counter)

• CPU는 다음에 어떤 명령어를 처리해야 할지 알아야 한다. 그래서 프로그램 카운터 레지스터에 다음에 실행할 명령어의 주소(위치)를 저장하고 있다가 제어장치에게 알려준다.

- 메모리 주소 레지스터(MAR, Memory Address Register)

• 메모리 주소 레지스터는 메모리 관리자가 접근 해야 할 메모리의 주소를 저장

• 메모리 주소 레지스터는 메모리에서 데이터를 가져오거나 반대로 메모리로 데이터를 보낼 때 주소를

저장하여 사용한다.

- 메모리 버퍼 레지스터(MBR, Memory Buffer Register)

• 메모리 관리자가 메모리에서 가져온 데이터를 임시로 저장

• 메모리에서 가져온 데이터나 메모리로 옮겨 갈 데이터를 임시로 저장

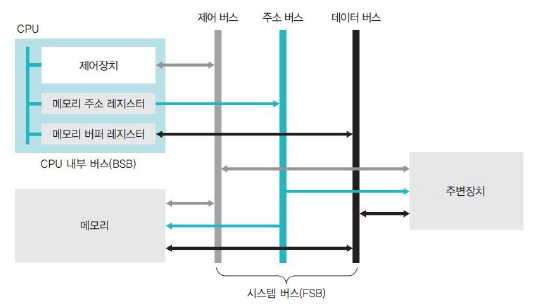
- 프로그램 상태 레지스터(PSR, Program Status Register)

• 연산 결과(양수, 음수 등)을 저장한다.

• CPU내 상태들을 전부 저장한다.

• flag register로 연산결과가 양수, 음수, 0인 지를 1비트씩 할당하여 flag처럼 저장

4. 버스의 종류

- 제어버스(Control Bus 제어장치가 연결된 버스)

• CPU가 메모리와 주변장치에 제어 신호를 보내기 위해 사용

• 메모리에서 데이터를 가져올 때 읽기 신호를 보내고, 처리한 데이터를 메모리로 옮겨 놓을 때는 쓰기 신호를 보낸다.(CPU->메모리/ read, write 신호는 CPU에서 메모리로 감) 주변 장치도 마찬가지로 하드디스크에 저장 명령을 내리거나 사운드 카드에 소리를 내라는 명령을 내릴 때 제어 버스를 통해 전달(CPU->주변장치)

• 주변장치가 CPU한테 일하게 해달라고 제어신호를 보내면 CPU는 항상 받기만 한다.(주변장치->CPU)

• 하나씩 살피면 단방향이지만 전체적으로 살피면 양방향이다.

- 주소 버스(Address Bus)

• 주소 버스는 메모리의 데이터를 읽거나 쓸 때 어느 위치에서 작업할 것인지를 알려주는 위치 정보가 오간다.(메모리나 주변장치에 데이터를 읽거나 쓸 때 위치 정보를 보내기 위해 사용) -> 단방향

• CPU에서 메모리나 주변장치로 나가는 주소 정보는 있지만 주소 버스를 통해 CPU로 전달되는 경우는 없다.

- 데이터 버스(Data Bus)

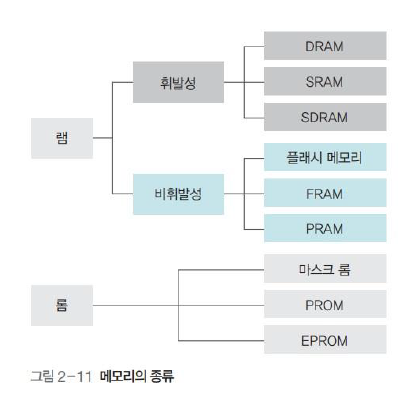
• 메모리 버퍼 레지스터와 연결된 버스로, 데이터의 이동이 양방향으로 이루어진다.

- 버스의 대역폭(bandwidth)은 한 번에 전달할 수 있는 데이터의 최대 크기이며, 한 번에 처리할 수 있는

데이터의 크기와 같다. 흔히 32bit CPU, 64bit CPU라고 하는데 32bit, 64bit는 CPU가 한 번에 처리할 수

있는 데이터의 최대 크기이다. 참고로 CPU가 한 번에 처리할 수 있는 데이터의 최대 크기를 워드(word)

라고 하며, 32bit CPU에서 1워드는 32bit이다.

 5. 메모리의 종류

- 램(RAM)과 롬(ROM)의 구분은 읽을 수만 있는지 읽고 쓸 수 있는 지로 구분

- 휘발성 메모리->전원이 끄면 다 날아간다.

1) DRAM

• 저장된 0과 1의 데이터가 일정 시간이 지나면 사라지므로 일정 시간

마다 다시 재생시켜주어야 한다.

• 전기가 차면 전류가 흐르고 다 빠지면 안 흐르는 방식으로 0,1 표시

하는 방법을 사용

• 시간이 지나면 방전이 되므로, 그렇게 되면 원하지 않는 데이터가

저장이 된다. 그래서 재충전이 필요하다.

2) SRAM

• 전력을 공급되는 동안에 데이터를 보관할 수 있어 재생할 필요가 없다.

• FF(Flip Flop)는 1bit를 저장이 가능한 storage로, SRAM는 FF을 이용해서 만들기 때문에 전원을 유지하고 있으면, 정보가 그대로 유지된다.

3) SDRAM

• 클록틱(펄스)이 발생할 때마다 데이터를 저장하는 동기 DRAM

• DRAM의 한 종류

- 비휘발성 메모리

1) 플래시 메모리(Flash Memory)

• 전력이 없어도 데이터를 보관하는 저장장치

• 반도체 성질을 가진 메모리이면서 읽고 쓰기가 된다.

2) SSD(Solid State Memory)

• 가격은 비싸지만 빠른 데이터 접근 속도, 저전력, 내구성 때문에 개인용 컴퓨터는 물론 노트북, 스마트폰 등 많은 기기에 사용

• 반도체 성질을 가지고 있으며, 일반 PC에 넣으면 하드디스크보다 빨리 부팅이 된다.

- 룸의 종류

1) 마스크 룸(mask ROM) -> CD와 유사

• 데이터를 지우거나 쓸 수 없음

2) PROM(Programmable ROM) -> CD-R과 유사

• 전용 기계를 이용하여 데이터를 한 번만 저장할 수 있다.

3) EPROM(Erasable Programmable ROM) -> CD-RW과 유사

• 전용 기계를 이용하여 데이터를 여러 번 쓰고 지울 수 있음.

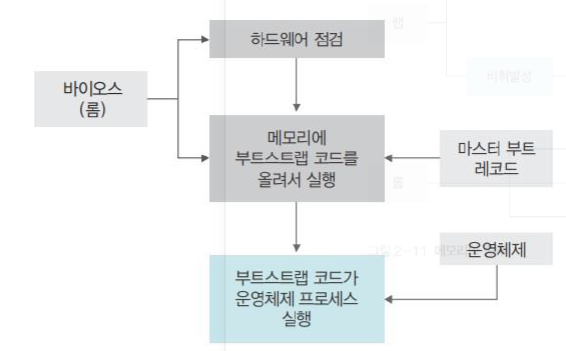
6. 메모리 보호의 필요성

1) 현대의 운영체제는 시분할 기법을 사용하여 여러 프로그램을 동시에 실행하므로, 사용자 영역이 여러 개의 작업 공간으로 나뉘어 있다.

2) 메모리가 보호되지 않으면 어떤 작업이 다른 작업의 영역을 침범하여 프로그램을 파괴하거나 데이터를 지울 수도 있으며, 최악의 경우 운영체제 영역을 침범하면 시스템이 멈출 수도 있다. 이처럼 다른 프로그램의 영역을 침범하려는 악성 소프트웨어를 바이러스라고 한다.

3) 현대의 컴퓨터에서는 시분할 방식을 사용하기 때문에 다른 소프트웨어가 CPU를 차지하는 동안에 운영체제는 작업이 중단된 상태가 된다. 그러므로 운영체제는 하드웨어의 도움을 받아 메모리를 보호해야 한다.

4) 메모리 보호를 위해 CPU는 현재 진행 중인 작업의 메모리 시작 주소를 경계 레지스터(Bound register)에 저장한 후 작업을 한다. 또한 현재 진행 중인 작업이 차지하고 있는 메모리의 크기, 즉 마지막 주소까지의 차이를 한계 레지스터(limit register)에 저장한다. 작업을 진행하는 동안 두 레지스터의 주소 범위를 벗어나는지 하드웨어적으로 점검함으로써 메모리를 보호하게 된다.

 7. 부팅

1) 바이오스(롬)

• CPU, 메모리, 하드디스크, 키보드, 마우스와 같은 주요 하드웨어가 제대로 작동하는지 확인한다.

• 컴퓨터를 꺼도 남아있으며, 하드웨어 점검하고 운영체제를 메인메모리로 로드 시켜 주기 위한 코드인 부트스트랩(Bootstrap) 코드를 실행시킨다.

(3) 컴퓨터 성능 향상 기술

1. 버퍼

1) 버퍼의 개념

• 속도의 차이와 상관없이 데이터가 날라가지 않고 저장할 수 있도록 해주는 임시저장소

• 속도에 차이가 있는 두 장치 사이에서 그 차이를 완화하는 역할을 하는 장치

• 일정량의 데이터를 모아 옮김으로써 속도의 차이를 완화

2) 스풀링

• 스풀은 CPU와 입출력장치가 독립적으로 동작하도록 고안된 소프트웨어적인 버퍼

• 스풀러는 인쇄할 내용을 순차적으로 출력하는 소프트웨어로 출력 명령을 내린 프로그램과 독립적으로

동작한다.

• 스풀링은 프린터 안에 디스크가 있다고 생각하면 된다. 그래서 프린트가 다 되기 전에 문서를 꺼도

출력이 된다.

• 버퍼는 데이터든 버퍼가 차면 이동이 시작되는 반면 스풀러는 인쇄물이 완료될 때까지 다른 인쇄물이 끼어들 수 없다.

2. 캐시

1) 캐시(cache)

- 캐시는 메모리와 CPU 간의 속도 차이(BSB와 FSB의 속도 차이)를 완화하기 위해 메모리의 데이터를 미리 가져와 저장해두는 임시 저장소

- 미리 가져오기(prefetch): 캐시는 필요한 데이터를 모아 한꺼번에 전달하는 버퍼의 일종으로 CPU가 앞으로 사용할 것으로 예상되는 데이터를 미리 가져다 놓는다.

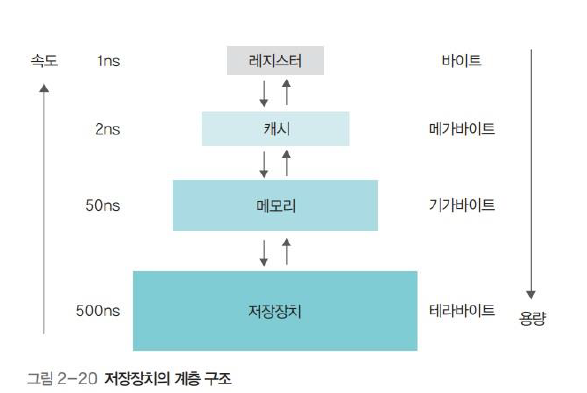
- 캐시 히트(cache hit): 캐시에서 원하는 데이터를 찾는 것

- 캐시 미스(cache miss): 원하는 데이터가 캐시에 없는 것

- 캐시 적중률(cache hit ratio): 캐시 히트가 되는 비율 -> 일반적인 컴퓨터의 캐시 적중률은 90%이다.

- 캐시 적중률이 높은 이유는 참조 지역성(locality) 때문이다. 참조 지역성은 현재 위치에 가까운 데이터가 멀리 있는 데이터보다 사용될 확률이 높다는 것이다. 즉, 메모리에서 자료를 참조하려고 할 때 일정 지역 혹은 일정 시간 내에 특정한 영역의 데이터들을 참조하려는 경향이다.

- 다단계 캐시: L1캐시와 L2캐시로 명령어 캐시나 데이터 캐시는 CPU 레지스터에 직접 연결되기 때문에 L1 캐시(level 1)캐시라고 부르며, 일반 캐시는 메모리와 연결되기 때문에 L2 캐시(level 2)캐시라고 부른다.

 2) 저장장치의 계층구조

- 가격과 컴퓨터 성능 사이의 타협점으로 저장장치의 계층 구조(storage hierarchy)가 존재

- 저장장치의 계층 구조는 속도가 빠르고 값이 비싼 저장장치를 CPU 가까운 쪽에 두고, 값이 싸고 용량이 큰 저장장치를 반대쪽에 배치하여 적당한 가격으로 빠른 속도와 큰 용량을 동시에 얻는 방법

- CPU와 가까운 쪽에 레지스터나 캐시를 배치하여 CPU가 작업을 빨리 진행할 수 있도록 해준다. 하드디스크와 같이

저렴하고 용량이 큰 저장장치에 영구적으로 저장할 수 있도록 해준다.

- 저장장치의 계층 구조는 사용자가 저렴한 가격으로 용량은 하드디스크처럼 사용하게 하고 작업 속도는 레지스터처럼 빠르도록 만들어준다.

- 저장장치의 계층 구조의 문제점으로는 데이터의 일관성 문제가 발생한다.

- 캐시는 주메모리에 비해 빠르고 양 적다. 그러나 메모리의 속도를 캐시 메모리에 가까울 수 있도록 메모리를 관리한다.

- 주메모리와 저장장치 간의 가상메모리라는 개념이 있다. 가상메모리(virtual memory)는 실제로 있지 않지만 있는 것처럼 보이게 하는 메모리로 하드디스크의 일부를 메모리로 사용하는 것이다.

3. 인터럽트(interrupt)

1) 폴링 방식(polling)

- 폴링 방식은 CPU가 주도권을 가지고 IO 장치들을 일일히 확인하는 방식이다.

- 폴링 방식에서는 CPU가 입출력장치의 상태를 주기적으로 검사하여 일정한 조건을 만족할 때 데이터를 처리한다.

- CPU가 명령어 해석과 실행이라는 본래 역할 외에 모든 입출력까지 관여하게 되므로 효율이 떨어진다.

2) 인터럽트 방식(interrupt)

- 입출력 관리자가 대신 입출력을 해주고 완료하면 CPU에게 완료신호를 보내어 처리하는 방식

-> IO장치가 일을 끝내면 CPU장치에게 interrupt를 거는 방식

- CPU의 작업과 저장장치의 데이터 이동을 독립적으로 운영함으로써 시스템의 효율이 높음

- 데이터의 입출력이 이루어지는 동안 CPU가 다른 작업을 할 수 있음

3) 인터럽트 번호(interrupt number)

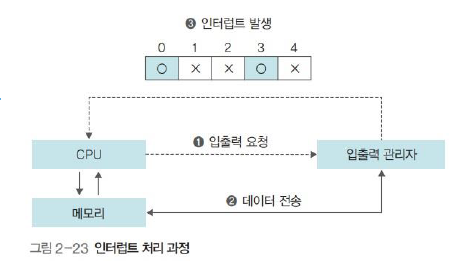
- 인터럽트 방식에서 많은 주변장치 중 어떤 것의 작업이 끝났는지를 CPU에게 알려주기 위해 사용한 번호

4) 인터럽트 벡터(interrupt vector)

- 여러 개의 인터럽트를 하나의 배열로 만든 것이다.

