

**2020학년 1학기 OS 중간고사**

**2015104124 전자공학과 진우빈**

**jwb0904@naver.com**

**2020.06.01**

목차

# os란?

### 운영체제의 의미와 구조

### 운영체제의 역할

# 하드웨어 구조와 용어

### 컴퓨터 구조

### Interrupt & Polling

### DMA

### Cache Memory

# system Call

### File System

### Process System

# Unix Bootstrap 과정

# Real-Time Scheduling

### Real-Time System

### Scheduling Algorithms

# Synchronization

### Mutual Exclusion Solution

### Semaphore & Critical Section

### Deadlock

# **Chapter 1. OS**

# 컴퓨터를 하나의 국가에 비유한다면 운영체제는 컴퓨터 세계에 있어서 정부와 같은 것이다. 쉽게 말해 컴퓨터의 대부분을 관리하는 역할을 한다. 운영체제는 컴퓨터 하드웨어를 잘 관리하여 성능을 높이고 사용자에게 편의성을 제공하는 프로그램이다. 운영체제가 없는 컴퓨터를 상상해 본다면 어떨까? 사용자는 컴퓨터를 이용하기 위한 어떠한 조작도 할 수 없을 것이며 컴퓨터는 마치 뇌사한 상태나 다름 없을 것이다. 즉 생명은 있으나 아무것도 할 수 없는 상태라는 것이다.

운영 체제의 중요한 목적은 이를테면 다음과 같다.[[1]](#footnote-1)

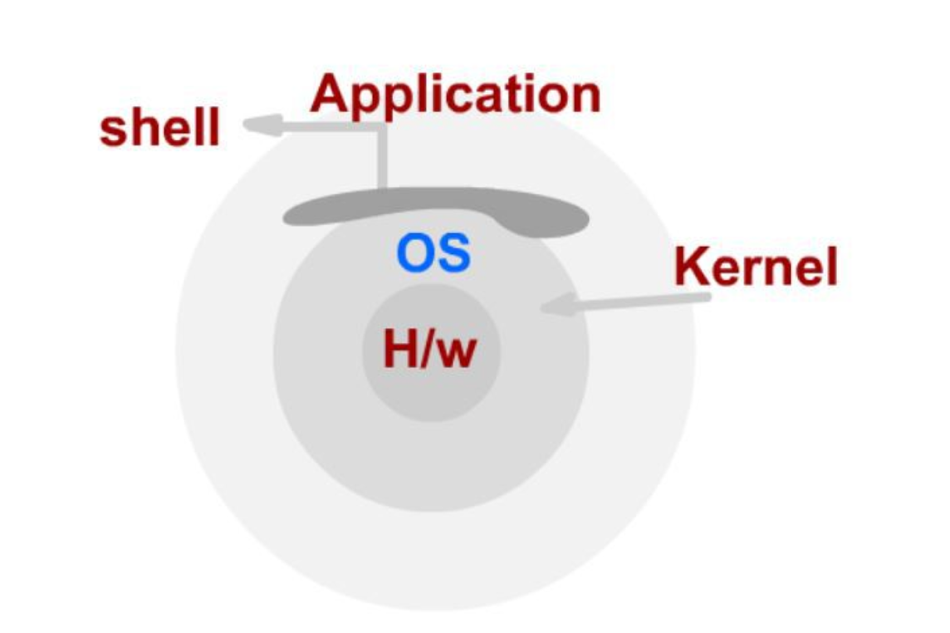
* *사용자에게 컴퓨터의 프로그램을 쉽고 효율적으로 실행할 수 있는 환경을 제공한다.*
* *컴퓨터 시스템 하드웨어 및 소프트웨어 자원을 여러 사용자 간에 효율적 할당, 관리, 보호한다.*
* *운영 체제는 제어 프로그램으로서 사용자 프로그램의 오류나 잘못된 자원 사용을 감시하는 것과 입출력 장치 등의 자원에 대한 연산과 제어를 관리한다.*

즉 자원 관리자 관점에서 본다면 운영체제는 각 자원의 상태를 추적, 저장해야 하고 어떤 프로세스에게 자원을 할당할 것인지를 결정한 후 할당 및 회수를 해야한다. OS는 크게 2가지로 나눌 수 있는데 실제로 하드웨어를 관리하는 OS부분을 kernel 이라고 부른다. 다른 한가지는 shell이다. 즉 shell 껍질인데  OS의 바깥부분에 위치하여 사용자로부터 명령을 받아들이고 그 명령을 해석하고 해당되는 명령을 실행해 주는 것을 쉘이라고 한다.

**커널** : 실제 CPU와 메모리와 디스크 등을 관리한다

**쉘** : 사용자가 명령을 내리면 그 명령을 해석하여 결과를 반응하여 보여준다.

하드웨어 위에 커널이 올라가고 커널 위에서 shell과 application이 실행된다. 사용자는 기본적으로 shell을 통해 application을 실행하는 것이다. 위에서 운영체제를 정부에 비유했다. 그만큼 컴퓨터에 있어서 없어서는 안될 만큼 많은 역할을 하고 있다는 것이다. 운영체제가 하는 역할은 무엇일까?

*[[2]](#footnote-2)*

**1. Process management**

프로세스는 메모리에서 실행 중인 프로그램이다. 보통의 경우 컴퓨터가 실행되면 아직 대부분의 프로그램들은 하드디스크에 보관이 되어있다. 그러다 우리가 더블클릭을 하거나 다른 동작을 하여 프로그램을 실행하게 되면 하드디스크에서 프로그램이 메모리로 옮겨지게 된다. 그러면 프로세스가 되는 것이다. 운영체제는 이렇게 올라온 프로세스에게 CPU를 할당해야 한다. 그런데 CPU는 하나이기 때문에 어떤 프로세스를 실행시킬지는 운영체제가 결정을 한다. 따라서 Process management라고 하면 CPU의 스케쥴링을 해주는 역할을 한다고 할 수 있다.

**2. Memory management**

앞에서 말한 프로그램이 프로세스가 되려면 메인 메모리로 프로그램이 올라와야 한다. 운영체제는 이 과정에서 역할을 한다.  프로그램에게 메모리 공간을 할당하고 다시 하드디스크로 보내주는 역할을 수행하는 것도 운영체제이다. 메모리의 공간이 한정되어 있으므로 프로세스를 잘 적재하는 것도 중요한 역할이 된다.

**3. File management**

파일이라고 하는 것은 Track/sector로 구성된 디스크를 논리적 관점으로 표현한 것이다.  파일을 생성하고 삭제, 디렉토리 생성, 삭제 등 파일에 관련된 역할을 수행하는 것도 운영체제가 한다.

**4. Secondary storage management**

하드디스크나 플래시 메모리와 같은 보조기억장치에 대한 관리도 운영체제가 맡아서 한다. 빈 공간에 대한 관리나 저장 공간 할당, 디스크 스케쥴링과 같은 역할을 수행한다.

**5. I/O device management**

장치 드라이브나 입출력 장치의 성능 향상을 위한 역할을 수행한다. 프로세스가 I/O의 사용이 필요하면 적절하게 연결해주는 역할을 한다.

이외에도 운영체제는 보호나 네트워크 등 하는 역할이 많이 있다.

# **Chapter 2. 하드웨어 구조와 용어**

운영체제를 잘 이해하기 위해서는 컴퓨터 하드웨어의 종류와 기능을 숙지하는 것이 중요하다. 따라서 이번 장에서는 하드웨어에 대해 자세하게 살펴볼 것이다. 컴퓨터 하드웨어의 구성은 크게 3가지로 나눌 수 있다.[[3]](#footnote-3)

**1. CPU (Central Processing Unit)**

중앙처리장치로서 컴퓨터의 연산을 담당, 데이터를 처리한다. CPU 내부에는 ALU, Control Unit, Register set, Bus Interface가 존재한다. CPU에서는 특정 번지에 저장되어 있는 명령어를 읽어오고 실행한다. CPU안에 있는 저장 공간을 Register라고 하고 밖에 있는 저장 공간을 Memory라고 한다.

**2. Main Memory**

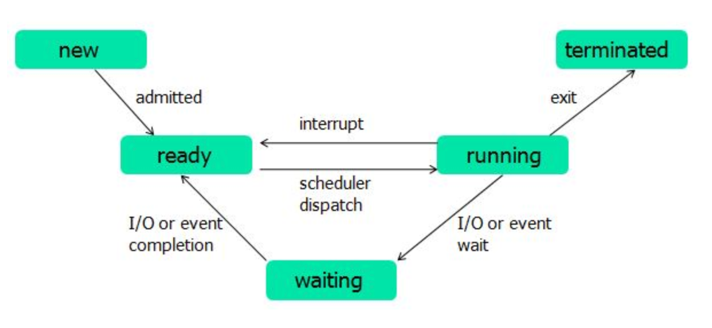
크게 RAM과 ROM으로 나눌 수 있다. 프로그램을 실행하거나 데이터를 참조하려면 모두 메인 메모리에 올려야 한다. 그렇다고 무작정 메인 메모리를 크게할 수 없는데, 불필요한 프로그램과 데이터는 보조기억장치에 저장했다가 실행,참조할 때만 메인 메모리에 옮기는 원리를 적용하고 CPU에서 자주 사용되는 데이터는 캐시메모리에 임시저장한다거나 해서 서로 상호보완한다.

**3. I/O Device[[4]](#footnote-4)**

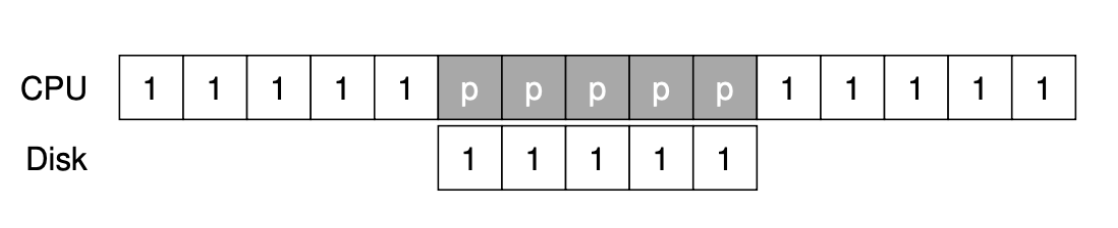
컴퓨터는 목적을 달성하기 위해 CPU/메모리와 외부 장치간에 정보를 주고 받는다. 이를 I/O라고 말할 수 있다. 컴퓨터에서 연산을 한다는 것은 CPU가 무언가 일을 한다는 뜻이다. 입출력 장치들의 I/O 연산은 I/O 컨트롤러가 담당하고 컴퓨터 내에서 수행되는 연산은 메인 CPU가 담당한다. I/O장치에 대해 살펴볼 때는 Interrupt 에 대해 알아야한다. Polling 방식은 CPU가 직접 I/O 장치에 일이 다 끝났니 물어보는 방법이고, Hardware interrupt는 일이 다 끝났으면 I/O장치가 CPU에 알리는 것이다. 당연 히 후자의 방식이 더 성능이 좋다.

**Polling**이란 하드웨어 장치의 상태를 수시로 체크하여 명령을 받을 수 있는 지를 확인하는 것을 말한다. Polling을 하는 동안에는 다른 프로세스에게 CPU를 양도하지 않고 하드웨어 장치가 동작을 완료하는 동안 계속 루프를 돌면서 하드웨어의 상태를 체크하게 된다. 하지만 하드웨어장치의 속도는 매우 느리기 때문에 CPU를 양도하지 않고 하드웨어 장치의 상태를 계속 확인하는 것은 CPU를 많이 낭비하게 되는 것이다.

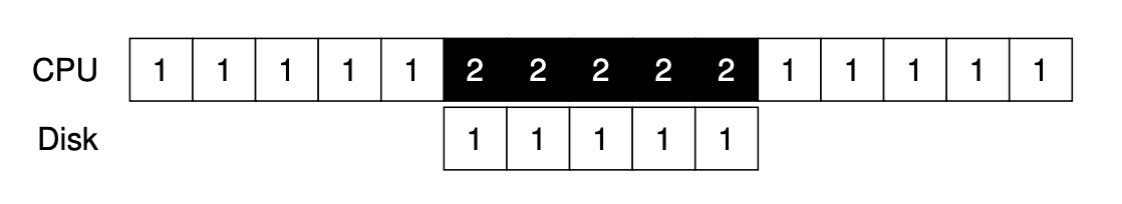
**Interrupt**는 CPU가 프로그램을 실행하고 있을 때 장치에서CPU로 보내는 하드웨어 신호로 실행 중인 프로그램을 미루고 다른 프로그램의 수행을 요구하는 명령이다. 장치가 작업을 끝마치면 하드웨어 인터럽트를 발생시키고 CPU는 운영체제가 미리 정의해 놓은 인터럽트 서비스 루틴(ISR)을 실행한다. Interrupt가 필요한 이유는 여러가지가 있다. 이후에 서술할 Scheduling 에서 프로세스가 Running중에 Scheduler가 이를 중단 시키고 다른 프로세스로 교체하는 경우 그리고 I/O Device와의 통신이 발생할 때, 마지막으로 예외 상황에 대한 Handling으로 Kernel에서 이를 처리할 수 있게 CPU에 알려줘야 한다.

[[5]](#footnote-5)

< 선점형 스케쥴러의 구현>

[[6]](#footnote-6)

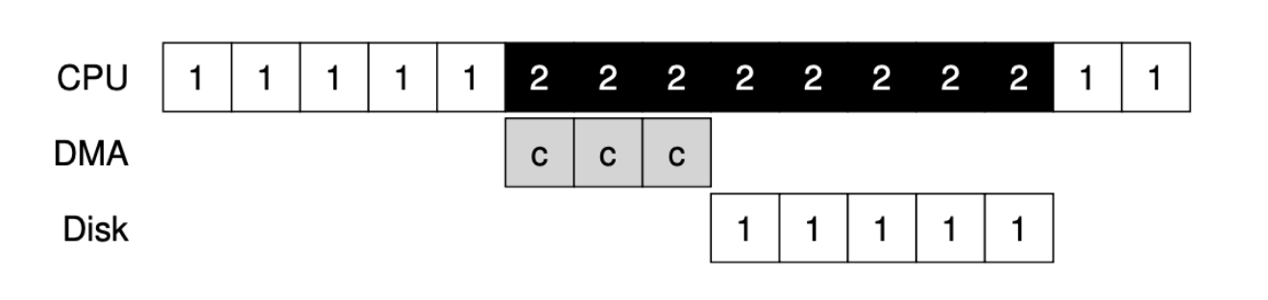
위의 그림은 Polling을 설명하는 그림이다. 프로세스1이 실행 중에 DISK로부터 I/O요청을 발생시키게 된다. 이 때 특별한 인터럽트가 없다면 운영체제는 반복적으로 Polling을 실행하게 된다. 디스크가 I/O 요청의 처리를 완벽하게 완료하게 되면 프로세스 1은 다시 동작하게 되는 것이다.



이 그림은 Interrupt에 대해 나타내고 있다. 프로세스1이 실행하는 도중 디스크로부터 I/O 요청을 받고 요청을 처리하는 도중에 운영체제는 프로세스2를 CPU에서 실행시키게 된다. 프로세스 1에 대한 디스크요청이 완료되면 Interrupt를 발생시켜 운영체제는 프로세스1을 다시 CPU에서 실행시키게 되는 것이다.

이처럼 Interrupt를 사용하게 된다면 CPU연산과 I/O장치 작업을 병렬적으로 수행할 수 있게 된다. 하지만 무조건 Interrupt가 Polling 방식보다 좋은 것은 아니다. Interrupt를 사용하게 되면 현재 실행중인 프로세스를 또다른 프로세스로 Context Switching을 하게 되는데 이 때 많은 비용이 들기 때문이다.

**DMA(Direct Memory Access)**란 CPU의 간섭이 발생하지 않고 메모리와 하드웨어 장치 간의 데이터 전송을 의미하는 것이다. 앞서 서술한 위의 그림에서 많은 데이터를 디스크로 전달하기 위해 Programmed I/O를 사용하게 된다면 CPU가 다른 프로세스를 처리하기 위해 사용될 수 있는 시간들을 낭비하게 된다. 이 문제를 해결하기 위해 DMA가 대두되었다. DMA는 단어 그대로 메모리와 I/O 장치 사이에 직접적인 데이터 전송을 의미한다.



위 그림처럼 데이터를 하드웨어 장치로 전송할 때 운영체제는 DMA에 데이터의 메모리 주소와 전송할 데이터의 크기, 대상 장치를 프로그래밍 한다. 이 시점에 데이터를 전송하기 위한 작업들은 끝나기 때문에 CPU는 다른 작업을 수행할 수 있게 된다. DMA 동작이 끝나면 DMA컨트롤러가 인터럽트를 발생시켜서 운영체제가 전송이 완료되었음을 알린다.

**Cache Memory**란 CPU와 메모리 사이에서 속도 차이에 따른 병목현상[[7]](#footnote-7)을 줄이기 위해 사용되는 범용 메모리이다. 입출력 데이터를 버퍼링하여 Memory에 접근하지 않고 빠른 입출력을 제공시켜주는 장점이 있다. 따라서 최근에 사용된 데이터나 자주 사용되는 데이터를 캐시 메모리에 저장한다. 캐시 메모리는 빠르지만 용량이 적고 비싸다는 단점이 있다. 대부분의 프로그램은 한번 사용한 데이터를 다시 사용할 가능성이 높고 그 주변의 데이터도 곧 사용될 가능성이 높은 데이터의 지역성을 가지고 있다. 지역성은

**1) 공간적 지역성:** CPU가 요청한 주소 지점에 인접한 데이터들이 참조될 가능성이 높다.

**2) 시간적 지역성:** 최근 사용된 데이터가 재사용될 가능성이 높다.

**3) 순차적 지역성:** 데이터가 기억장치에 저장된 순서대로 순차적으로 실행될 가능성이 높다.

이렇게 구분할 수 있다.