- 하나의 배열로 만들어 여러 개의 입출력 작업을 한꺼번에 처리한다.

5) 인터럽트 방식의 동작 과정

① CPU가 입출력 관리자에게 입출력 명령을 내림

② 입출력 관리자는 입출력 명령을 실행

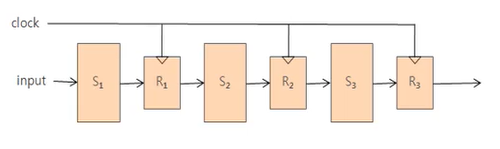
③ 입출력이 완료되면 입출력 관리자는 완료 신호를 CPU에게 보낸다.

- 다른 일을 하다가 완료 신호를 받으면 하던 일을 중단하고 옮겨진 데이터를 처리한다. 그래서 작업을 중단하고 처리해야 하는 신호라서 인터럽트라고 불린다.

4. 병렬 처리

1) 병렬 처리(parallel processing)

- 병렬 처리는 동시에 여러 개의 명령을 처리하여 작업의 능률을 올리는 방식

컴퓨터 구조 시간-파이프라이닝 구조(pipelining)

-> 하나의 일은 S1,S2,S3로 나눈 것이다. 이상적으로는 3,3,3으로 나눈 것이지만 현실적으로 그렇지 않다.

-> S1,S2,S3 사이에 있는 R은 레지스터로 S1,S2 사이의 속도 차이로 인해

생겨난 버퍼 공간이다. Buffer는 느린 세그먼트에 맞춰서 동작한다.

2) 병렬 처리 시 고려 사항

① 상호 의존성이 없어야 병렬 처리가 가능하다

- 각 명령이 서로 독립적이고 앞의 결과가 뒤의 명령에 영향을 미치 않아야 한다.

- 앞뒤 단계 간에 상호 의존적인 명령어를 사용하면 병렬 처리를 할 수 없다.

② 각 단계의 시간을 거의 일정하게 맞춰야 병렬 처리가 원만하게 이루어진다.

- 파이프라인에서 불균형 하게 나뉘면 병렬 처리 효과가 낮아진다.

- 단계별 시간의 차이가 크면 병렬 처리 효과가 떨어진다.

③ 전체 작업 시간을 몇 단계로 나눌지 잘 따져보아야 한다.

- 병렬 처리에서 작업을 N개로 쪼갰을 때 N을 병렬 처리의 깊이(depth of parallel processing)라고 한다.

- 이론적으로 N이 클수록 동시에 작업할 수 있는 작업의 개수 많아 성능이 높아지지만 작업을 너무 많이 나누면 각 단계마다 작업을 이동하고 새로운 작업을 불러오는 데 시간이 너무 많이 걸려서 오히려 성능이 떨어진다. 이러한 오버헤드를 고려하여 깊이를 10~20정도로 한다.

3) 병렬 처리 기법

① 명령어 패치(Instruction Fetch, IF): 다음에 실행할 명령어를 명령어 레지스터에 저장

② 명령어 해석(Instruction Decode, ID): 명령어 해석

③ 실행(Execution, EX): 해석한 결과를 토대로 명령어 실행

④ 쓰기(Write Back, WB): 실행된 결과를 메모리에 저장

-> 하나 처리하기 위해 4펄스(IF,ID,EX,WB)씩 사용

4) 파이프라인

- 파이프라인 기법(pipeline)은 CPU의 사용을 극대화하기 위해 명령을 겹쳐서 실행하는 방법으로, CPU의 사양과 연관 지어 보면 하나의 코어에 여러 개의 스레드를 사용하는 것이다.

- 파이프라인 위험

• 데이터 위험(Data hazard)

-> 데이터의 의존성 때문에 발생하는 문제

-> dumy라는 명령어를 사용하여 명령어 단계를 지연하여 해결한다. (but 좋은 방법X)

• 제어 위험(control hazard)

-> if문 혹은 goto문 같은 명령(Branch 명령) 때문에 발생하는 문제

-> 첫 명령어를 실행하고 보니 goto문이어서 다음 문장이 아니라 다른 문장으로 이동하게 되면 현재 동시에 처리되고 있는 명령어들이 쓸모가 없어진다. 제어 위험은 분기 예측이나 분기 지연 방법으로 해결된다.

• 구조 위험(structural hazard)

-> 서로 다른 명령어와 같은 자원에 접근하려 할 때 발생하는 문제(resource conflict, 자원충돌)

-> IF(명령어 패치)와 WE(쓰기)라는 서로 다른 명령어가 같은 자원 접근 시 충돌하게 된다. 해결책으로는 명령어를 사용하는 캐시와 데이터만 사용하는 캐시로 분리해서 사용한다.

5) 슈퍼스칼라 기법

• 슈퍼스칼라(super scalar)기법은 파이프라인을 처리할 수 있는 코어를 여러 개

구성하여 복수의명령어가 동시에 실행되도록 하는 방식이다.

• 파이프라인과 비슷하지만 코어를 2개 구성하여 각 단계에서 동시에

실행되는 명령어가 2개라는 점이 다르다.

6) VLIW 기법

• VLIW 기법은 CPU가 병렬 처리를 지원하지 않을 경우 소프트웨어적으로 병렬 처리를 하는 방법

• 명령어들을 컴파일러가 추출하고 하나의 명령어로 압축하여 실행한다.

• 명령어와 주소 구조를 여러 개 압축하여 소프트웨어적으로 한꺼번에 실행할 수 있도록 한다.

5. 심화학습

1) 무어의 법칙(Moore’s law)

• CPU의 속도가 24개월마다 2배 빨라진다는 법칙

• 반도체의 발열문제, 기술적인 한계로 실제로 적용되지 않는다.

• 무어의 법칙을 이용해 기술이 발달하게 됨

2) 암달의 법칙(Amdahl’s law)

• 일부를 개선하여도 전체 성능이 좋아지지 않는다.

• CPU의 속도를 2GHz에서 4GHz로 늘리더라도 컴퓨터의 성능이 2배 빨라지지 않음

Chapter3) 프로세스와 스레드

(1) 프로세스의 개요

1. 프로세스의 개념

1) 프로그램

• 저장장치에 저장되어 있는 정적인 상태

• 프로그램이란 어떤 데이터를 사용하여 어떤 작업을 할지 그 절차를 적어 놓은 것이다.

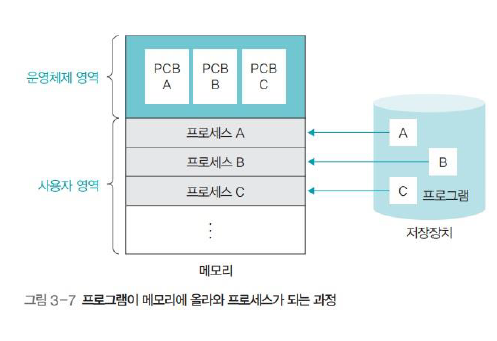
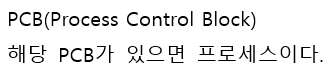
2) 프로세스

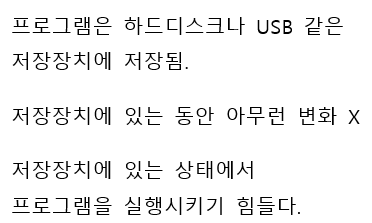
• 프로세스는 실행을 위해 메모리에 올라온 동적인 상태

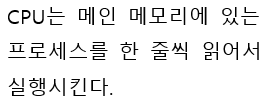
• 프로세스는 ‘실행한다’라고 표현하는 데, 이는 프로그램으로 작성된 작업 절차를 실제로 실행에 옮긴다.

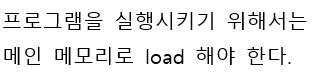
• 메모리에 적재되어 운영체제의 제어를 받는 상태

• 프로세스가 사용하는 메모리 영역이 존재하는 상태

 • 운영체제로부터 프로세스 제어 블록을 얻은 상태

 CPU





2. 프로세스 제어 블록(Process Control Block, PCB)이란?

1) 프로세스 제어 블록

• 운영체제가 해당 프로세스를 위해 관리하는 자료 구조

• 프로세스 제어 블록이 없으면 프로그램이 프로세스로 전환되지 못한다.

• 프로세스가 되었다는 것은 운영체제로부터 프로세스 제어 블록을 받았다는 의미이다.

2) 프로세스 제어 블록에 있는 다양한 정보 중 3가지

• 프로세스 구분자(Process IDentification, PID)

- 각 프로세스를 구분하는 구분자(IDentification, ID)

• 메모리 관련 정보

- 프로세스의 메모리 위치 정보-> CPU를 실행하기 위해서는 메모리의 위치가 어디에 저장되어 있는 지

알아야 작업을 할 수 있다.

• 각종 중간값

- 프로세스 제어 블록에는 프로세스가 사용했던 중간값을 저장

3) 프로그램 종료 시 PCB는 폐기된다. Register(레지스터): 특정 목적에 사용되는 일시적인

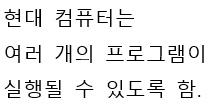
기억장치

PC CPU 메인 메모리 저장장치

(Program Counter)

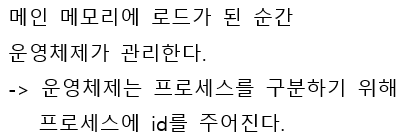
다음에 실행될 명령어의 주소 저장

기억장치로부터 읽혀 질 다음 명령어의

주소 저장

 IR

(Instruction Register)

실행될 명령어의 연산코드 저장

hello.exe 실행 시

PC: 500 그 후 PC: 501

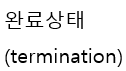
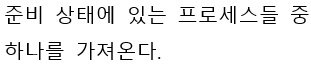
IR: 500줄에 있는 부분을 한 줄 읽다가

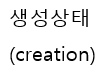
IR에 연산 코드를 넣어준다.

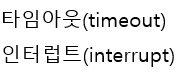
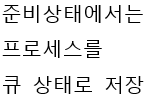
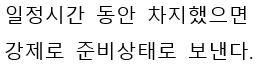
-> 이 때, hello.exe 실행이 끝나지 않고 test.exe 실행 시 번갈아 가면서 실행해야 하므로,

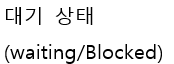
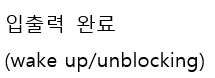
프로세스가 사용했던 중간 값들을 저장하는 자료구조가 PCB이다.

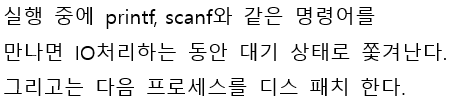
4. 프로세스의 상태

 1) 프로세스 상태 전이도









-> CPU는 한 개 프로그램을 실행하고 있지만 여러 프로그램들을 교체하여 실행하기 때문에 여러 프로그램

을 실행하는 것처럼 보인다.

2) 준비 상태(Ready)

• 실행 대기 중인 모든 프로세스가 자기 순서를 기다리는 상태

• 프로세스 제어 블록은 준비 큐(ready queue)에서 기다리며 CPU 스케줄러에 의해 관리된다.

-> CPU 스케줄러에 의해 어떤 프로세스가 먼저 서비스를 받게 되는 지가 결정된다.

• CPU 스케줄러가 어떤 프로세스 제어 블록을 선택하는 작업은 dispatch(PID) 명령으로 처리하는데,

Dispatch(PID)를 실행하면 해당 프로세스가 준비 상태에서 실행 상태로 바뀌어 작업이 이루어진다.

-> dispatch할 때 우선순위가 높은 프로세스에게 CPU를 할당해준다.

3) 실행 상태(Running)

• 실행 상태(running status)는 프로세스가 CPU를 할당 받아 실행되는 상태이다.

• 타임 슬라이스 동안만 작업을 할 수 있어 CPU를 오랫동안 사용하지 못한다.

• timeout(PID)이 실행되면 timeout(PID)는 프로세스 제어 블록을 실행 상태에서 준비 상태로 옮긴다.

• 실해 상태 동안 작업이 완료되면 exit(PID)가 실행되어 프로세스가 정상적으로 종료 가능하다.

-> Running 상태에서만 종료가 가능하다.

• 프로세스가 입출력 요청 시 block(PID)를 실행한다. block(PID)는 입출력이 완료될 때까지 작업을

진행할 수 없기 때문에 해당 프로세스를 대기 상태로 옮긴다.

• 비정상적으로 종료되는 강제 종료를 만나면 디버깅을 하기 위해 이전 상태를 저장장치로 옮기는 데

이를 코어 덤프(core dump)라고 한다.

4) 대기 상태(waiting/blocking)

• 실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태

• 입출력이 완료되면 인터럽트가 발생하고 대기 상태에 있는 여러 프로세스 중 해당 인터럽트로 깨어 날

프로세스를 찾는데 이것이 wake up(PID)이다. wake up(PID)로 해당 프로세스의 프로세스 제어 블록이

준비 상태로 이동하게 된다.

-> 준비 상태와 대기 상태에 있는 프로세스들은 각각 준비 리스트(ready list)와 대기 리스트(block list)로

운영체제에 의해 관리된다.

5) 보류 상태(suspended)

• 보류 상태는 프로세스가 메모리에서 잠시 쫓겨난 상태로 휴식 상태와 차이가 있다.

• 다음과 같은 상태 일 때, 보류 상태가 된다.