캐쉬는 L1, L2로 나눌 수 있는데 보통 CPU에서 L1부터 빠르게 접근할 수 있고 속도가 빠르다. L1에 찾고 싶은 데이터가 없으면 L2에 접근하여 데이터를 찾는다. 속도는 L1>L2>RAM순이고 L3는 있는 경우도 없는 경우도 있다.

# **Chapter 3. System Call**

응용프로그램이 운영체제 기능을 요청할 때를 위해서 운영체제는 시스템 콜을 사용한다. 시스템 콜을 커널과 사용자 사이의 인터페이스 역할을 하는 것으로 Shell에서 명령어나 서브 루틴 형식으로 운영체제의 기능을 호출할 수 있다. 즉, 사용자가 직접 커널에 접근을 할 수 없기 때문에 시스템 콜을 활용해야 한다.  보통 시스템 콜을 직접 사용하기 보다는, 해당 시스템 콜을 사용해서 만든 각 언어별 라이브러리(API)를 사용한다. 쉽게 말하면, 시스템 콜은 운영체제의 기능을 호출하는 함수이다. 수업시간에는 시스템 콜에서 파일과 관련된 File system 관련 함수 및 프로세스와 관련된 관련 함수의 사용 용도와 원리에 대해서 자세하게 살펴보았다.

**1. File System**

간단하게 파일과 관련된 모든 함수가 담겨있는 시스템이다. Open을 예로 들자면, 사용자가 파일 작업을 하기로 마음을 먹었다면 커널에 정해진 파일을 열 수 있도록 허용해 달라고 요청을 해야 할 것이다. 이 때 리눅스의 경우 파일 오픈과 관련된 요청을 위해서 open이라는 시스템 함수를 제공한다. 파일을 다루는 기본적인 흐름은 첫번째로 파일을 열고 두번째로 열려진 파일에서 데이터를 읽거나 쓰고, 마지막으로 모든 작업이 끝나면 파일을 닫는 것이다.

**1) File Table:**

현재 열려있는 파일의 읽기/쓰기 동작을 지원하기 위한 자료구조. open()이나 create()시스템 콜에 의해 파일이 열릴 때마다 하나씩 할당되고 close()시스템 콜에 의해 닫을 때 해제된다. 따라서 하나의 파일에 대해 동시의 여러개의 파일테이블 엔트리가 존재할 수 있다. 테이블 엔트리에는 다음과 같은 정보가 들어간다 f\_pos는 현재 읽기/쓰기 위치를 가리키고 f\_count(참조 카운터) 는 파일 디스크립터 테이블 엔트리의 수를 의미한다.

**2) File Descriptor Table:[[8]](#footnote-8)**

이 테이블은 프로세스마다 하나씩 가지는 것으로 프로세스가 사용중인 파일을 관리하기 위한 테이블이다. 크기가 256이고 파일 테이블에 대한 포인터를 저장하는 배열이다. 프로세스가 open(), create(), dup() 등의 시스템 호출을 할 때마다 하나의 빈 엔트리가 할당되고 해당하는 파일테이블에 대한 포인터가 들어간다. open(), create(). dup()에 의해 반환되는 파일 디스크립터는 이 배열의 인덱스이다. close() 시스템 콜에 의해 엔트리가 제거된다. 예를 들어 "file1"이라는 파일을 열기위한 fd1=open("file1", O\_RDONLY) 시스템 콜은 커널 내에서 다음과 같은 일을 한다.

**1.** 디스크에서 파일(file1)을 찾아 그 inode 정보를 가져와 inode 테이블의 빈 엔트리를 채우고 참조 카운터(i\_count)를 1로 한다. 그 파일의 inode 정보가 이미 테이블에 있으면 새로운 엔트리를 할당하지 않고 참조 카운터만 증가시킨다.

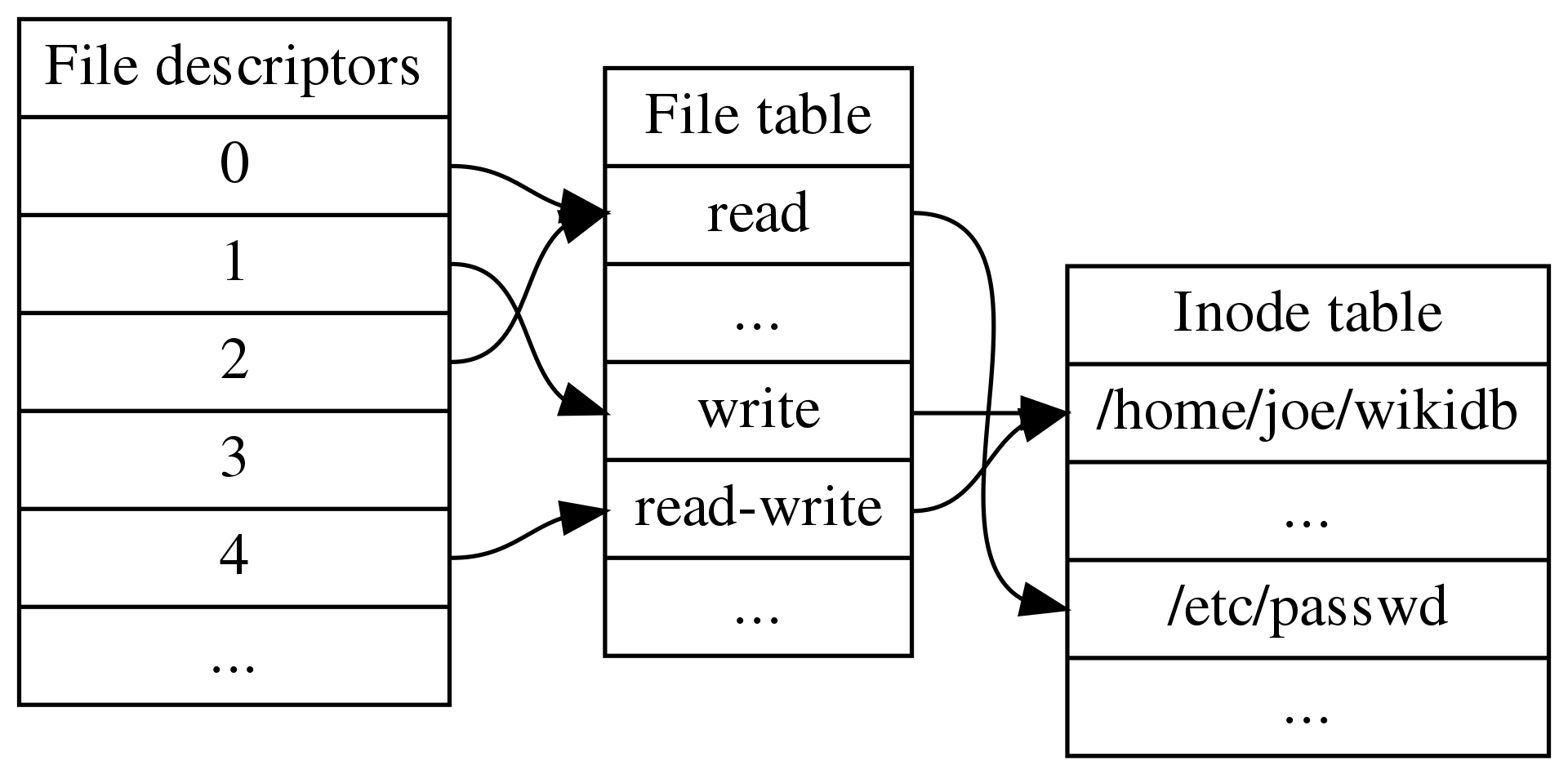
**2.** inode에 있는 접근 권한이 열기 모드를 허용하는지 조사한다.

**3.** 파일테이블 엔트리를 할당하고 읽기/쓰기위치(f\_pos)를 0으로 한다. (만약 열기모드에 O\_APPEND가 있으면 f\_pos를 파일 크기와 같게 한다.) 그리고 참조 카운터(f\_count)를 1로 한다.

**4.** fd 테이블을 처음부터 탐색해서 사용되지 않는 영역에 파일 테이블 엔트리의 포인터를 기록하고, 그 인덱스를 반환한다.

응용 프로세스가 파일을 열거나 생성하게 되면 정수로 된 파일 디스크립터를 얻게 되는데 이 파일 디스크립터는 이후에 일어나는 모든 파일 동작 즉, 읽기(read()), 쓰기(write()). 파일 동작제어(fcntl()), 파일 닫기(close())등의 동작에서 그 파일을 가리키는데 사용된다. 대부분의 리눅스 프로세스에서 파일디스크립터 0은 표준입력, 1은 표준출력, 2는 표준 오류장치를 나타내도록 미리 열려 있다. 이중 표준 입력 장치는 키보드, 표준 출력/오류장치는 모니터를 나타낸다

파일 테이블 엔트리는 파일을 열 때마다 새로 할당되고 파일 디스크립터 테이블의 엔트리는 파일 디스크립터가 새로 생성될 때마다 새로 할당된다. 한편 파일 닫기 동작 close()는 open()의 반대 과정을 수행한다. 먼저 디스크립터 테이블에서 해당 포인터를 지우고 파일테이블의 f\_count를 감소시킨다. f\_count가 0이 되면 이 엔트리를 해제하고 다시 inode 테이블의 i\_count를 감소시킨다. inode테이블의 i\_count가 0이되면 이 엔트리를 해제한다

[[9]](#footnote-9)

<File Descriptor, FILE Table, Inode Table 자료구조 원리>

**2. Process System**

운영체제에서의 프로세스 생성 원리는 다음과 같다.[[10]](#footnote-10)

1. 먼저 프로세스가 생성이 되려면 파일 시스템의 그 대상이 되는 exe파일이 있어야 한다. 이 exe파일의 Path를 OS에게 전달 해야한다.
2. OS는 실행 가능한 파일들의 코드를 읽어 들인다. -> 즉 Memory Context의 한 부분인 Code 세그먼트로 exe 파일의 코드들을 읽어 들인다.
3. 또 exe파일에는 전역 변수의 선언과 정보 들이 있다. 이 정보들을 기반으로 Data 세그먼트에 알맞게 영역을 잡아준다.
4. 이렇게 Code 세그먼트와, Data 세그먼트로 exe파일들을 불러들이는이 작업을 프로그램을 로드 한다고 말한다.
5. 스택 세그먼트와, 힙 세그먼트는 아직 아무것도 없으므로 초기화만 시킨다.
6. 그리고 그 프로세스의 정보를 담은 PCB(Process control block)를 Malloc 해서 필요한 정보를 채워 넣는다.
7. 이렇게 생성돤 PCB를 레디큐에 장착시킨다.

이렇게 생성된 프로세스를 커널에서는 커널 프로세스 테이블로 관리하고 각 프로세스에는 이 테이블에 여러 항목이 있다. 이 항목은 커널에 의해서만 조작되는 데이터가 들어 있는 U(사용자)영역을 가리키며, 각 프로세스에는 프로세스 영역별 테이블이 있다. 영역 테이블의 항목은 영역의 속성을 설명합니다. 예를 들어 영역의 데이터가 공유되어 있거나 개인 데이터가 있는 경우에는 영역의 속성을 설명합니다. 영역의 "데이터"가 주 메모리를 가리킨다.

각 테이블 별로 정리하면 이렇게 나타낼 수 있다.[[11]](#footnote-11)

**Kernel Process Table :** 프로세스 상태, 프로세스 ID 등이 저장

**U(사용자) 영역 :** 시스템 콜 반환 코드, File Descraptor, 현재 디렉토리 및 루트 정보를 저장한다. PCB 내용 중의 swap정보를 저장하기도 한다.

**Proc Table :** 프로세스 상태에 상관없이 필요한 정보들을 저장하며 크기가 작다. Proc table entry는 process의 생성/종료까지 메모리에 상주하게 된다.

**Region Table:** 영역의 특성, 프로세스 주소 공간을 저장한다. 한 Process의 가상 주소공간의 연속된 부분으로 각각 공유되거나 보호될 수 있다.

# 

<Process Table, U Region, Kernel Process Table의 자료구조 원리>

수업시간에는 프로세스 관련 시스템 콜에 대해서 fork()함수와 exec()를 배웠고 각 함수의 특징과 data structure에 대해 정리하였다.

**fork():[[12]](#footnote-12)**

fork는 자기 자신을 복제한다. 복제 후에는 부모 프로세스와 자식 프로세스로 나뉘고 복제된 새로운 프로세스는 부모 프로세스와 같은 코드와 데이터를 가진다. fork()의 return 값은 -1이면 실패, 0이면 자식 프로세스, 그 외이면 부모 프로세스를 의미하는데 이는 자식 프로세스의 PID 값이다. fork()를 호출한 프로세스를 새로운 공간으로 전부 복사하게 되고, 원래 프로세스는 원래 프로세스대로 작업을 실행하고 fork()를 이용해서 생성된 프로세스도 그 나름대로 fork() 시스템 콜이 수행된 라인의 다음 라인부터 실행이 된다.

**exec():**

fork와 마찬가지로 모두 한 프로세스가 다른 프로세스를 실행시키기 위해 사용된다. 하지만 fork와의 차이점은 fork 시스템 콜은 새로운 프로세스를 위한 메모리를 할당한다는 것이다.

반면, exec()는 fork()처럼 새로운 프로세스를 위한 메모리를 할당하지 않고, exec()를 호출한 프로세스가 아닌 exec()에 의해 호출된 프로세스만 메모리에 남게된다. exec()실행의 결과로 생성되는 새로운 프로세스는 없고, exec()를 호출한 프로세스의 PID가 그대로 새로운 프로세스에 적용이 되며 exec()를 호출한 프로세스는 새로운 프로세스에 의해 덮어 쓰여지게 된다.

간단히 정리하면 fork()의 결과로만 프로세스가 하나 더 생긴다는 것이다.

**Chapter 4. Unix Bootstrap 과정**

fork를 통해서 프로세스는 어떻게 생성될까? 가장 초기의 시스템은 어떻게 생성되는 것일까? 하나님이 세상을 만들었다면 하나님은 누가 창조했을까 하는 질문과 비슷한 맥락일 것이다. fork의 부모를 찾아서 계속 올라가게 되면 프로세스 0이 존재하게 된다. 프로세스 0의 부모는 존재하지 않으므로 시스템에서 수동으로 생성된 것이다. 그러므로 이 때는 fork가 수행된 것이 아니게 되고 프로세스 0에서 생성된 init이라는 프로세스는 이 때 부터 fork()를 통해 여러 프로세스를 생성할 수 있는 것이다.

프로세스 컴퓨터의 power를 키면 어떤 식으로 하드웨어가 위에 운영체제가 작동하게 되는지 unix bootstrap과정을 통해 알아볼 것이다. (UNIX)

1. 디스크에 있는 bootstrap의 loader를 메모리에 loading 한다. (디스크 섹터의 가장 첫번째에 위치해있는 bootstrap loader를 메모리로 가져오는 것이다.)
2. Bootstrap은 UNIX의 이미지를 디스크로부터 메모리에 loading 한다.
3. UNIX커널은 memory mapping, register, 같은 하드웨어를 초기화 하고 커널 자료구조에 초기값을 부여해준다.
4. 프로세스의 계층구조에서 시스템의 root가 되는 값을 수동으로 형성한다. (process 0 생성)
5. 프로세스 0은 프로세스 1을 fork시키고 프로세스1을 위한 메모리영역을 커널이 확보해준다. (초기에 프로세스 1은 프로세스 0으로 부터 생성되었기 때문에 메모리 영역이 없는 상태)
6. 확보된 메모리 영역에서부터 init File을 디스크로 부터 수행시킨다.
7. 이후 프로세스 1을 init process라고 부른다.
8. 커널이 필요로하여 만든 프로세스 생성 과정을 종료시킨다.

**Chapter 5. Real-Time Scheduling**

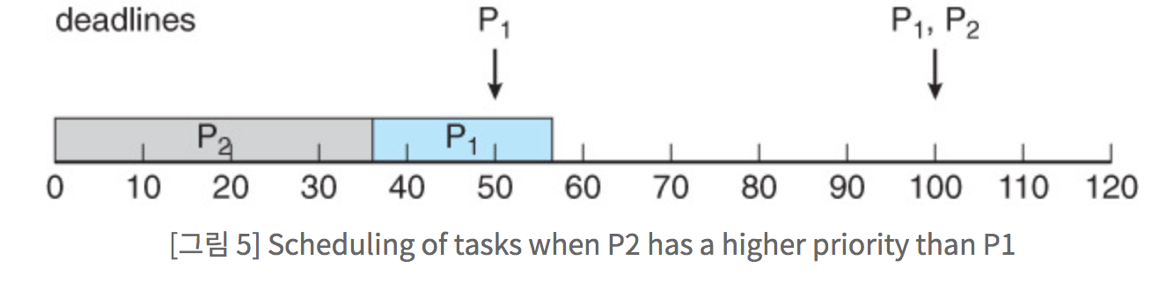
**실시간 운영체제(RTOS)**는 원하는 작업을 요구되는 시간 내에 처리할 수 있는 걸 보장하는 운영체제를 말한다. 대부분 RTOS가 들어가는 기기들은 임베디드 시스템인데, 이것들은 보통 일반 PC보다 성능이 낮은 것들이 들어간다.