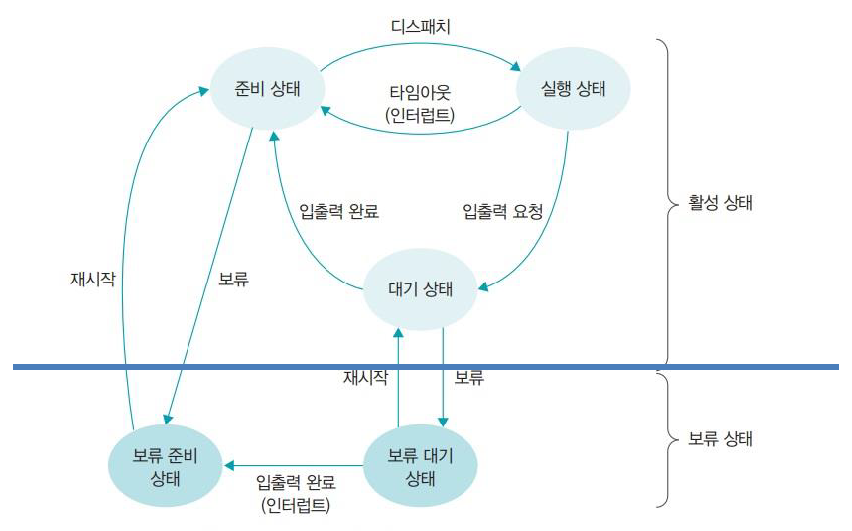
- 메모리가 꽉 차서 일부 프로세스를 메모리 밖으로 내보낼 때

- 프로그램 오류가 있어서 실행을 미루어야 할 때

- 바이러스와 같이 악의적인 공격을 하는 프로세스라고 판단할 때

- 매우 긴 주기로 반복되는 프로세스 일 때

- 입출력을 기다리는 프로세스의 입출력이 계속 지연될 때

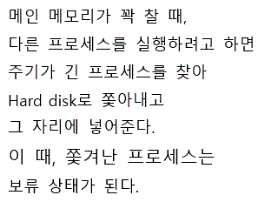


준비상태, 대기상태에서

모두 보류 상태로 쫓겨날

수 있다.







5. 프로세스 전이와 정리

1) 프로세스 전이

• 디스 패치(dispatch): 준비 상태 -> 실행 상태

- 준비 상태에 있는 프로세스들 중에서 우선 순위가 높은 프로세스를 선택하여 CPU를 할당

• 할당 시간 초과(timeout): 실행 상태 -> 준비 상태

- CPU의 지정된 할당 시간을 모두 사용한 프로세스는 다른 프로세스를 위해 다시 준비 상태로

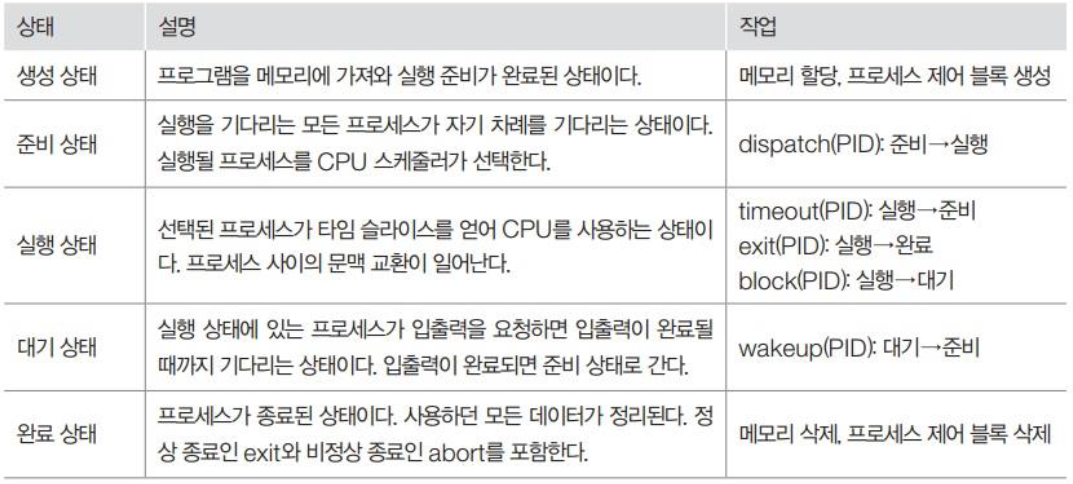
되돌아 감.

• 대기(block): 실행 상태 -> 대기 상태

- 실행 중인 프로세스가 입출력 명령을 만나면 대기 상태로 전환

• 깨움(wake up): 대기 상태 -> 준비 상태

- 입출력 완료신호가 들어오면 대기 중인 프로세스는 준비 상태로 전환

 2) 프로세스 정리







(2) 프로세스 제어 블록과 문맥 교환

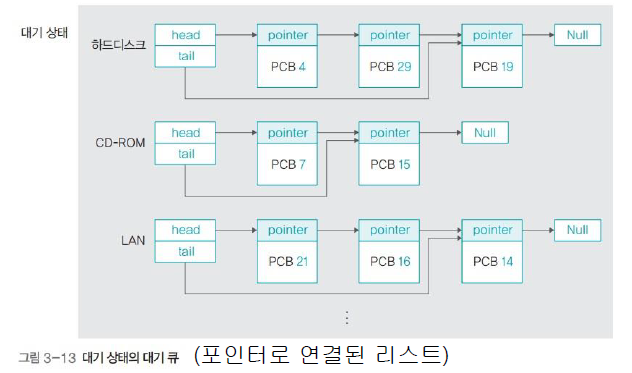
1. 프로세스 제어 블록(PCB)

1) 프로세스 제어 블록(PCB)

• 프로세스 제어 블록은 프로세스를 실행하는 데 필요한 중요한 정보를 보관하는 자료 구조

• 프로세스는 고유의 프로세스 제어 블록을 가짐

• 프로세스 생성 시 만들어져서 프로세스가 실행을 완료하면 폐기된다.

 2) 프로세스 제어 블록의 구성



3) 프로세스 제어 블록의 구성

• 프로세스 상태: 프로세스가 현재 어떤 상태에 있는 지를 나타냄

• 프로세스 구분자(PID): 운영체제 내에 있는 여러 프로세스를 구별하기 위한 구분자

• 프로그램 카운터: 다음에 실행될 명령의 위치를 가리키는 프로그램 카운터의 값을 저장

• 프로세스 우선순위: 프로세스의 실행 순서를 결정하는 우선순위

-> 사용자 프로세스보다 커널 프로세스는 우선순위가 높고, 사용자 프로세스끼리도 우선순위가 다름

-> 우선순위가 클수록 낮은 우선순위의 프로세스보다 먼저 실행되고 더 자주 실행된다.

• 각종 레지스터 정보: 프로세스 제어 블록에는 프로세스가 실행되는 중에 사용하던 레지스터

• 메모리 관리 정보: 프로세스가 메모리의 어디에 있는지 나타내는 메모리 위치 정보, 메모리 보호를

위해 사용하는 경계 레지스터 값과 한계 레스터 값 등이 저장

• 할당된 자원 정보: 입출력 자원이나 오픈 파일 등에 대한 정보

• 계정정보: 계정 번호, CPU 할당 시간, CPU 사용 시간 등

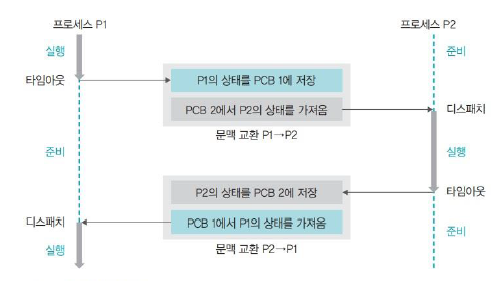
• 부모 프로세스 구분자와 자식 프로세스 구분자

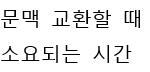
-> 프로세스는 단독으로 실행하지 않는다. 프로세스는 부모, 자식 관계에 대한 정보를 저장한다.

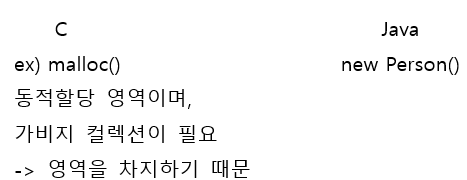
2. 문맥 교환(context switch)

• CPU를 차지하던 프로세스가 나가고 새로운 프로세스를 받아들이는 작업

• 실행 상태에서 나가는 프로세스 제어 블록에는 지금까지의 작업 내용을 저장하고, 반대로 실행 상태로

 들어오는 프로세스 제어 블록의 내용은 CPU가 다시 세팅한다.

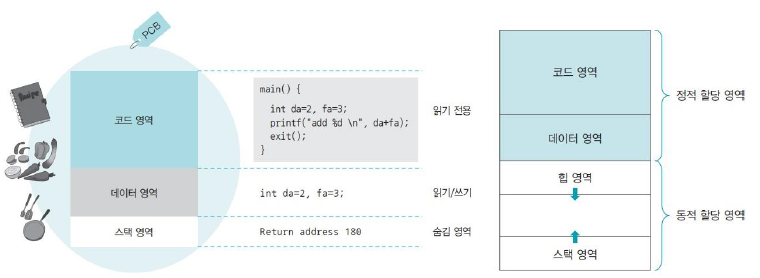


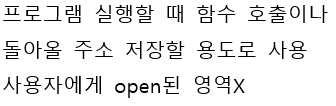




 -> 문맥교환이 많이 일어나는 게 좋지 않다.

(3) 프로세스의 연산

 1. 프로세스의 구조



1) 코드 영역 code area

• 프로그램의 본문이 기술된 곳

• 프로그래머가 작성한 코드가 탑재되며 탑재된 코드는 읽기 전용으로 처리됨

2) 데이터 영역 data area

• 데이터 영역은 코드가 실행되면서 사용하는 변수나 파일 등의 각종 데이터를 모아 놓은 곳이다.

• 데이터는 변하는 값이기 때문에 기본적으로 읽기와 쓰기 가능하다.

3) 스택 영역 stack area

• 운영체제가 프로세스를 실행하기 위해 부수적으로 필요한 데이터를 모아 놓은 곳이다.

• 스택 영역은 운영체제가 사용자의 프로세스를 작동하기 위해 유지하는 영역이므로 사용자에게 보이지X

4) 힙 영역 heap area

• 프로그램 실행 중 동적으로 메모리가 할당되는 데이터 영역 ex) malloc, new Person

2. 프로세스 생성과 복사

1) fork() 시스템 호출의 개념

• fork() 시스템 호출은 실행 중인 프로세스로부터 새로운 프로세스를 복사하는 함수

-> 이 때, 실행하던 프로세스는 부모 프로세스, 새로 생긴 프로세스는 자식 프로세스로서 부모-자식 관계

가 된다.

2) fork() 시스템 호출의 동작 과정

• fork() 시스템 호출을 하면 프로세스 제어 블록을 포함한 부모 프로세스 영역의 대부분이 자식 프로세스

에 복사되어 똑 같은 프로세스가 만들어짐.

• 단, 프로세스 구분자(PID), 메모리 관련 정보, 부모 프로세스 구분자와 자식 프로세스 구분자 내용이

다릅니다.

3) fork() 시스템 호출의 장점

• 프로세스의 생성 속도가 빠르다

-> fork를 이용해 크롬 하나 열 때 메인 메모리에 있는 내용을 copy해서 자식 프로세스를 만들 수 있다.

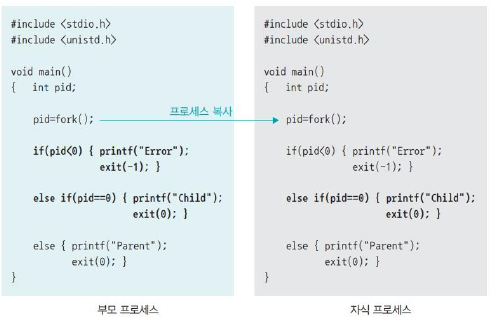
그러나, 크롬 다시 더블 클릭하는 것은 하드디스크에 있는 것을 생성하기 때문에 생성이 되는 속도가

느리다.

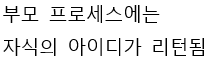
• 추가 작업 없이 자원을 상속할 수 있다.

• 시스템 관리를 효율적으로 할 수 있다.

-> 자원을 부모로부터 받아서 실행되기 때문에 자원관리는 부모의 책임이 됨

 4) fork() 시스템 호출의 예





-> fork() 문을 만나면 똑 같은 내용의 자식 프로세스를 하나 생성하게 된다. fork() 문은 부모 프로세스에

0보다 큰 값을 반환하고 자식 프로세스에 0을 반환한다. 만약 0보다 작은 값을 반환하면 자식 프로세스

가 생성되지 않은 것으로 여겨 ‘Error’를 출력한다.

3. 프로세스의 전환

1) exec() 시스템 호출의 개념 ex) execl()/exep()

• exec() 시스템 호출은 기존의 프로세스를 새로운 프로세스로 전환하는 함수

• 프로세스는 그대로 둔 채 내용만 바꾸는 시스템 호출이다.

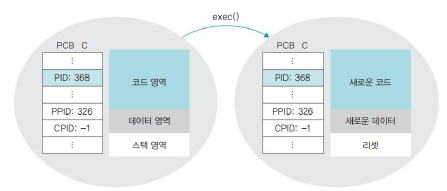
2) exec() 시스템 호출의 동작 과정

• exec()시스템 호출을 하면 코드 영역에 있는 내용을 지우고 새로운 코드로 바꿔버린다.

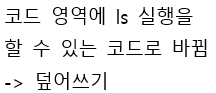
• 새로운 변수로 채워지고 스택 영역이 리셋 된다.

• 프로세스 제어 블록의 내용 중 프로세스 구분자, 부모 프로세스 구분자, 자식 프로세스 구분자,

메모리 관련 사항 등은 변하지 않지만 프로그램 카운터 레스터 값을 비롯한 각종 레지스터와 사용한

 파일 정보가 모두 리셋된다. -> 환경이 변하게 된다.



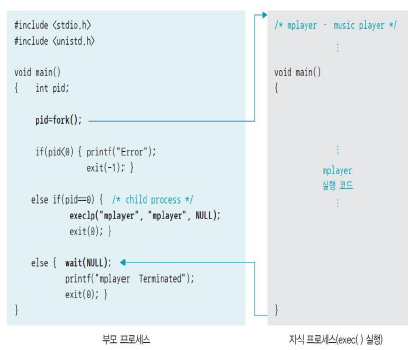


코드 예)

int main(){

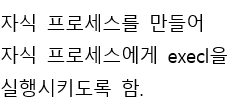
printf(“Begin \n”);

printf(“bin/ls”,”ls”,”-al”,Null); // ls -al 명령어를 실행

 printf(“End \n”); //overwrite가 되어 버려서 이 부분을 실행X

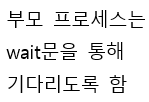
}

3) exec() 시스템 호출의 예

 - fork() 문을 실행하여 자식 프로세스를

생성하고, wait()문을 실행하여 자식

프로세스가 끝날 때까지 기다린다.

 - execlp(“mplayer”,”mplayer”,NULL);이

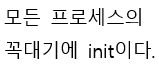
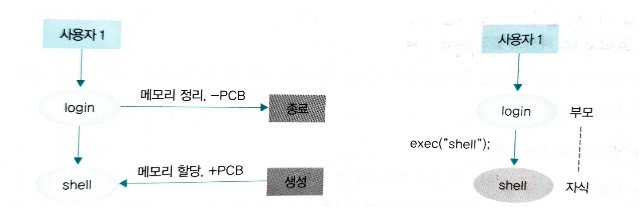
실행되는 순간 자식 프로세스 코드영역이

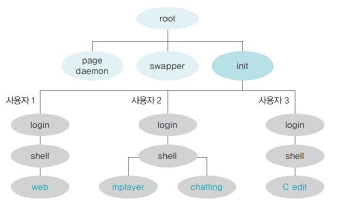
바뀌게 된다.

4. 프로세스의 계층 구조

1) 유닉스의 프로세스 계층 구조 -> 계층 구조를 이룸

• 프로세스를 효율적으로 관리하기 위해 init 프로세스를 만든 다음 나머지 프로세스를 init 프로세스의

자식으로 만든다.





2) 프로세스 계층 구조의 장점

• 여러 작업의 동시 처리

- fork() 시스템 호출로 login 프로세스를 여러 개 만들어 사용자에게 나누어준다.

- 새로운 사용자가 들어올 때마다 이러한 작업을 반복하면 여러 사용자들 동시에 처리할 수 있다.

• 프로세스 재사용이 용이하다

- exec() 시스템 호출을 사용하여 login 프로세스의 구조를 shell 프로세스로 다시 활용하면

자원을 효율적으로 관리할 수 있다.

• 용이한 자원 회수

- 프로세스 간의 책임 관계가 분명해져서 시스템을 관리하기가 수월하다.

- 프로세스가 작업을 마쳐서 그 프로세스가 사용하던 자원을 회수할 때 편리하다.

- 모든 프로세스를 부모-자식 관계로 만들면 자식 프로세스가 작업을 마쳤을 때 사용하던 자원을

부모 프로세스가 회수하면 된다.

3) 미아 프로세스

• 프로세스가 종료된 후에도 비정상적으로 남아 있는 프로세스

• 자식 프로세스가 비정상적으로 종료되어 부모 프로세스에 연락이 안 되는 경우가 발생한다.

-> 이와 같은 문제 발생시 자식 프로세스가 종료되지 않거나, 종료되었는데도 사용하던 자원이 남음.

ex)

int main(){ -> exit() 또는 return() 문은 자식 프로세스가 작업이 끝났음을

printf(“Hello \n”); 부모 프로세스에게 알린다.

exit(0); -> 명시적으로 코드에 넣어 미아 프로세스를 방지

}

(4) 스레드

1. 스레드의 개념

1) 스레드의 정의

• 프로세스에서 실행 제어만 분리한 실행 단위

• 경량 프로세스(Light Weight Process, LWP)

• 프로세스를 여러 가락으로 나눈 단위가 스레드

• 프로세스가 생성되면 CPU 스케줄러는 프로세스가 해야 할 일을 CPU에 전달하고 실제 작업은 CPU가 수행한다. -> CPU가 처리하는 작업의 단위는 스레드이다.

• 운영체제 입장에서의 작업 단위는 프로세스이고, CPU 입장에서의 작업 단위는 스레드이다.

2) 멀티태스크와 멀티 스레드 그리고 멀티프로세싱 차이

• 멀티태스크

- 여러 개의 프로세스를 구성된 것

- 서로 독립적인 프로세스들이 데이터를 주고 받을 때 사용하며, 이 때 데이터를 주고 받을 때 프로세스 간 통신(Inter Process Communication, IPC)를 사용한다.

Ex) 워드프로세서와 프린터 스풀러 -> 워드프로세스가 비정상적으로 종료해도 프린터 스풀러는 정상적으로 작동한다.

• 멀티스레드

- 하나의 프로세스에 여러 개의 스레드로 나누어 동시에 실행한다.

- 한 프로그램 안에서 여러 스레드를 나누어 서비스를 해주도록 한다.

- 운영체제가 소프트웨어적으로 프로세스를 작은 단위의 스레드로 분할하여 운영한다.

• 멀티태스킹

- 동시에 여러 개의 프로그램을 실행시키는 것

- 운영체제가 CPU에 작업을 줄 때 시간을 잘게 나누어 배분한다.

• 멀티프로세싱

- CPU를 여러 개 사용하여 여러 개의 스레드를 동시에 처리하는 작업 환경

• CPU 멀티스레드

- 하드웨어적인 방법으로 하나의 CPU에서 여러 스레드를 동시에 처리하는 병렬 처리 기법

3) 멀티 태스킹과 멀티 스레드의 차이

• 멀티 태스킹의 단점

- fork() 시스템 호출로 프로세스를 복사하면 코드 영역과 데이터 영역의 일부가 메모리 중복이 존재하며, 부모-자식 관계이지만 서로 독립적인 프로세스이므로 이러한 낭비 요소를 제거할 수 없다.

- 코드 영역, 프로세스 제어 블록, 공유 번슈가 메모리의 여러 곳에 중복되어 메모리 낭비가 된다.

• 멀티 스레드의 장점

- 응답성 향상

-> 여러 스레드를 동시에 실행하는 것으로, 한 스레드에서 긴 작업을 해도 다른 스레드가 실행을

하여 사용자에게 응답해준다.

- 코드, 파일 등의 자원을 공유함으로써 자원 중복을 최소화하여 자원 낭비를 막고 효율성을 향상

- 다중 CPU 지원

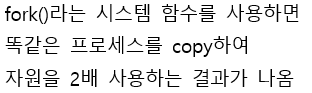
-> 2개 이상의 CPU를 가진 컴퓨터에서 멀티 스레드를 사용하면 다중 CPU가 멀티 스레드를 동시에

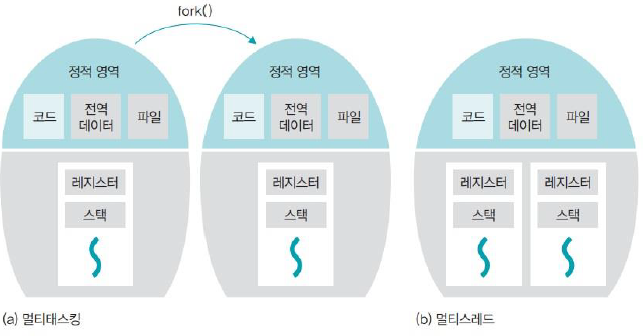
처리하여 CPU 사용량이 증가하고 프로세스의 처리 시간이 단축된다.

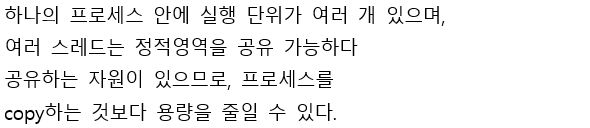
-> 멀티코어, 멀티 스레드 지원하는 CPU를 효율적으로 사용하기 위해 멀티 스레드 실행

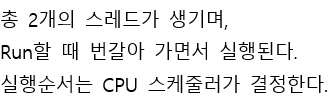
• 멀티 스레드의 단점

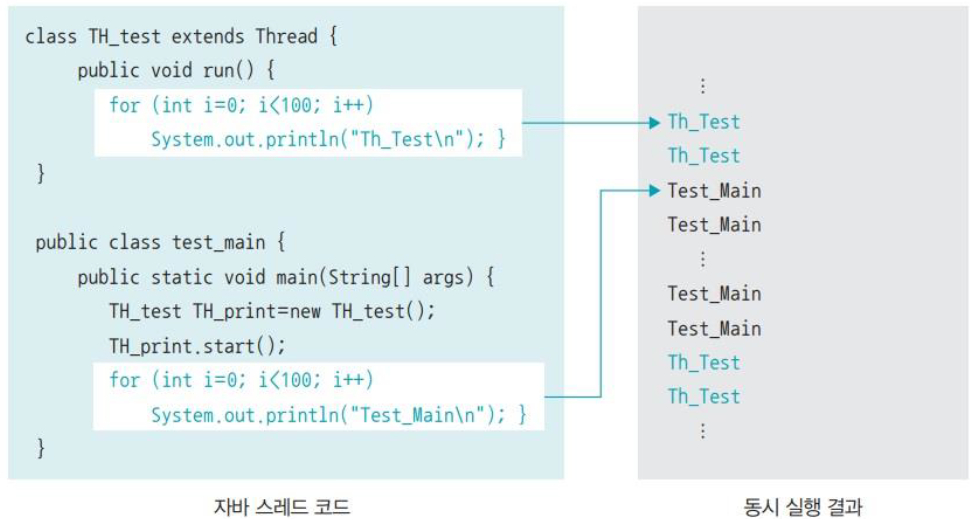
- 멀티 스레드의 경우 모든 스레드가 자원을 공유하기 때문에 한 스레드에 문제가 생기면 전체

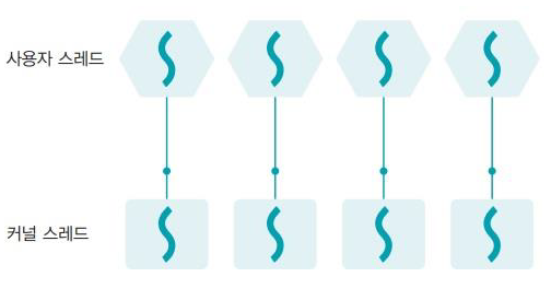
 프로세스에 영향을 미친다.









2. 멀티스레드 모델

1) 커널 스레드

• 커널이 직접 생성하고 관리하는 스레드

• 운영체제 자체에서 제공하는 스레드

• 스레드 관리와 관련된 모든 작업이 커널에 의해서 이루어짐

• 하나의 사용자 스레드가 하나의 커널 스레드와 연결되기 때문에 1 to 1 모델이라고 부른다.

• 커널 스레드 여러 개가 실행 단위로 제공되기 때문에 사용자 스레드가 하나씩 mapping됨->1:1 모델

• 커널 스레드는 독립적으로 스케줄링이 되므로, 특정 스레드가 대기 상태에 들어가도 다른 스레드는

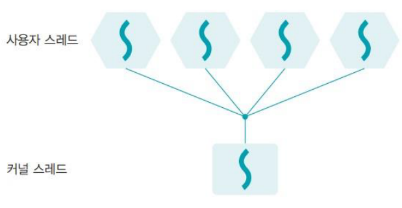
작업을 계속할 수 있다.

• 커널 스레드의 장점은 하나의 스레드가 대기 상태에 있어도 다른 스레드는 작업을 계속 할 수 있다.

• 장점: 멀티 스레드를 커널 레벨에서 관리하기 때문에 보안에 강하고 안정적이다.

• 장점: 멀티코어나 멀티 스레드를 제공하는 CPU가 있다면 mapping이 쉬어서 자원을 제대로 활용

• 단점: 스레드 생성이나 관리가 커널이 직접 하므로 커널 모드와 사용자 모드 간의 모드 전환이

 자주 발생되고 문맥 교환을 할 때 오버헤드 때문에 느리게 작동한다.

2) 사용자 스레드

• 라이브러리에 의해 구현된 일반적인 스레드

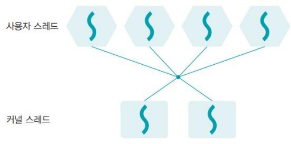
• 운영체제가 멀티 스레드를 지원하지 않을 때 사용하는 방법

• 사용자 프로세스 내에 여러 개의 스레드가 존재하지만 커널의 스레드 하나와 연결되기 때문에 1 to N 모델

• 사용자 레벨에서 여러 개의 스레드가 존재하지만 운영체제에서는 멀티 스레드를 지원하지 않기 때문에 실제로는 하나의 커널 스레드에 mapping 되는 관계라서 1:다 이다.

• 라이브러리가 직접 스케줄링을 하고 작업에 필요한 정보를 처리하기 때문에 문맥 교환이 필요 없다.

-> 운영체제 레벨에서 이루어지며 하나의 커널 스레드에서는 필요 X

 • 단점: 커널 스레드가 입출력 작업을 위해 대기 상태에 들어가면 모든 사용자 스레드가 같이 대기하게 됨

• 단점: 여러 스레드가 공유하기 때문에 여러 개의 CPU를 동시에 사용할 수 없다.

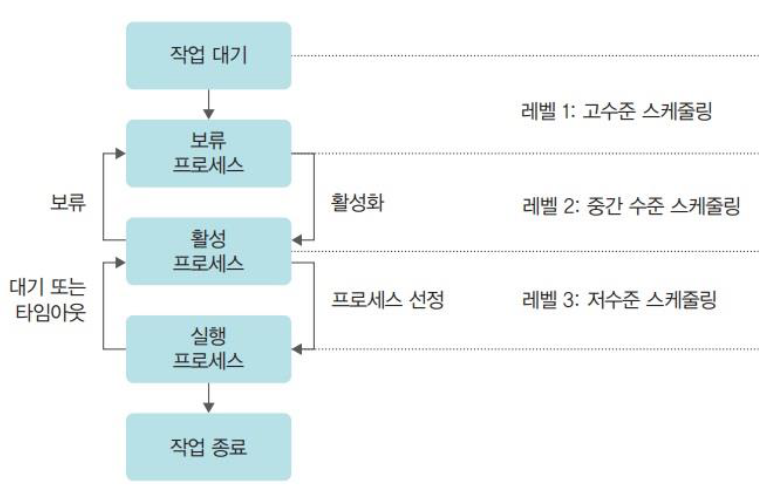
3) 멀티 레벨 스레드

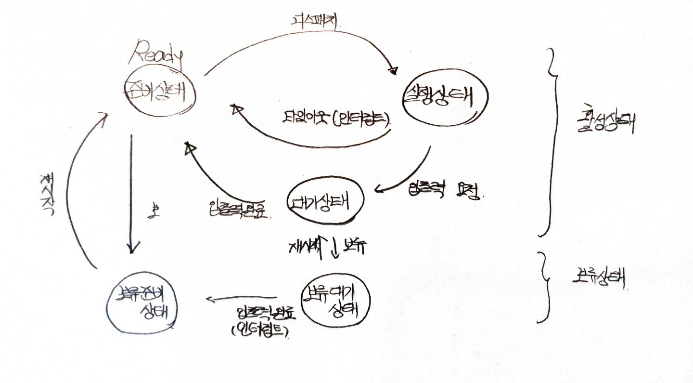
• 사용자 레벨 스레드와 커널 레벨 스레드를 혼합한 방식이므로 M to N 모델이라고 부른다.

• 하나의 커널 스레드가 대기 상태에 들어가면 다른 커널 스레드가 대신 작업을 하여 사용자 스레드보다 유연하게 작업을 처리 할 수 있다.

• 커널 스레드를 같이 사용하기 때문에 문맥 교환 시 오버헤드가 있어 사용자 스레드만큼 빠르지 않다.

-> 빠르게 움직여야 하는 스레드는 사용자 스레드 작동하고, 안정적으로 움직일 때 커널 스레드 작동함.