RTOS를 탑재한 기기는 Performance가 일반 PC 보다 좋지 않더라도 정확한 시간 안에 끝내는 것을 보장해야 한다. [[13]](#footnote-13)그러므로 RTOS는 대체로 수 밀리초(1000분의 1초), 심지어는 마이크로초(100만 분의 1초)단위로 컨트롤하기 위해 쓰인다. 이러한 특성[[14]](#footnote-14) 때문에 엄밀한 처리가 요구되는 시스템에서 주로 사용한다. 대개 데스크톱 OS는 실행 시간이 몇 초 정도 느려져도 별 문제가 없지만, 항공 제어, 생산 관리나 병원 등 특정한 시스템에서는 단 1, 2초의 지연이 처참한 결과를 불러올 수 있기 때문이다. 실시간 운영체제는 Hard Real-Time System과 Soft Real-Time System으로 나뉜다.

**Hard Real Time**의 경우 시스템이 주어진 시간에 작업을 완료하지 못하면 안되면 큰일이 생긴다. 따라서 어떤 작업을 일정 시간 안에 반드시 처리해야 하며 그 시간이 지난 후의 결과 값은 정확하여도 소용이 없게 된다. 따라서 군사장비나 비행기 등에서 주로 사용된다.

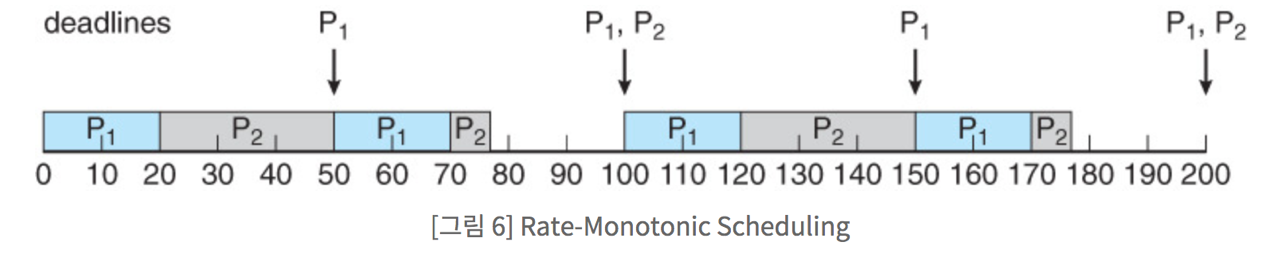
**Soft Real Time** 의 경우 Deadline시간에서 약간 경과한 값도 인정해 주는 경우이다. 물론 시간 안에 처리되는 것이 좋다. 보통 핸드폰 및 router에서 사용된다. 그렇다면 Real Time System에서Scheduling은 어떻게 적용될까? Real-time system은 task 처리에 걸리는 시간을 일관되게 유지할 수 있느냐가 중요한 성능의 척도이다. 그러므로 Real-time system의 목표는 실시간 성능 보장에 있다. 첫번째로는 DeadLIne 내에 스케쥴링이 되어야 하고 두번째로는 예측성이 보장되어야 한다. 수업시간에는 주기적 태스크 스케쥴링 알고리즘과 비주기적 태스크가 포함될 때의 스케쥴링 기법에 대해 배웠다.

**1. Periodic Task Scheduling**

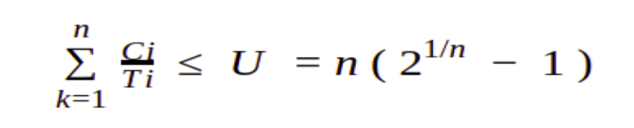
Periodic Task Scheduling 기법 중 하나로 RMS를 사용할 수 있다. Rate-Monotonic scheduling algorithm은 정적 우선순위의 선점형 알고리즘이다. 우선순위는 주기에 반비례하여 할당되며, 짧은 주기의 작업이 높은 우선순위를 받게 된다. 예를 들어, 두 개의 process P1, P2가 있는데 주기가 각각 50, 100, CPU burst가 각각 20, 35, deadline은 주기와 같다고 하자. 만약 우선순위가 제대로 지켜지지 않아서 P2가 먼저 실행된다면, P1은 deadline 내에 작업을 끝내지 못할것이다. 

[[15]](#footnote-15)

반대로, P1이 높은 우선순위를 받으면, P1이 먼저 시작되고 P1이 작업을 마치면 P2가 시작된다. P2는 35 중 30만 작업을 완료하지만 P1의 주기가 돌아와 선점된다. P1이 작업을 마치면 P2가 다시 시작되고, 나머지 작업 5를 마친다.

****

위 그림처럼 두 Process는 75에 완료되고 반복하기 전까지 25는 Slack으로 유휴 상태이다. 이 알고리즘으로 scheduling 할 수 없는 process 집합은 다른 priority scheduler로도 scheduling 할 수 없다. 따라서 정적 우선순위를 사용하는 알고리즘 중에서 Rate-Monotonic Scheduling이 가장 적합한 것으로 여겨진다. 이처럼 주기적 Task Scheduling에서 마감시간 내에 작업이 안전하게 종료할 수 있는 지의 여부를 다음과 같은 Rate-Monotonic Algorithm공식을 통해 확인할 수 있다.

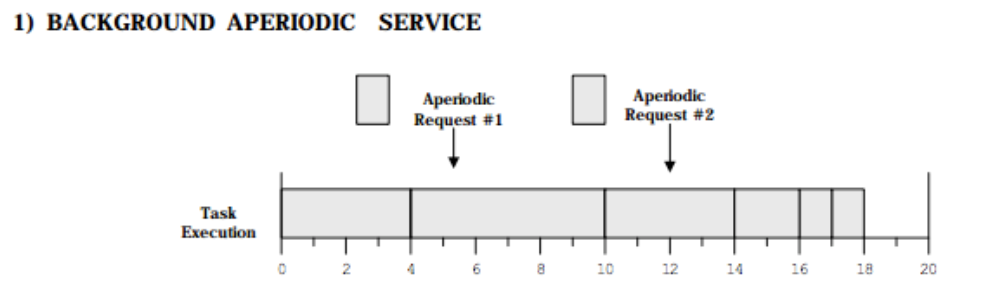
[[16]](#footnote-16)

U는 utilization을 의미하고 N은 주기적 태스크이 개수를 의미한다. C는 태스크가 수행되는 Computation time을 의미하고 T는 주기이다. 이 공식으로 주기적 task의 worst case에서 scheduling을 분석할 수 있다. (n이 무한대로 발산하면 0.69의 값으로 수렴한다.) 즉 태스크의 개수가 엄청나게 많을 때는 마감시간 내의 모든 주기적 태스크가 안전하게 보장될 때 효율성이 약 70퍼센트로 수렴하게 된다.

**1. Aperiodic Task Scheduling**

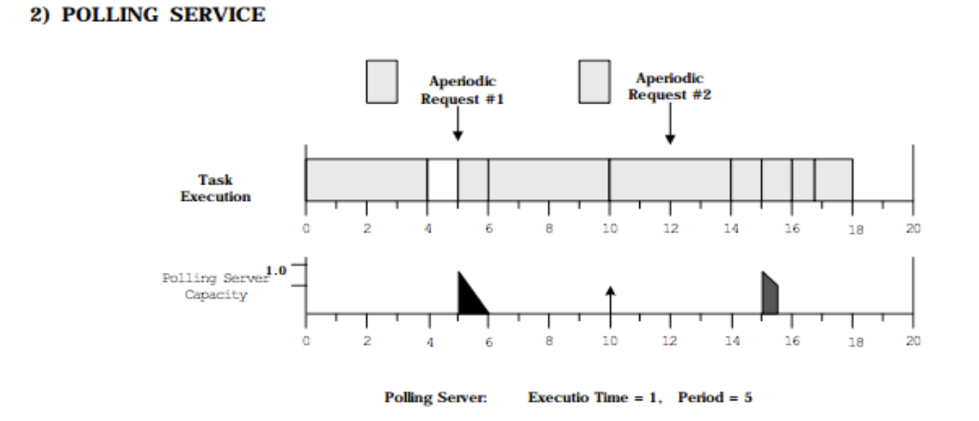
Periodic Task Scheduling 에서의 목적은 마감시간을 보장해주는 것 이였다면 Aperiodic Task Scheduling 반응시간을 빠르게 하는 것이다. 이 스케줄링 목적을 달성하기 위해서 수행되는 스케줄링 알고리즘은 3가지가 있고 과제를 통해 2가지를 실습해보았다.

**1) Background Aperiodic Service**

[[17]](#footnote-17)

간단하게 말해서 Arrival Time에 일치하는 시간에 Aperiodic task가 와도 Periodic task를 수행할 수 있다면 먼저 수행하는 방식이다. 그리고 남은 시간(slack)에 Aperiodic task를 수행하게 된다. 여기서 Aperiodic Task의 반응시간(평균대기시간)을 더 빠르게 하고싶은 방법이 대두되었다.