Chapter4) CPU 스케줄링

(1) 스케줄리의 개요



1. 스케줄링의 개요

1) CPU 스케줄러

• 프로세스가 생성된 후 종료될 때까지 모든 상태 변화를 조정하는 일을 한다.

• 스케줄링은 여러 프로세스의 상황을 고려하여 CPU와 시스템 자원을 어떻게 배정할 지를 결정한다.

2) 스케줄링의 단계

• 고수준 스케줄링

- 고수준 스케줄링은 시스템 내의 전체 작업 수를 조절하는 것으로 프로그램들을 중에 어떤 것을 준비 상태로 넣어줄 지를 결정한다.

- 어떤 작업을 시스템으로 받아들일지 또는 거부할 지를 결정한다.

- 고수준 스케줄링에 따라 시스템 내에서 동시에 실행 가능한 프로세스의 총 개수가 정해짐.

- 전체 시스템의 부하를 고려하여 작업을 시작할지 말지를 결정

• 중수준 스케줄링

- 중지(suspend)와 활성화(active)로 전체 시스템의 활성화된 프로세스 수를 조절하여 과부하를 막는다.

- 일부 프로세스를 중지 상태로 옮겨서 나머지 프로세스가 원만하게 작동할 수 있도록 지원한다.

-> 이 때 프로세스의 상태가 보류 상태이다.

- 시스템에 부하를 조절하기 위해 전체 프로세스의 수를 조절

• 저수준 스케줄링

- 어떤 프로세스에 CPU를 할당할 지, 어떤 프로세스를 대기 상태로 보낼지 등을 결정한다.

- 저수준 스케줄링은 아주 짧은 시간에 일어나기 때문에 단기 스케줄링(short-term scheduling)이다.

3. 스케줄링의 목적

1) 공평성

• 모든 프로세스가 자원을 공평하게 배정받아야 하며, 자원 배정 과정에서 특정 프로세스가 배제되어서는 안된다.

2) 효율성

• 시스템 자원이 유휴 시간 없이 사용되도록 스케줄링을 하고, 유휴 자원을 사용하려는 프로세스에는 우선권을 준다.

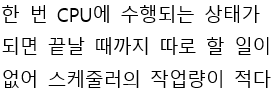
3) 안정성

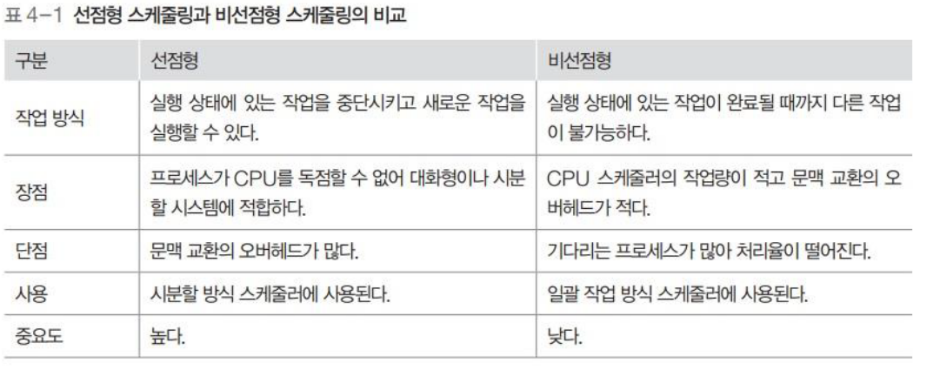
• 우선순위를 사용하여 중요 프로세스가 먼저 작동하도록 배정한다.

4) 확장성

• 프로세스 증가해도 시스템이 안정적으로 작동하도록 조치해야 한다.

5) 반응 시간 보장

 • 응답이 없는 경우 사용자는 시스템이 멈춘 것으로 가정하기 때문에 적절한 시간에 프로세스의 요구에 반응해야 한다.

 6) 무한 연기 방지

• 특정 프로세스의 작업이 무한히 연기되어서는 안 된다.

• 적절한 시간 내에 프로세스 요구에 반응해야 한다.

(2) 스케줄링 시 고려사항

1. 선점형 스케줄링과 비선점형 스케줄링

1) 선점형 스케줄링

• 실행 상태에 있는 프로세스의 작업을 중단시키고 새로운 작업을 시작할 수 있다.

• 하나의 프로세스가 CPU를 독점할 수 없기 때문에 빠른 응답 시간을 요구하는 대화형 시스템이나 시분할 시스템에 적합하다.

• 대부분의 저수준 스케줄러는 선점형 스케줄링 방식을 사용한다.

2) 비선점형 스케줄링

• 어떤 프로세스가 실행 상태에 들어가 CPU를 사용하면 그 프로세스가 종료되거나 자발적으로 대기 상태에 들어가기 전까지는 계속 실행된다.

• 선점형 스케줄링보다 스케줄러의 작업량이 적고 문맥 교환에 의한 낭비도 적다.

• CPU 사용 시간이 긴 프로세스 때문에 CPU 사용 기간이 짧은 여러 프로세스가 오랫동안 기다리게 되어

전체 시스템의 처리율이 떨어짐

• 과거의 일괄 작업 시스템에서 사용하던 방식

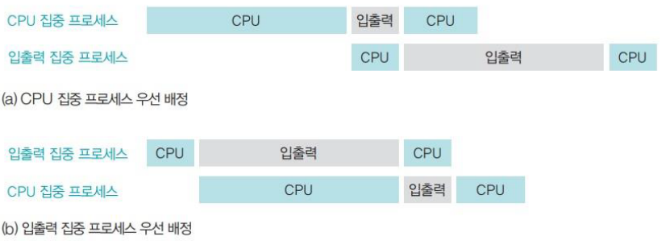
2. 프로세스의 우선순위

1) 프로세스의 우선순위

• 커널 프로세스의 우선순위가 일반 프로세스보다 높다.

• 시스템에는 다양한 우선순위의 프로세스가 공존하며, 우선순위가 높은 프로세스가 CPU를 먼저, 더 오래 차지를 하게 된다.

• 시스템에 따라 높은 숫자가 높은 우선순위를 나타내기도 하고, 낮은 숫자가 높은 우선순위를 나타냄

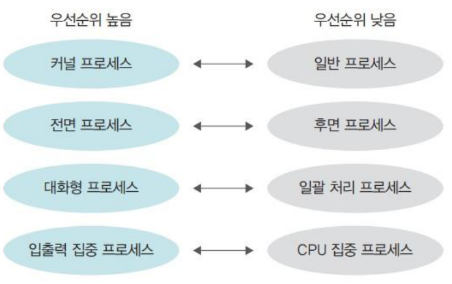
 2) CPU 집중 프로세스와 입출력 집중 프로세스

• CPU 집중 프로세스

- 수학 연산과 같이 CPU를 많이 사용하는 프로세스를 말한다.

• 입출력 집중 프로세스

- 입출력을 많이 사용하는 프로세스이다.

 -> CPU 집중 프로세스와 입출력 집중 프로세스가 같이 있을 때 입출력 집중 프로세스를 먼저 사용하는 것이 효율적이다.

-> 스케줄러 입장에서는 CPU를 적게 사용하는 것이 우선순위가 높다.

3) 전면 프로세스와 후면 프로세스

• 전면 프로세스

- GUI를 사용하는 운영체제에서 화면의 맨 앞에 높은 프로세스를 말한다.

- 입출력을 사용하는 프로세스이며, 사용자와 상호작용이 가능하여 상호작용 프로세스라고 한다.

• 후면 프로세스

- 사용자와 상호작용이 없는 프로세스

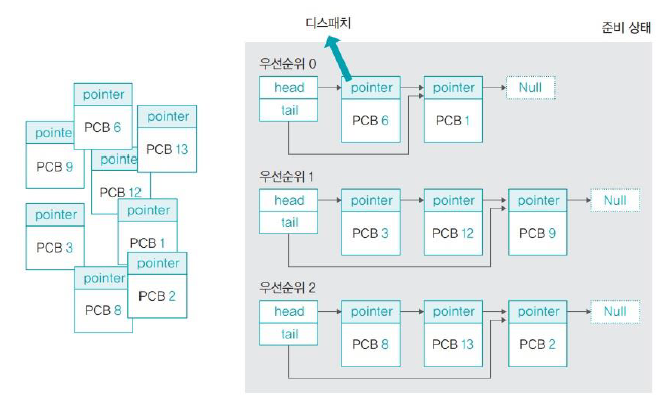
- 사용자의 입력 없이 작동하기 때문에 일괄 작업 프로세스라고 한다.

-> 전면 프로세스는 사용자의 요구에 즉각 반응해야 하지만, 후면 프로세스는 상호작용이 없으므로, 전면 프로세스가 후면 프로세스보다 우선순위가 높다.

(3) 다중 큐

1. 준비 상태의 다중 큐

1) 자신의 우선순위에 해당하는 큐의 마지막에 삽입된다.

 2) CPU 스케줄러는 우선순위가 가장 높은 큐(0번 큐)의 맨 앞에 있는 프로세스 6에 할당한다.

3) 자신이 레벨에 맞는 우선 순위의 큐 tail에 추가 되도록

함.

4) 우선순위 0인 프로세스가 모두 실행되어야 우선순위 1인

프로세스를 실행이 된다.

5) 프로세스 우선순위를 배정하는 방식

• 고정 우선순위 방식

- 운영체제가 프로세스에 우선순위를 부여하면 프로세스가 끝날 때까지 바뀌지 않는 방식

- 우선순위가 변하지 않기 때문에 구현하기 쉽지만, 시스템의 상황이 시시가각 변하는 데 우선순위를 고정하면 시스템 변화에 대응하기 어려워 작업의 효율이 떨어진다.

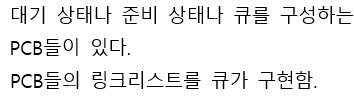
• 변동 우선순위 방식

- 우선순위가 프로세스 작업 중간에 변하는 방식이다.

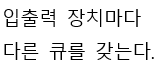
- 구현하기 어렵지만 유연성 있게 우선순위를 부여할 수 있어 시스템의 효율성을 높일 수 있다.

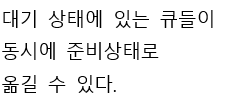
2. 대기 상태의 다중 큐

 1) 시스템의 효율을 높이기 위해 대기 상태에서는 같은 입출력을 요구한 프로세스끼리 모아 놓음.









2) 다중 큐 비교

• 준비 큐

- 한 번에 하나의 프로세스를 꺼내어 CPU를 할당

• 대기 큐

- 여러 개의 프로세스 제어 블록을 동시에 꺼내어 준비 상태로 옮긴다.

- 동시에 여러 개의 인터럽트를 처리하기 위해 인터럽트 벡터라는 자료 구조 사용

(4) 스케줄링 알고리즘

1. 스케줄링 알고리즘 평가 기준

1) CPU 사용률 : CPU가 사용된 시간을 측정하는 방법

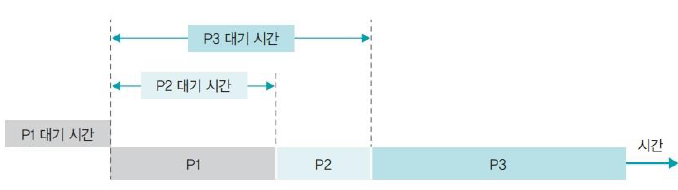
2) 처리량(throughput): 처리량은 단위 시간당 작업을 마친 프로세스의 수 -> 수치가 클수록 좋다.

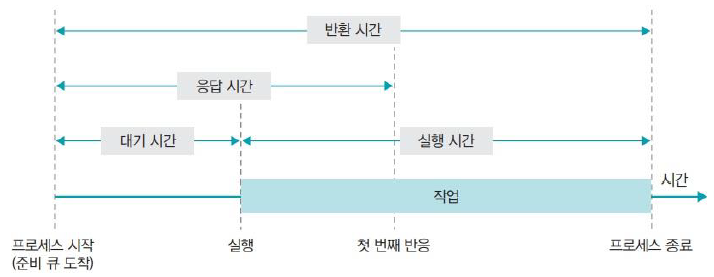
3) 대기 시간(waiting): 실행되기 전까지 대기하는 시간 -> 적게 걸릴수록 좋다.

4) 응답 시간(response): 첫 번째 출력(반응)이 나오기까지의 시간

5) 실행 시간(running): 프로세스 작업 시작 후 종료되기까지의 시간

6) 반환 시간(turnaround): 대기 시간을 포함하여 실행이 종료될 때까지의 시간

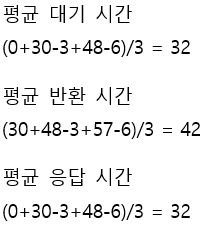




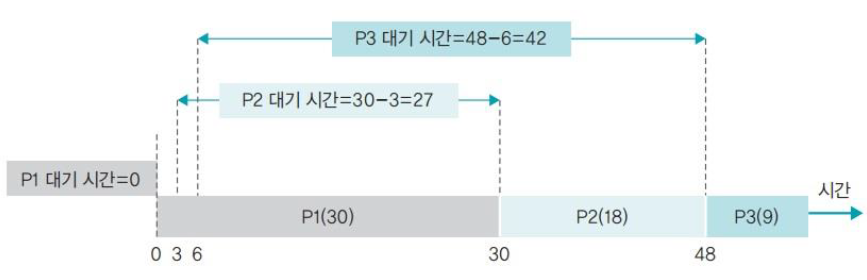


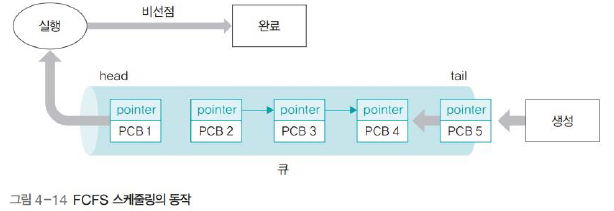
2. FCFS 스케줄링

1) FCFS 스케줄링 동작 방식

 • 준비 큐에 도착한 순서대로 CPU를 할당하는 비선점형 방식

• 한 번 실행되면 그 프로세스가 끝나야만 다음 프로세스를 실행

 • 큐가 하나라 모든 프로세스는 우선순위가 동일



2) FCFS 스케줄링 평가

• 처리 시간이 긴 프로세스가 CPU를 차지하면 다른 프로세스들은 하염없이 기다려 시스템 효율성이 떨어지는 문제

• 현재 작업 중인 프로세스가 입출력 작업을 요청하는 경우 CPU가 작업하지 않고 쉬는 시간이 많아져 작업 효율이 떨어진다.(bath system)

-> 일괄처리방식으로 동작하게 되면 굉장히 오래 기다려 하고 CPU의 idle 타입이 많아 진다는 단점

3. SJF 스케줄링

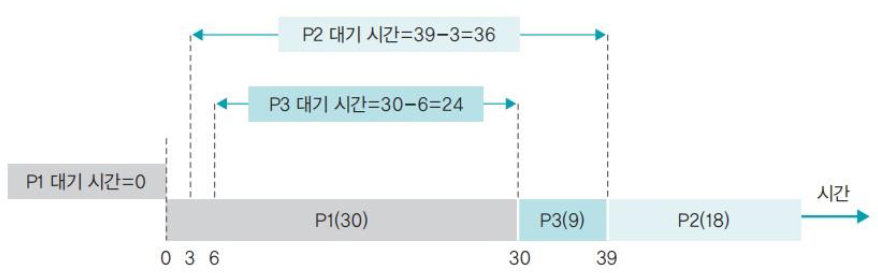
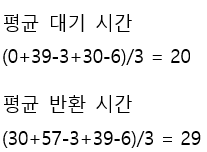
1) SJF 스케줄링 동작 방식

• 실행 시간이 가장 짧은 작업부터 CPU를 할당하는 비선점형 방식으로 최단 작업 우선 스케줄링

• FCFS 스케줄링의 콘보이 효과(convoy effect)를 완화하여 시스템의 효율성을 높인다.

-> 콘보이 효과란 CPU를 매우 오래 사용하는 프로세스가 도착하면, 다른 프로세스가 CPU를 사용하는

데 기다리는 대기 시간



2) SJF 스케줄링의 평가

• 운영체제가 프로세스의 종료 시간을 정확하게 예측하기 어렵다.

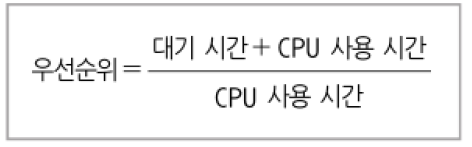
• 공평하지 못하다.

-> 작업 시간이 길다는 이유만으로 계속 뒤로 밀린다면 공평성이 현저히 떨어진다.

• 기아 현상을 방지하기 위해서 에이징(나이 먹기,aging)으로 완화

-> 양보할 수 있는 상한선을 정하고 최대 몇 살까지 양보하도록 규정

• 결론적으로 SJF 스케줄링은 프로세스의 종료 시간을 파악하기 어렵고 아사 현상이 일어나기 때문에

잘 사용하지는 않는다.

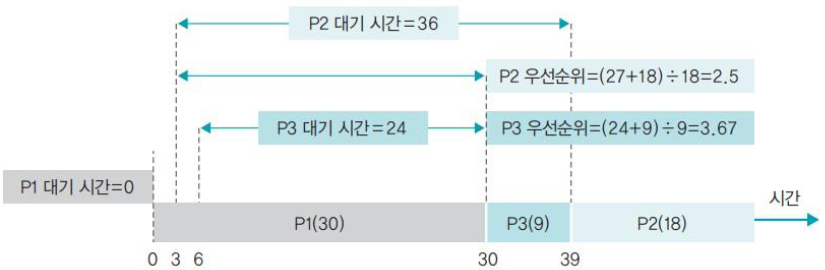
4. HRN 스케줄링

1) HRN 스케줄링의 동작 방식

• SJF 스케줄링에서 발생할 수 있는 아사 현상을 해결하기 위해 만들어진 비선점형 알고리즘

• 대기 시간과 CPU 사용 시간을 고려하여 스케줄링하는 방식

 • 대기 시간이 길어질수록 우선순위가 높아진다.



2) HRN 스케줄링의 평가

• 실행 시간이 짧은 프로세스의 우선순위를 높게 설정하면서 대기 시간을 고려하여 아사 현상을 완화

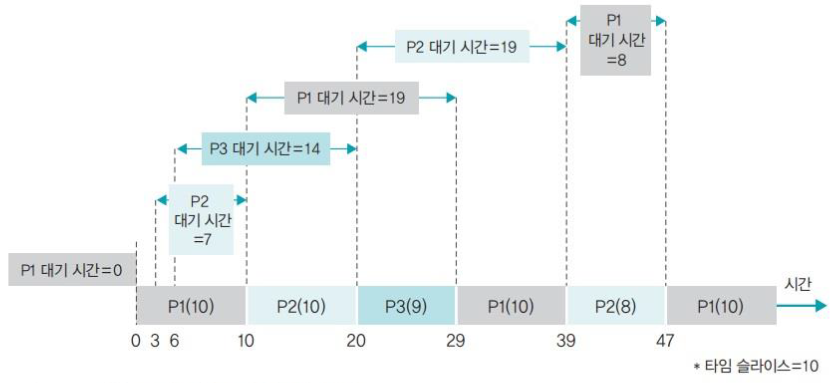
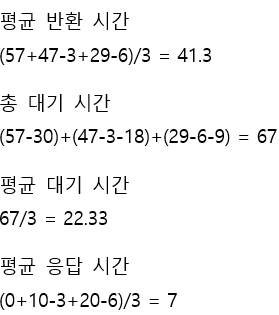
• 대기 시간이 긴 프로세스의 우선순위를 높임으로써 CPU를 할당 받을 확률을 높임

5. 라운드 로빈 스케줄링(Round Robin)

1) 라운드 로빈 스케줄링의 동작방식

• 한 프로세스가 할당받은 시간(타임 슬라이스) 동안 작업을 하다가 작업을 완료하지 못하면 준비 큐의

맨 뒤로 가서 자기 차례를 기다리는 방식

 • 우선순위를 적용하지 않아, 선점형 알고리즘 중 가장 단순하고 대표적인 방식

2) 타임 슬라이스의 크기와 문맥 교환

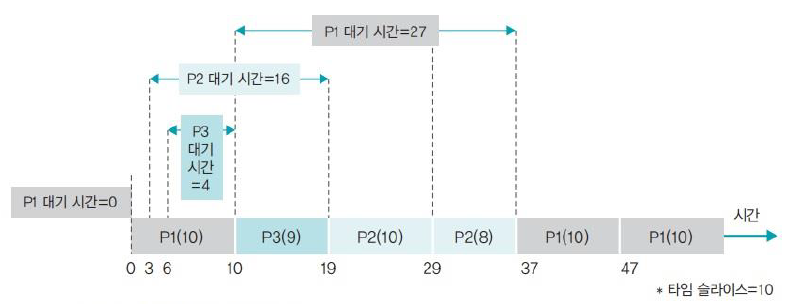
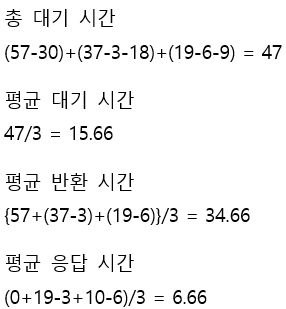
• 타임 슬라이스는 되도록 작게 설정하되 문맥 교환에 걸리는 시간을 고려하여 적당한 크기로 하는

것이 중요

• 타임 슬라이스가 큰 경우 -> FCFS방식 / 타임 슬라이스가 작은 경우 -> 문맥 교환이 많이 발생

6. SRT 우선 스케줄링

1) SRT 스케줄링의 동작 방식

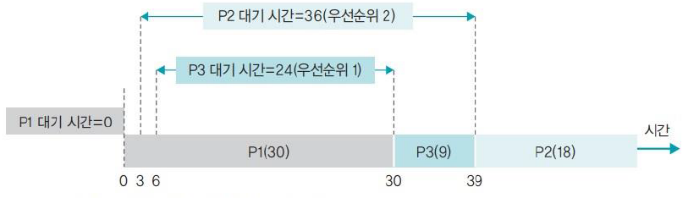
 • CPU를 할당 받을 때 남아 있는 작업 시간이 가장 적은 프로세스를 선택한다.

2) SRT 스케줄링의 평가

• 현재 실행 중인 프로세스와 큐에 있는 프로세스의 남은 시간을 주기적으로 계산하고, 남은 시간이

더 적은 프로세스와 문맥 교환을 해야 함.

• 운영체제가 프로세스의 종료 시간을 예측하기 어렵고 아사 현상이 일어나기 때문에 잘 사용 X

 7. 우선순위 스케줄링

1) 우선순위 스케줄링의 동작 방식

 • 우선순위를 반영한 스케줄링 알고리즘

• 우선순위의 숫자가 낮을수록 우선순위가 높은 것이라고 가정

• 비선점형 우선순위 SJF 스케줄링과 형태 유사

2) 우선순위 적용

• (비선점형 방식) SJF 스케줄링: 작업 시간이 짧은 프로세스에 높은 우선순위를 부여

• (비선점형 방식) HRN 스케줄링: 작업 시간이 짧거나 대기 시간이 긴 프로세스에 높은 우선순위 부여

• (선점형 방식) SRT 스케줄링: 남은 시간이 짧은 프로세스에 높은 우선순위를 부여

3) 고정 우선순위 알고리즘 & 변동 우선순위 알고리즘

• 고정 우선순위 알고리즘

- 한 번 우선순위를 부여 받으면 종료될 때까지 우선순위가 고정된다.

- 시시각각 변하는 시스템 상황을 반영하지 못해 효율성이 떨어진다.

• 변동 우선순위 알고리즘

- 일정 시간마다 우선순위가 변한다.

- 새로 계산하고 이를 반영하기 때문에 시스템이 복잡하지만 시스템의 상황을 반영하여 효율적인

운영이 가능

4) 우선순위 알고리즘 평가

• 프로세스의 순서를 무시하고 우선순위가 높은 프로세스에 먼저 CPU를 할당하므로, 공평성을 위배하고 아사현상을 일으킨다는 것이 문제

• 우선순위를 매번 바꿔야 하기 때문에 오버헤드가 발생하여 시스템의 효율성을 떨어뜨린다.

• 시스템의 효율성이 아니라 프로세스의 중요도를 기준으로 결정하여 스케줄링을 한다.

8. 다단계 큐 스케줄링

1) 다단계 큐 스케줄링의 동작 방식

• 우선순위에 따라 준비 큐를 여러 개 사용하는 방식

 • 운영체제로부터 부여 받은 우선순위에 해당 우선순위 큐에 삼입한다.

자신이 속한 우선순위의 레벨에 넣게 된다.

상위 우선순위에 있는 큐가 다 실행 되어야

하위 우선순위에 있는 큐가 실행이 된다.

(5) 인터럽트 처리

1. 인터럽트의 개념

1) 인터럽트

• 입출력 완료되면 이벤트를 발생시켜 이 사실을 알리게 되는 게 인터럽트

2) 폴링

• 입출력 요청하면 운영체제가 주기적으로 입출력 장치를 직접 확인하여 처리하는 방식

2. 동기적 인터럽트와 비동기적 인터럽트

1) 동기적 인터럽트

• 프로세스가 실행 중인 명령어로 인해 발생 -> 현재 수행중인 작업과 관련된 것이다.

• 소프트웨어적이다. Ex) trap, 자바에서의 예외처리

2) 비동기적 인터럽트

• 하드웨적인 오류로 발생하는 인터럽트 -> 하드웨어적이다.

3. 인터럽트 처리 과정

1) 인터럽트 발생 시 현재 실행 중인 프로세스는 일시 정지 되고 프로세스 관련 정보를 임시로 저장

2) 인터럽트가 여러 개 올 수 있으므로, 처리 순서를 결정한다.

-> 우선 순위를 고려하여 중요한 인터럽트를 처리할 수 있도록 한다.

3) 인터럽트를 처리해주는 프로그램인 인터럽트 핸들러를 실행시켜준다.

4) 인터럽트 처리 후에 일시 정지된 프로세스를 다시 실행되거나 종료한다.

4. 인터럽트와 이중 모드

1) 커널 모드

• 운영체제와 관련된 커널 프로세스가 실행된 상태

2) 사용자 모드

• 사용자 프로세스가 실행되는 상태

3) 이중 모드

• 운영체제가 두 모드를 전환하며 일 처리하는 것

4) 시스템 호출과 API

• 사용자 프로세스가 시스템 호출을 이용하여 자원에 접근 하는 경우는 자발적인 경우

-> 이 때 APU가 준비해 놓은 다양한 함수를 이용하여 시스템 자원에 접근한다.

• 인터럽트에 의해 커널 모드로 진입하는 것은 비자발적인 경우

-> 잘못된 일을 발생 시 인터럽트 핸들러 발생

Chapter5) 프로세스 동기화

(1) 프로세스 간 통신

1. 프로세스 간 통신의 개념

1) 프로세스 내부 데이터 통신

• 하나의 프로세스 내에 2개 이상의 스레드가 존재하는 경우의 통신

• 전역 변수나 파일을 이용하여 데이터를 주고 받는다.

2) 프로세스 간 데이터 통신

• 같은 컴퓨터에 있는 여러 프로세스끼리 통신하는 경우

• 공용 파일 또는 운영체제가 제공하는 파이프를 사용하여 통신한다.

3) 네트워크를 이용하여 데이터 통신

• 여러 컴퓨터가 네트워크로 연결되어 있을 때 통신이 가능

• 소켓(물리적으로 떨어진 컴퓨터에서 프로세스 간의 통신을 함)을 이용하는 프로세스 간 통신을 네트워킹

-> 전역 변수나 파일을 사용하여 통신하는 것은 운영체제의 도움 없이 진행되는 통신 방식이며,

파이프, 소켓, 원격 프로시저 호출은 운영체제가 제공하는 통신 방식이다.

2. 프로세스 간 통신의 분류

1) 통신 방향의 종류

• 양방향 통신: 데이터를 동시에 양쪽 방향으로 전송할 수 있는 구조 ex) 소켓 통신

• 반양방향 통신: 동시 전송은 불가능하고 특정 시점에 한쪽 방향으로만 전송할 수 있는 구조 ex) 무전기

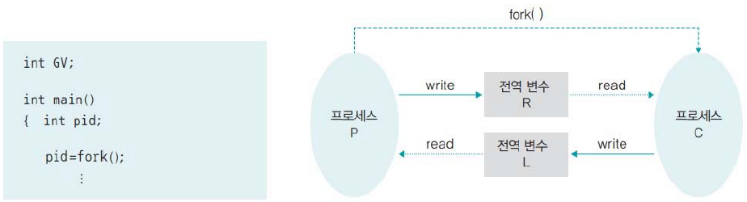
• 단방향 통신: 모스 신호처럼 한쪽 방향으로만 데이터를 전송할 수 있는 구조 ex) 전역 변수, 파이프

 2) 통신 구현 방식에 따른 분류

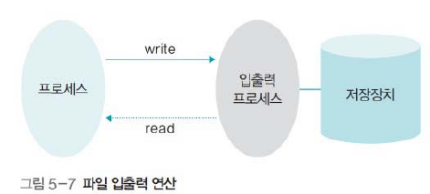
• 대기가 있는 통신: 동기화 지원하는 통신 방식

• 대기가 없는 통신: 동기화 지원하지 않는 통신 방식

데이터를 받는 쪽은 바쁜 대기를 사용하여 데이터 도착 여부를 확인해야 한다.