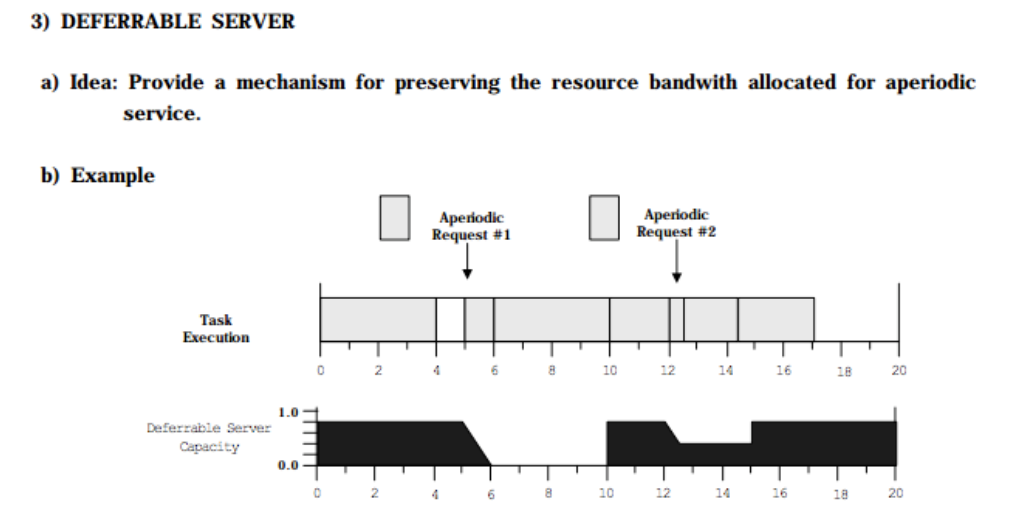
**2) Pollng Service**

****

주기적으로 Aperiodic task가 들어갈 수 있는 지를 체크하게 된다. 이 때 Polling server의 execution time이 나오게 되는데 이는 한번에 수행할 수 있는 수행 시간을 말한다. 그리고 Polling Server의 주기는 Aperiodic Task의 유무를 검사하게 되는 주기이다.

**3) Deferable Server**

서버가 항상 자기의 bandwith(capacity)를 유지하는 방식으로 스케줄링을 구현한다. 쉽게 말하면 양동이에 일정량의 물을 채워놓고 period가 될 때마다 일정량 만큼의 물을 채우는 방식이다. Polling Service는 Polling Period마다 Aperiodic Task를 체크하지만 Deferable Server의 경우 바로바로 서비스를 실행 할 수 있으므로 Polling Server보다 반응속도가 빠르다.

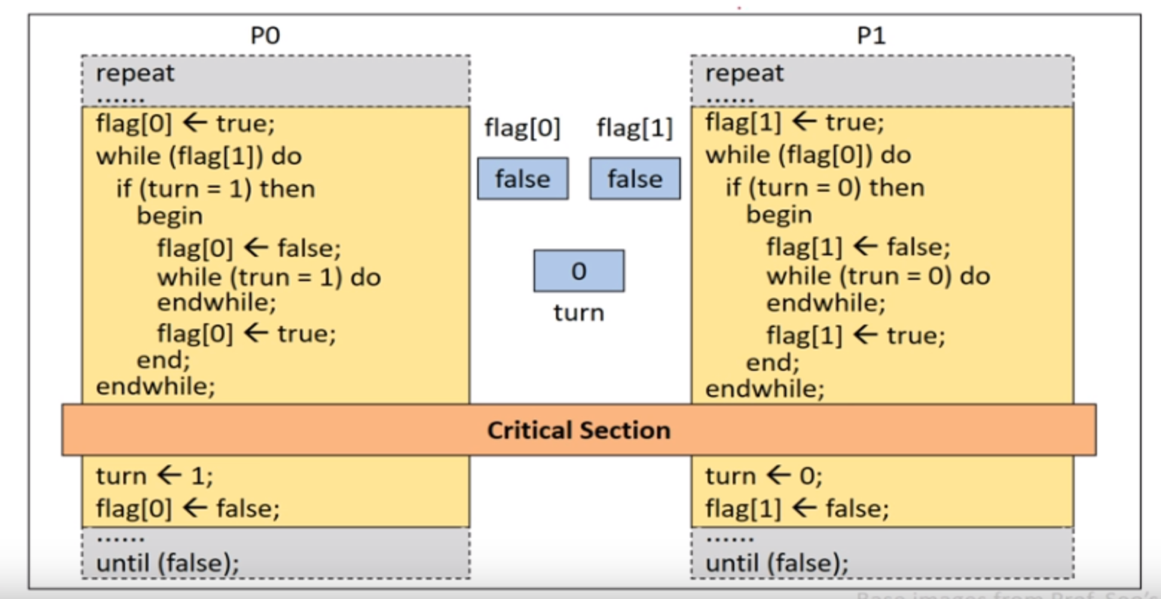


Polling Service와 Deferable Server를 사용할 경우 주의할 점은 이 때의 주기는 주어진 주기적 task들의 가장 높은 우선순위를 가진 task의 주기보다 같거나 짧아야 한다는 것이다. 즉, Polling 및 Deferable의 우선순위가 가장 높아야 한다.

# **Chapter 6. Synchronization**

서로간의 프로세스가 접속을 가능할 수 있게 되어 공유하는 데이터에 동시에 접근하면 문제점이 발생할 수 있다. 이런 문제를 해결하기 위한 것이 프로세스의 동기화이다. 서로 다른 프로세스 간에 문제가 발생하지 않도록 일관성을 유지시키는 것이 프로세스 동기화이다. 예시로 은행계좌에 두 사람이 동시에 접근을 한다고 할 때 한사람은 입금, 다른 사람은 출금을 한다고 가정해보자. 만일 이때 작업의 동기화가 제대로 이루어지지 않고 같은 시간에 접근을 하게 된다면 문제가 발생할 수 있다. 이 때 **Mutual exclusion**을 보장해야한다. Mutual exclusion이란 특정 비공유 자원을 한 순간에 한 개의 프로세스만 사용할 수 있도록 하는 것이다. 즉, 하나의 프로세스가 공유 데이터를 엑세스하는 동안 다른 모든 프로세스들이 그 데이터를 엑세스할 수 없도록 하는 것이다. Mutual Exclusion을 보장하기 위해 여러가지 솔루션이 있다.

**1. SW Solutions: Peterson’s Algorithm**, **Dekker’s Algorithm** 등이 존재하고 둘의 개념은 비슷하다. 두 개의 프로세스 기준으로 설명할 수 있다. 아이디어는 flag와 turn을 같이 쓰는 것으로 우선 임계 구역에 진입하기 위해 프로세스가 깃발을 들고 확인하고 그 다음 turn을 확인한다. 턴을 확인하고 본인 턴이 아니면 깃발을 내리고 자기 턴을 기다린다. 자기턴이 오면 그제서야 깃발을 들고 임계 구역에 진입한다. 임계 구역에서 할 일이끝나면 turn을 넘겨주고 깃발을 내리는 원리이다. SW 솔루션의 문제점은 속도가 느리다는 것이다. 그리고 다익스트라 알고리즘 같은 경우는 구현이 복잡할 수도 있다. 또한 Busy waiting(아무것도 안하고 기다리는 데 실행은 계속하는 중인 상태)가 발생할 수 있다는 것이다.

[[18]](#footnote-18)

< 두 개의 프로세스에서 Mutual Exclusion을 보장하기 위한 SW Solution>

**2. HW Solutions: TestAndSet(TAS)** instrunction으로 구성된다. 하드웨어 명령어의 중요한 특징은 Interrupt되지 않고 프로세스가 하나의 단위처럼 처리된다는 것이다. 따라서 임계 구역 문제를 보다 쉽고 효율적으로 해결할 수 있다. Test-and-set의 명령어는 다음과 같이 정의될 수 있다.

**function** TestandSet(Boolean \*target) {         

**Boolean** temp = \*target;

\*target = true;

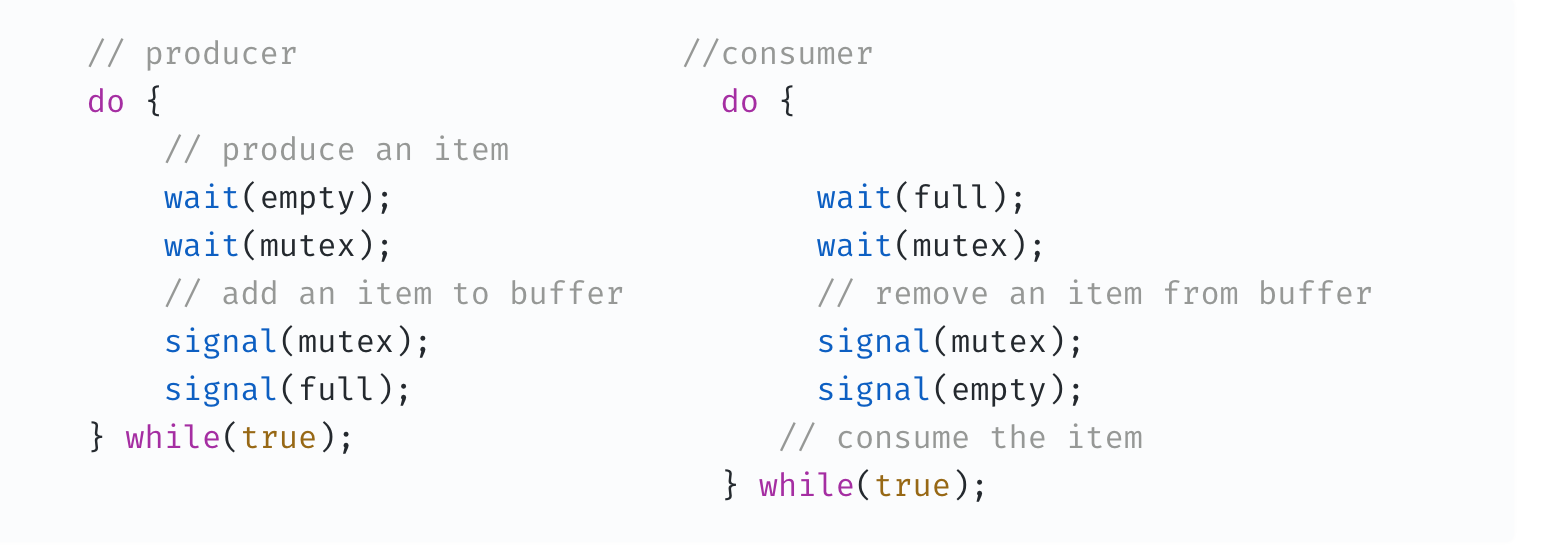
return temp;