 -> 바쁜 대기란? 상태 변화를 살펴보기 위해 반복문을 무한 실행

3. 프로세스 간 통신의 종류

 1) 전역 변수/ 공유 메모리(shared memory) 이용한 통신

• 공동으로 관리하는 메모리르 사용해서 데이터를 주고 받는다.

• 전역 변수의 값이 변할 때까지 바쁜 대기를 돌면서 계속 주시하는 것이다.

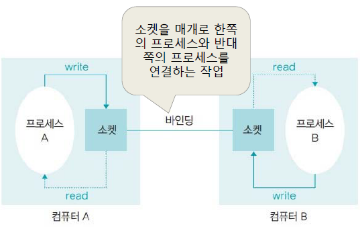
2) 파일을 이용한 통신

• 저장장치에 파일을 읽고 쓰는 코드 / 파일을 열어서 읽고 쓰는 작업을 다한 뒤 닫으면 된다.

3) 파이프을 이용한 통신

• 파이프는 운영체제가 제공하는 동기화 통신 방식

• 파일 입출력과 같이 open() 함수로 기술자를 얻고 작업을 한 후 close() 함수로 마무리

 • 파이프로 양방향 통신을 하려면 파이프 2개 사용해야 한다.

• 이름 없는 파이프(anonymous pipe): 서로 관련 있는 프로세스 간 통신 이용

• 이름 있는 파이프(named pipe): 서로 관련 없는 프로세스 간 통신 이용

4) 소켓을 이용한 통신

• 여러 컴퓨터에 있는 프로세스 간 통신은 네트워킹이며, 네트워킹 상황에서의 통신은 원격 프로시저

호출이나 소켓을 이용

• 통신하고자 하는 프로세스는 소켓에 쓰기 연산을 하면 데이터가 전송되고, 읽기 연산을 하면 데이터를

받게 된다.

• 소켓은 하나만 사용해도 양방향 통신이 가능하다.

• 네트워크 프로그래밍을 흔히 소켓 프로그래밍이라고 부르는 이유는 네트워크의 기본이 소켓이기 때문

(2) 공유 자원과 임계 구역

1. 공유 자원의 접근

1) 공유 자원(shared memory)

• 여러 프로세스가 공동으로 이용하는 변수, 메모리, 파일 등을 말한다.

• 공동으로 이용되기 때문에 누가 언제 데이터를 읽거나 쓰느냐에 따라 그 결과 달라질 수 있다.

2) 경쟁 조건(race condition)

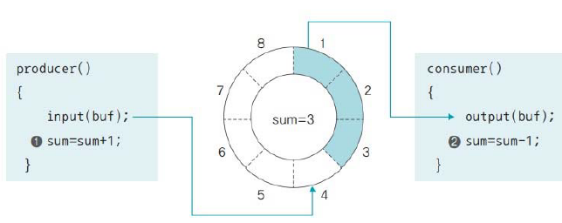
• 2개 이상의 프로세스가 공유 자원을 병행적으로 읽거나 쓰는 상황

2. 임계구역

1) 임계 구역(critical section)

• 공유 자원 접근 순서에 따라 실행 결과가 달라지는 프로그램의 영역

• 프로세스 실행 상황에서는 공유할 수 없는 자원

 3. 생산자-소비자 문제

생산자와 소비자가 타이밍이 맞춰지지 않을 때 sum의 값이 제대로 계산이 되지 않는 문제가 발생한다.

임계구역의 전역 변수를 사용할 때 뿐만 아니라, 하드웨어 자원을 동시에 사용하고 있을 때도 적용되는 개념이다.

4. 임계구역 해결 조건

1) 상호 배제(mutual exclusion)

• 한 프로세스가 임계구역에 들어가면 다른 프로세스는 임계구역에 들어갈 수 없다.

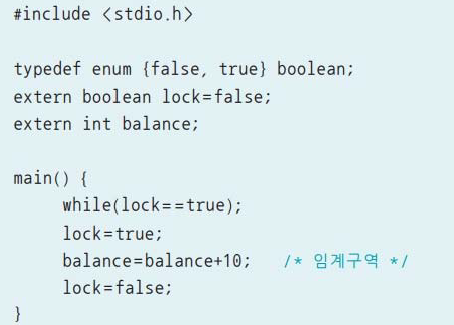
2) 한정 대기(bounded waiting)

• 어떤 프로세스도 무한 대기하지 않아도 된다.

3) 진행의 융통성(process flexibility)

• 진행의 융통성은 한 프로세스가 다른 프로세스의 진행을 방해해서는 안된다.

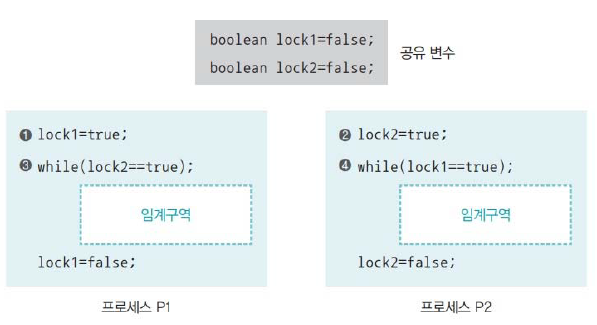
(3) 임계구역 해결 방법

 1. 기본 코드 소개

1) 문제점: 상호배제 보장이 안됨

-> 두 프로세스가 동시에 진입을 할 수 있다는 문제점이 발생

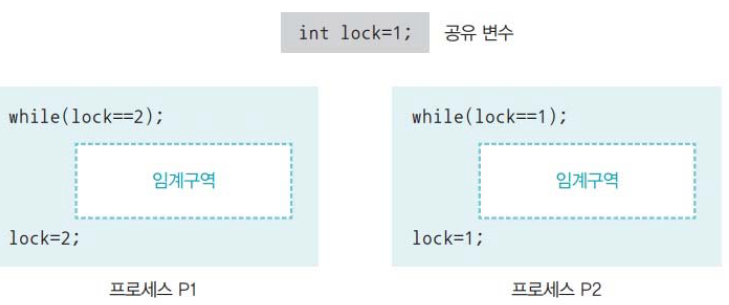
2. 임계구역 해결 조건 코드 설계

 1) 상호 배제 조건을 충족하는 코드

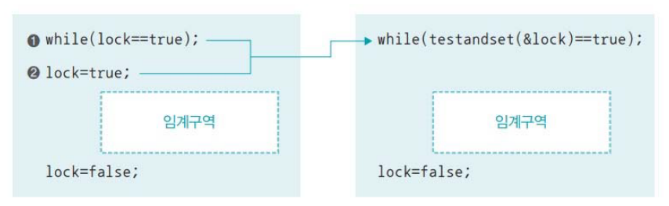
• 문제점: P1과 P2가 while문에 갇히게 되어 교착상태가

발생이 된다.

2) 상호 배제와 한정 대기 조건을 충족하는 코드

 • 문제점: 정해진 순서대로 번갈아 가면서 실행이 되어,

진행의 융통성을 지키지 못했다.

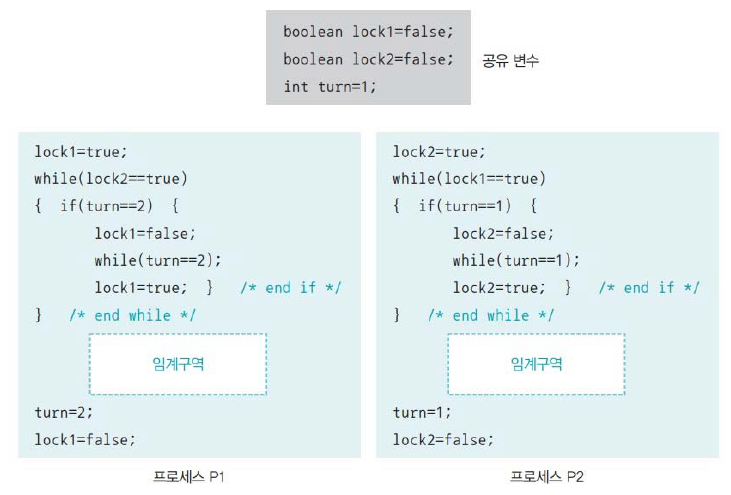
 3) 하드웨어적인 해결 방법

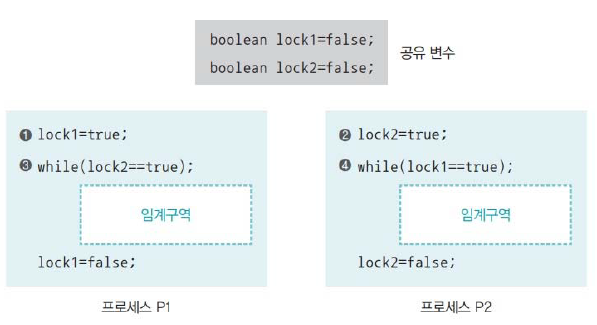
• lock = true와 while(lock==true)문을 동시에 실행하여 임계구역 문제를 해결 할 수 있다.

• 문제점: 검사와 지정 코드 사용 시 명령어 실행 중간에 타임

아웃이 걸려 임계구역을 보호하지 못한다는 문제 발생

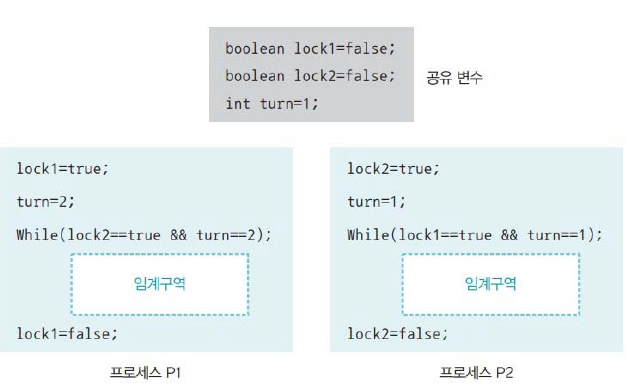
• 문제점: 바쁜 대기를 사용하여 검사하기 때문에 자원낭비 존재

3. 데커 알고리즘

• 한정 대기 문제가 있는 코드인

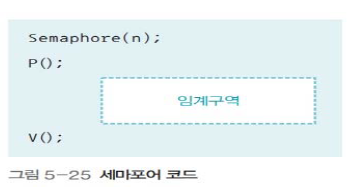
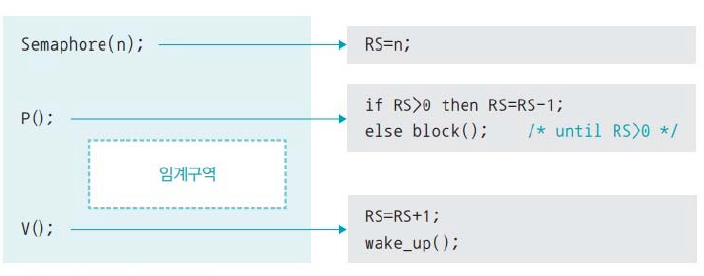
이것을 해결하기 만들어졌다.

• 교착 상태에 빠진 것을 while문 이용하여 해결하고자 했다.

 4. 피터슨 알고리즘

• 데커 알고리즘 부분에서 한 부분을 한꺼번에 체크 해주는 코드이다.

• 2개 프로세스만 사용 가능하다는 점이 한계이다.

 5. 세마포어

• P() -> 잠금을 수행하는 코드

• V() -> 잠금 해제와 동기화를 같이 수행하는 코드

• 세마포어에서 잠금이 해제되기를 기다리는 프로세스는 세마포어 큐에 저장되어 있다가 wake\_up 신호를

받으면 큐에서 나와 임계구역에 진입한다. 그래서 바쁜 대기를 하는 프로세스가 없다.

6. 모니터

1) 세마포어의 한계

• 편리하지만 잘못 사용하는 경우가 있다.

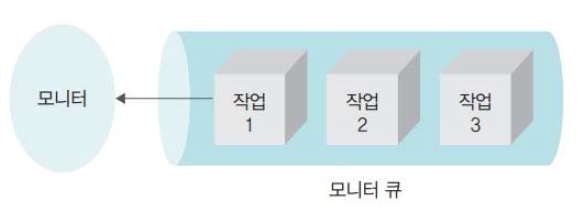
2) 모니터

• 모니터는 공유 자원을 내부적으로 숨기고 공유 자원에 접근하기 위한 인터페이스만 제공

• 자원을 보호하고 프로세스 간에 동기화를 시킨다.

• 시스템 자원을 사용자로부터 숨기고 사용자의 요구 사항을 처리할 수 있는 인터페이스만 제공

• 모니터 사용시 사용자 입장에서는 복잡한 코드를 실행할 필요가 없으며, 시스템 입장에서는 임계구역을

 보호할 수 있다.

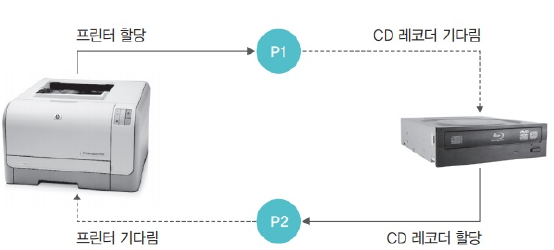
Chapter6) 교착 상태

(1) 교착 상태 개요

1. 교착 상태의 정의

1) 교착 상태

• 2개 이상의 프로세스가 다른 프로세스의 작업이 끝나기만 기다리며 작업

 2) 기아 상태

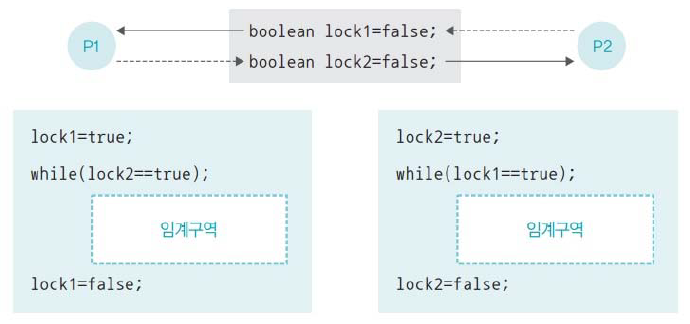
• 어떤 프로세스가 우선 순위가 낮아서 CPU에 할당을 받지 못한 상태

2. 교착 상태 발생

1) 시스템 자원

- 프로세스 P1은 프린터를 할당 받은 후 CD 레코더를 기다리고 프로세스 P2는 CD 레코더를 할당 받은 후

프린터를 기다리면 교착 상태가 발생한다.