}

target이라는 값이 있고 이를 잠시 temp에 넣고 target을 true로 하고 temp를 리턴한다. target의 현재 값을 반환하면서 target 값을 바꾼다는 것을 한 번에 수행한다는 것을 유의해야 한다. 이렇게 하면 간단하게 Mutual Exclusion을 보장할 수 있다. lock이라는 변수를 하나 더 두어서 lock의 초기값이 false이면 TAS를 거치면 return은 false가 되고 lock은 true가 된다. 이 상태에서 다른 프로세스가 들어가려 하면 True인 상태이니 임계 구역에 들어가지 못한다. HW Solution의 장점은 구현이 간단하지만 SW와 마찬가지로 Busy waiting이 발생한다는 것이다. 이 문제를 해결하기 위해 Semaphore가 등장하였다.

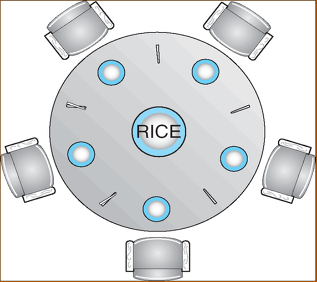
**3. Semaphore :** 세마포(Semaphore)는 동기화 문제(임계구역 문제) 해결을 위한 소프트웨어 도구이다. 구조적으로 정수형 변수와 두 개의 동작(P,V )으로 구성되어 있다. 두 개의 동작은 정수 값을 증가시키거나(V동작) 검사하는 역할(P동작)을 한다. P동작을 wait()라고 하고 V동작을 signal()이라 한다.  P동작을 하게 되면 정수 값이 1만큼 감소되고 값이 0보다 작게 되면 세마포에 존재하는 리스트(Queue)에 쓰레드를 넣고 block시킨다. V동작은 정수 값을 1 증가시키고 리스트에서 쓰레드를 방출한다. 세마포는 메인 메모리에서 쓰레드가 동작할 때 갈 수 있는 또 다른 루트인데 동기화를 위해 존재하는 것이다. 여기서 정수 값은 권한의 개수라고 할 수 있다. 세마포는 동기화의 조건 중에서 상호배타의 역할을 위해서 존재한다. 또한 세마포는 프로세스 실행 순서를 원하는 대로 제어하는 역할로도 사용할 수 있고 이를 ordering이라 한다.

**3\_1) Producer-Consumer Problems:** 생산자-소비자 문제는 생산자가 데이터를 생상하면 소비자는 그것을 소비하는 형태에서 발생하는 문제를 말한다. 생산자가 데이터를 생산하면 버퍼에 보관을 하게 되고 소비자는 버퍼에서 데이터를 빼내어 사용한다. 하지만 현실 시스템에는 버퍼의 크기가 유한하다. 크기가 정해져 있는 버퍼이기 때문에 Bounded Buffer라고 불린다. 생산자는 버퍼가 가득 차면 더 이상 넣을 수가 없다. 반대로 소비자는 버퍼가 비면 뺄 수가 없다. 생산자와 버퍼를 제어하는 세마포는 정수 값의 초기 값을 size 값으로 가지게 된다. 데이터가 계속 버퍼에 들어가는데 버퍼가 size만큼 들어가게 되면 생산자는 block이 되어야한다. 반대로 소비자의 세마포는 비어 있을 때 block되어야 하므로 정수 값의 초기 값으로 0을 가진다. 버퍼에 데이터가 들어오면 정수의 값을 증가시키고 데이터가 없는 빈 버퍼가 되었을 때 acquire()명령으로 데이터를 빼내오려고 한다면 정수 값이 0보다 작은 값을 가지게 되므로 소비자 쓰레드를 block 시킨다. 두 개의 세마포는 release를 명령을 보내게 되면 상대방의 세마포를 풀어주어 동작을 할 수 있게 한다. 이렇게 제어를 하게 되면 CPU가 무한 루프를 돌고 있는 작업을 하는 것을 막을 수 있다.

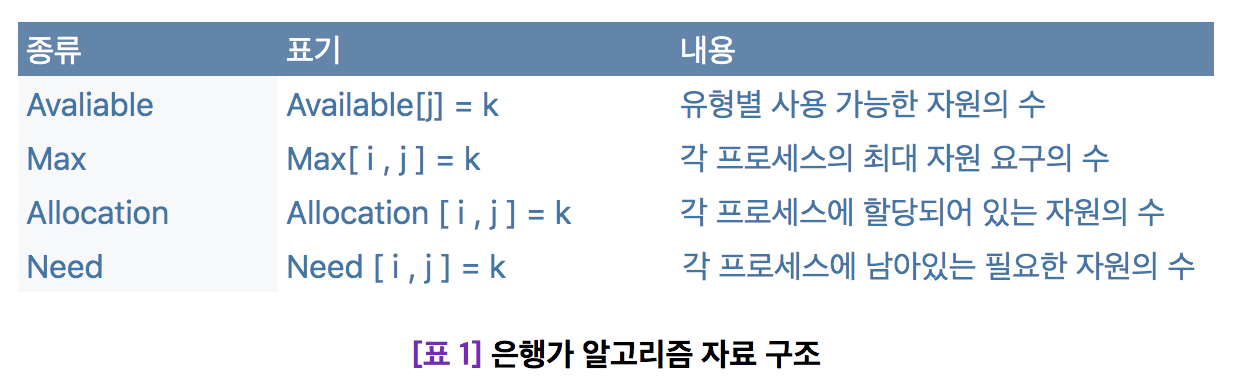


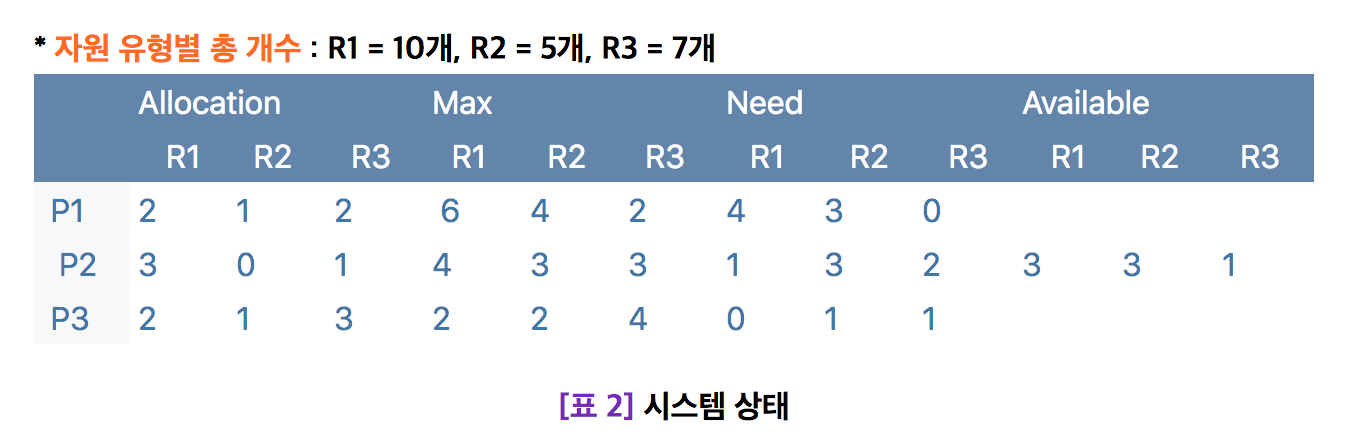
< 생산자 소비자 문제를 해결하기 위한 Semaphore Solution>

**3\_2) Dining-Philosophers Problems:** 5명의 철학자가 식사를 하고 있는 상황을 가정한 문제이다. 철학자들이 원형의 테이블에 앉아 있는데 철학자들 사이에는 젓가락이 놓여있다. 하지만 젓가락이 한 쌍이 놓여 있는 게 아니라 하나가 놓여있는 상황이다.



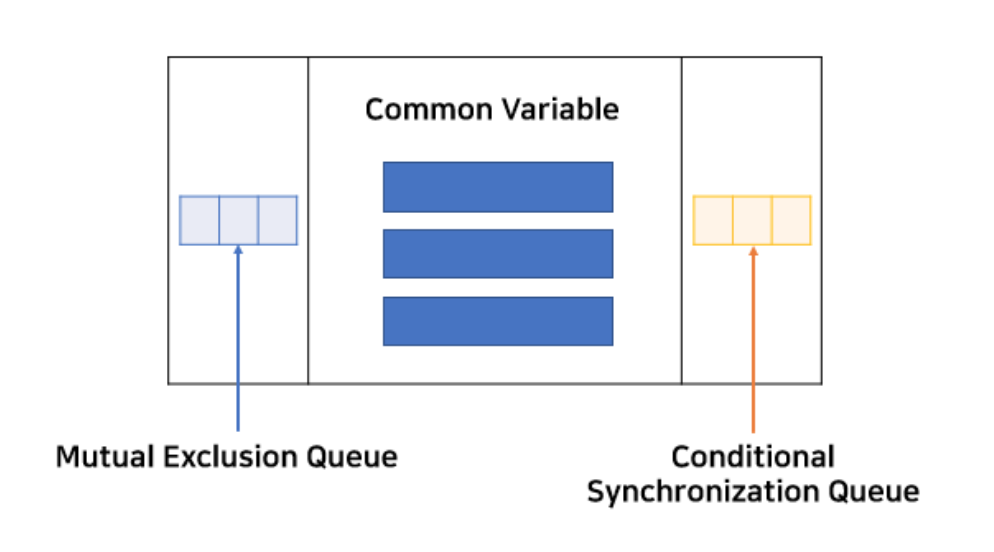
[[19]](#footnote-19)철학자에 대한 정보는 쓰레드와 같다. 젓가락에 대한 접근은 양 옆 철학자에 대해 공통적인 데이터를 접근하는 것과 같다. 임계구역에 접근하는 것과 같아 한 명의 철학자가 젓가락 사용, 즉 임계구역에 접근을 하면 다른 철학자는 그 구역에 접근을 할 수 없다. 그러면 이와 같은 상황을 세마포를 이용해서 상호배타를 만족시키는 상황을 만들 수 있을 것이다.  하나의 젓가락에 대한 세마포는 정수 값의 초기 값으로 1을 가진다. 철학자 한 명이 접근을 하면 다른 철학자가 접근을 하지 못하므로 1명만 접근을 하게 만들기 위해서이다. 철학자라는 쓰레드는 자신의 고유번호를 가지고 왼쪽 젓가락과 오른쪽 젓가락을 가질 수 있다.  만약 왼쪽 젓가락과 오른쪽 젓가락을 모두 가지게 되면 식사를 하는 동작을 수행하고 그 후 젓가락을 놓고 생각을 하는 동작에 빠지게 된다. 하지만 위와 같은 코드를 실행하면 문제가 발생한다. 바로 모든 철학자가 식사를 하지 못해 굶어 죽는 상황인 starvation인 상태가 발생할 수 있기 때문이다 이런 상태를  Deadlock이라고 한다. 동기화를 세마포를 통해 설계를 하였는데 교착상태에 빠져 모든 자원이 임계구역에 접근을 하지 못하게 되는 것이다. Deadlock은 **Banker’s Algorithm**을 통해서 회피 할 수 있다. 교착상태에 빠질 가능성이 있는지 판단하기 위해 상태를 안전상태, 불안전상태로 나누는 것이다. 즉, Banker’s Algorithm에서 운영체제는 안전상태를 유지할 수 있는 요구만을 수락하고 불안전 상태를 초래할 사용자의 요구는 나중에 만족될 수 있도록 하는 것이다.





3개의 프로세스와 자원 R1= 10, R2 = 5, R3 = 7이라고 할 때 표2의 시스템 상태를 보면 이 시스템은 안전 상태라고 판단할 수 있다. 그 이유는 현재 Avaialble 자원이 [3,3,1]인데 p1, P2에는 할당할 수 없지만 P3 프로세스에 할당하여 할당한 자원을 해제하면 Avaliable 자원은 [5,4,4]가 되기 때문이다. 따라서 순서를 [P3, P2, P1] 로 하게 되면 안정 조건에 해당하는 것이다.

**4. Monitor:** 동기화 문제를 해결하기 위해서 우리는 세마포라는 도구를 사용하였다. 하지만 동기화 문제를 해결하는데 세마포만이 사용되지는 않는다. 사실 세마포의 경우 오래된 동기화 도구라고 할 수 있다. 현재 사용되는 도구 중 하나가 모니터이다. 세마포가 어셈블리 언어에 적합한 도구라면 모니터는 그보다 고수준인 언어의 도구라고 할 수 있다. 공유자원과 공유자원에 대한 접근함수가 존재한다. 이러한 구역을 임계구역이라고 한다. 모니터의 경우 두 개의 queue가 있는데 각각 배타동기와 조건동기의 역할을 한다. 배타동기의 queue는 하나의 쓰레드만 공유자원에 접근할 수 있게 하는 작용을 하는 공간이다. 특정 쓰레드가 공유자원을 사용하는 함수를 사용하고 있으면 다른 쓰레드는 접근을 할 수 없고 대기해야 한다. 조건 동기의 queue는 진입 쓰레드가 블록되면서 새 쓰레드가 진입가능하게 하는 공간이다. 새 쓰레드는 조건동기로 블록된 쓰레드를 깨울 수 있다. 깨워진 쓰레드는 현재 쓰레드가 나가면 재진입할 수 있다. 아래 그림은 모니터의 구조를 간단히 나타낸 그림이다. 모니터는 공유 자원 + 공유 자원 접근함수 로 이루어져 있고, 2개의 큐를 가지고 있다. 각각 mutual exclusion(상호배타) queue, conditional synchronization(조건동기) queue 이다.

[[20]](#footnote-20)

사진까지 첨부하려니 20장이 넘어버린 점 양해 부탁드립니다.

1. <https://ko.wikipedia.org/wiki/%EC%9A%B4%EC%98%81_%EC%B2%B4%EC%A0%9C> 운영체제 [↑](#footnote-ref-1)
2. <https://100100e.tistory.com/126> 운영체제의 kernel과 shell [↑](#footnote-ref-2)
3. 운영체제 Chapter 01. 조진성 교수님 운영체제 강의 [↑](#footnote-ref-3)
4. <https://jongmin92.github.io/2019/02/18/Programming/computer-structure/> 컴퓨터구조 [↑](#footnote-ref-4)
5. <https://m.blog.naver.com/PostView.nhn?blogId=sillllver&logNo=90125624751> 선점형 스케쥴러 구현 [↑](#footnote-ref-5)
6. <https://jaebworld.tistory.com/27> 폴링 & 인터럽트 [↑](#footnote-ref-6)
7. 시스템 내 데이터의 집중적인 사용으로 인해 전체 시스템에 절대적 영향을 미치는 부분의 사용빈도가 늘어나 성능이 저하되는 현상 [↑](#footnote-ref-7)
8. <https://frontjang.tistory.com/132> 파일 디스크립터 테이블 [↑](#footnote-ref-8)
9. <https://en.wikipedia.org/wiki/File_descriptor> 위키피디아 파일 디스크립터 [↑](#footnote-ref-9)
10. <http://yimoyimo.tk/OS-Process-Creation-and-Termination/> 프로세스 생성 방법 [↑](#footnote-ref-10)
11. <https://wuqingjun.wordpress.com/2010/03/23/the-design-of-unix-operating-system-2-kernel/> Design of unix operating system [↑](#footnote-ref-11)
12. <https://sbell92.tistory.com/21> 프로세스의 생성 [↑](#footnote-ref-12)
13. <https://namu.wiki/w/>실시간운영체제 [↑](#footnote-ref-13)
14. [↑](#footnote-ref-14)
15. <https://lazymankook.tistory.com/49> Real-Time Scheduling [↑](#footnote-ref-15)
16. <https://www.geeksforgeeks.org/rate-monotonic-scheduling/> Rate Monotonic Schedulin [↑](#footnote-ref-16)
17. <https://wnsdud1861.tistory.com/9> Realtime system Aperiodic Scheduling Algorithm [↑](#footnote-ref-17)
18. <https://ggodong.tistory.com/97> Mutual Exclusion [↑](#footnote-ref-18)
19. <https://copycode.tistory.com/71?category=740133> Deadlock [↑](#footnote-ref-19)
20. <https://codemcd.github.io/study/OperatingSystem-11%EC%9E%A5-%EB%AA%A8%EB%8B%88%ED%84%B0/> 모니터 [↑](#footnote-ref-20)