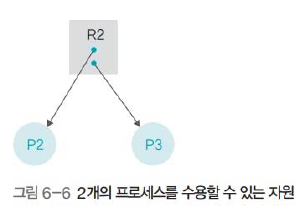
 2) 공유 변수

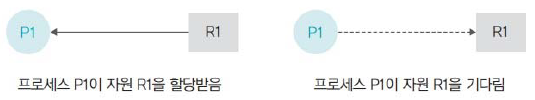
- 교착 상태를 발생시키는 코드로 바쁜 대기를 발생하게 된다.

- 바쁜 대기가 발생하면 CPU가 조건을 체크해야 하기 때문에 안 좋은 상황이다.

3) 응용 프로그램

- 데이터베이스 같은 응용 프로그램에서도 임계구역을 다루기 때문에 교착상태가 발생하게 된다.

 - 데이터베이스의 일괄성을 유지하기 위해 잠금을 사용하는 데 이때 교착 상태가 발생할 수 있다.

 3. 자원 할당 그래프

1) 자원 할당 그래프

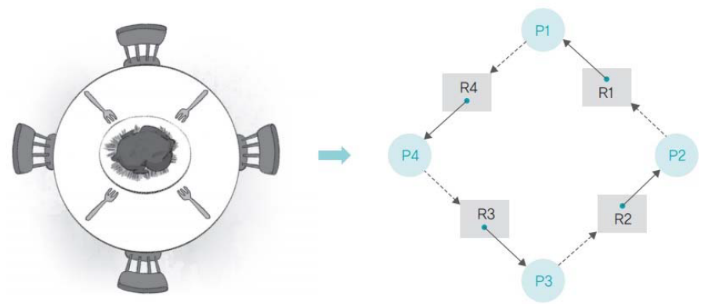
• 프로세스가 어떤 자원을 사용 중이고 어떤 자원을 기다리고 있는 지를

방향성이 있는 그래프

• 다중 자원: 여러 프로세스가 하나의 자원을 동시에 사용하는 경우

-> 다중 자원은 수용할 수 있는 프로세스 수를 사각형 안에 작은 동그라미로 표현

2) 식사하는 철학자 문제

• 왼쪽에 있는 포크를 잡은 뒤 오른쪽에 있는 포크 잡아야 식사 가능

• 실행 조건

- 서로 포크 공유 X, 다른 철학자 포크를 빼앗을 수 X

왼쪽 포크를 잡은 채 오른쪽 포크 기다림, 그래프가 원형

(2) 교착 상태 필요조건

1. 교착 상태 필요조건

1) 상호 배제

• 한 프로세스가 사용하는 자원은 다른 프로세스와 공유할 수 없는 배타적인 자원

2) 비선점

• 한 프로세스가 사용 중인 자원은 중간에 다른 프로세스가 빼앗을 수 없는 비선점 자원

3) 점유와 대기

• 프로세스가 어떤 자원을 할당 받은 상태에서 다른 자원을 기다리는 상태

4) 원형 대기

• 점유와 대기를 하는 프로세스 간의 관계가 원을 이루는 경우

-> 상호 배제와 비선점 조건은 자원이 어떤 특징을 가지는지를 나타내며, 점유와 대기, 원형 대기 조건은

프로세스가 어떤 행위를 하고 있는지를 나타낸다.

2. 식사하는 철학자 문제와 교착 상태 필요조건

1) 상호 배제

• 임계 구역을 다루기 때문에 상호배제 조건을 지켜야 한다.

• 교착 상태에서는 배타적인 자원을 여러 프로세스가 사용해야 하는 경우에 발생

2) 비선점

• 자원을 빼앗을 수 있으면 시간 간격을 두고 그 자원을 공유 할 수 있다는 의미이기 때문에 상호 배제가

성립되지 않는다.

• 교착 상태에서는 비선점 자원을 사용할 때 발생한다.

3) 점유와 대기

• 한번에 점유하거나 둘 다 놓는 행위로 해결

• 두 자원을 다 점유하거나 기다린다는 것은 서로 진행을 방해하는 것이 아니라, 현재 작업을 진행하는 쪽

과 기다리는 쪽의 선후 관계를 만든다.

4) 원형 대기

• 점유와 대기를 하는 프로세스들이 원을 이루면 서로 진행을 방해하는 상태가 되므로 교착 상태가 발생

• 철학자들이 둥그런 식탁이 아니라 사각형 식탁에서 한 줄로 앉아서 식사를 한다면 교착 상태가 발생X

(3) 교착 상태 해결 방법

1. 교착 상태 해결 방법

1) 교착 상태 예방

• 네 가지 조건이 발생하지 않도록 무력화하는 방식

2) 교착 상태 회피

• 자원 할당량을 조절하여 교착 상태를 해결하는 방식

3) 교착 상태 검출과 회복

• 교착 상태 검출은 어떤 제약을 가하지 않고 자원 할당 그래프를 모니터링하면서 교착 상태가 발생하는

지 살펴보는 방식

• 교착 상태를 검출한 후 이를 회복시키는 것이 교착 상태를 해결하는 현실적인 방법

2. 교착 상태 예방

1) 상호 배제 예방

• 상호 배타적인 모든 자원, 즉 독점적으로 사용할 수 있는 자원을 없애 버리는 방법

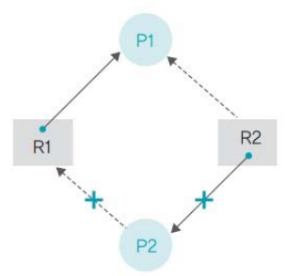
• 현실적으로는 모든 자원을 공유할 수 없으며 상호 배제를 적용하여 보호해야 하는 자원 필요

• 상호 배제 무력화하는 것은 어렵다.

2) 비선점 예방

• 모든 자원을 빼앗을 수 있도록 만드는 방법

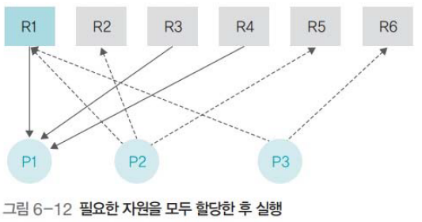
• 자원 특성상 한 번 할당되면 중간에 뺏으면 안 되는 특성이 있어 비선점을 무력화하는 것은 어려운

 방법이다. 또, 자원을 계속 뺏으면 아사현상이 발생한다.

3) 점유와 대기 예방

• 프로세스가 자원을 점유한 상태에서 다른 자원을 기다리지 못하게 하는 방법

• 전부 할당하거나 아니면 아예 할당하지 않는 all or nothing 방식을 적용하는 것

 • 단점

- 프로세스가 자신이 사용하는 모든 자원을 자세히 알기 어렵다.

-> 당장 사용하지도 않을 자원을 미리 선점하여 자원 낭비가 심하다.

- 자원의 활용성이 떨어진다.

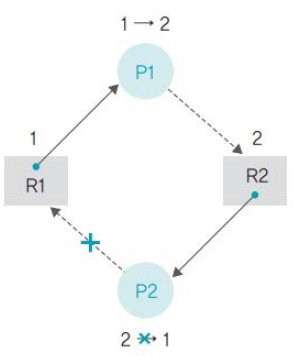
- 많은 자원을 사용하는 프로세스가 적은 자원을 사용하는 프로세스보다 불리하다.

- 결국 일괄 작업 방식으로 동작한다.

-> 한꺼번에 모든 자원을 받아서 작업하기 때문에 결국 P1이 다하고 P2를 하는 식으로 일괄작업 동작한다.

4) 원형 대기 예방

• 점유와 대기를 하는 프로세스들이 원형을 이루지 못하도록 막는 방법이다.

 • 원형 대기 예방의 경우 프로세스들이 자원을 사용하려고 할 때 작은 번호의 자원을 할당 받은 후 큰 번호의 자원을 할당 받도록 한다.

• 단점

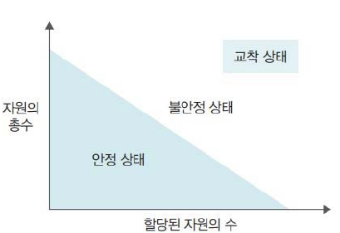
- 프로세스 작업 진행에 대한 유연성이 떨어진다.

-> 우선순위대로 사용해야 하므로 유연성이 떨어진다.

- 자원의 번호를 어떻게 부여할 것인지가 문제이다.

5) 정리

• 교착 상태 예방은 교착 상태를 유발하는 네 가지 조건이 일어나지 않도록 제약을 가하는 방법이다.

 • 상호 배제와 비선점을 예방하기 어려우며, 점유와 대기, 원형 대기는 프로세스 작업 장식을 제한하고, 자원을 낭비한다.

3. 교착 상태 회피

1) 교착 상태 회피 개념

• 교착 상태가 발생하지 않는 범위 내에서만 자원을 할당하고, 교착 상태가 발생하는 범위에 있으면 프로세스를 대기 시킨다.

• 교착 상태 회피는 할당되는 자원의 수를 조절하여 교착 상태를 피한다.

• 교착 상태 회피는 자원의 총수와 현재 할당된 자원의 수를 기준으로 시스템을 안정 상태와 불안정 상태로 나누고 시스템이 안정 상태일 때 자원을 할당하도록 한다.

• 할당된 자원이 적으면 안정 상태가 크고, 할당된 자원이 늘어날수록 불안정 상태가 커진다.

• 불안정 상태에서 항상 교착 상태가 발생하는 것은 아니다.

2) 은행원 알고리즘

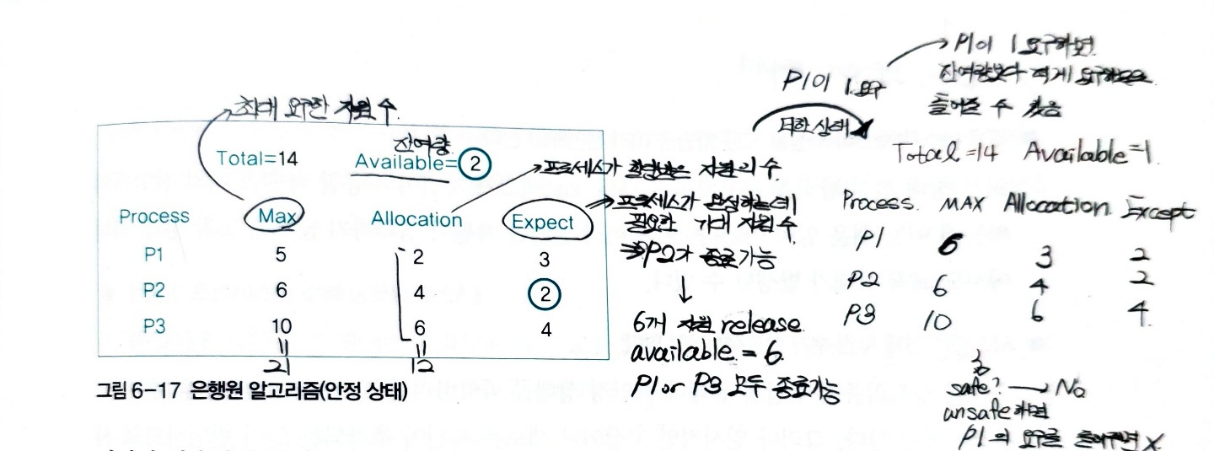
• 교착 상태 회피를 구현하는 대표적인 알고리즘

 • 은행이 대출을 해주는 방식. 즉, 대출 금액이 가능한 범위 내이면(안정 상태) 허용되지만 그렇지 않으면 거부되는 것과 유사

• 각 프로세스의 기대 자원과 비교하여 가용 자원이 하나라도 크거나 같으면 자원을 할당

• 가용 자원이 어떤 기대 자원보다 크지 않으면 할당하지 않는다.

• 안정 상태: 각 프로세스의 기대 자원과 비교하여 가용 자원이 크거나 같은 경우가 한 번 이상인 경우

 • 안정 상태 & 불안정상태

• 문제점 -> 보수적으로 동작하게 된다.

- 프로세스가 자신이 사용할 모든 자원을 미리 선언해야 한다.

- 시스템의 전체 자원 수가 고정적이어야 한다.

- 자원이 낭비된다.

-> 모든 불안정 상태가 교착 상태가 되는 것이 아님에도 불구하고 자원을 할당하지 않는 것은 자원 낭비이다. 프로세스에 따라 드물지만 최대 자원을 사용하지 않고 작업을 마치는 경우가 있다.

4. 교착 상태 검출

1) 교착 상태 검출의 개념

• 교착 상태 검출은 운영체제가 프로세스의 작업을 관찰하면서 교착 상태 발생 여부를 계속 주시하는

방식

• 교착 상태가 발견되면 이를 해결하기 위해 교착 상태 회복 단계를 밟는다.

2) 타임아웃을 이용한 교착 상태 검출 -> 타임아웃을 이용한 예

• 일정 시간 동안 작업이 진행되지 않은 프로세스를 교착 상태가 발생한 것으로 간주하여 처리하는 방법

• 교착 상태가 자주 발생하지 않을 것이라는 가정하에 사용하는 것으로 특별한 알고리즘 없이 구현 가능

-> 타임아웃이 되면 프로세스가 종료

3) 데이터베이스에서 타임 아웃 문제

• 타임아웃으로 데이터의 일관성이 깨지는 문제를 해결하기 위해 데이터베이스에서는 체크포인트와 롤백 사용

• 체크포인트는 작업을 하다가 문제가 발생하면 저장된 상태로 되돌아오기 위한 표시

 • 롤백은 작업을 하다가 문제가 발생하면 과거의 체크포인트로 되돌아가는 것

4) 자원 할당 그래프를 이용한 교착 상태 검출

• 교착 상태 발생시 누구를 희생할 지를 선택

5. 교착 상태 회복

1) 교착 상태 회복: 교착 상태가 검출되면 교착 상태를 푸는 후속 작업을 하는 데 이를 교착 상태 회복이다.

2) 교착 상태 회복 단계에서는 교착 상태를 유발한 프로세스를 강제로 종료한다.

3) 교착 상태를 일으킨 모든 프로세스를 동시에 종료

-> 모든 프로세스를 동시에 종료하는 방법이다. 그러나 프로세스들이 동시에 작업을 하면 다시 교착 상태 를 일으킬 가능성이 있어 어떤 프로세스를 먼저 실행할 것인지 기준이 필요하다.

4) 교착 상태를 일으킨 프로세스 중 하나를 골라 순서대로 종료

-> 어떤 프로세스부터 종료할 석인지 다음과 같은 기준이 필요

• 우선순위가 낮은 프로세스를 먼저 종료

• 우선순위가 같은 경우 작업 시간이 짧은 프로세스를 먼저 종료 -> 남은 시간이 많은 프로세스 종료

• 위의 두 조건이 같을 경우 자원을 많이 사용하는 프로세스를 먼저 종료

-> 많은 자원을 차지하는 프로세스부터 종료한다 그러면 자원이 많이 release가 된다.