OS (운영체제)

- 컴퓨터라는 H/W 상에 애플리케이션들이 동작되려면 CPU, 메모리, 디스크 등의 H/W 들과 적절하게 데이터를 주고 받으며 논리적인 일들을 해야 하는데, 이러한 응용 프로그램들을 수행하기 위해서 H/W 와의 인터페이스를 제공하는 기본 프로그램을 '운영체제' 라고 한다.
- 운영 체제는 CPU 의 수행 시간을 나누는 '프로세스 관리'와 서로 다른 애플리케이션이 메모리를 사용하는데 있어서 문제가 없도록 해주는 '메모리 관리', 파일이 저장되어 있는 '디스크 관리' 라는 3대 기능을 가지고 있다.
    - CPU 관리
        - 스레드(Thread)는 각 프로세스(Process)에 한 개 이상 존재하고, 그 프로세스가 포함하고 있는 수행부를 한개씩 수행하는 역할 (운영체제 상에서 스레드의 개수는 프로세스 갯수 이상)
        - 여러 개의 스레드가 동시에 CPU를 사용하려 요청하는데, 이 때 효율적으로 CPU를 사용하게끔 하는 것이 프로세스/스레드 관리 혹은 CPU 관리 라고 한다.
        - 멀티 태스크(Multi-Task) 운영체제일 경우에 여러 개의 어플리케이션이 마치 동시에 동작되는 것처럼 보이게 하는 것은 프로세스/스레드라 불리는 것을 시분할로 조금씩 번갈아 가며 CPU가 수행을 해주기 때문에 가능하다.
        - 프로그램이 수행된다는 의미는 디스크에 있는 파일이 메모리 상으로 로드가 되고, CPU가 그 명령어를 수행하는 것을 의미한다.
        - 컴퓨터에는 기본적으로 여러 개의 프로세스가 실행되고 있으며 이는 해당 명령어가 CPU에 의해서 수행이 된다는 것이고, 이를 수행해주는 역할을 하는 개체가 '스레드'이다.
        - 여러 개의 스레드가 순차적으로 번갈아가며 수행이 되는데, 기존에 수행되고 있던 것은 수행을 멈추고 다음 스레드에게 CPU 사용권을 넘겨줘야 한다. 이때, 현재 수행되고 있는 레지스트리나 CPU 내의 값의 정보를 남겨둬야 하는데, 이 상태를 'context' 라고 하고 이것을 남기고 다음 환경으로 복원하는 과정을 'Context Switching'이라고 한다.
            - 즉 'Context Switching' 이란 스케줄링에 따라 프로세스를 준비에서 실행으로, 실행에서 대기로, 실행에서 준비로, 실행에서 종료로 같은 상태 변경에 필요한 내용을 PCB에 저장 및 로드 하는 과정을 뜻한다.
    - 메모리 관리
        - 가상 메모리
            - 프로그램이 바라볼 때는 가상(논리)주소로 보고 실제 메모리를 접근할 때에는 실제 주소로 대응
            - 실제 메모리를 사용하다가 사용하지 않는 데이터를 디스크에 저장하게 되고 수행을 할 때에는 실제 메모리로 로드하게 된다.
            - 이러한 동작의 반복은 하드웨어와 운영체제간의 동작이며 애플리케이션 측면에서는 메모리를 사용하는 것과 동일한 상태로 보여진다.
    - 디스크 관리
        - 디스크는 전원이 꺼진 상태에도 내용이 유지되는 저장장치 이다.
        - 하드 디스크의 기본 단위는 'Sector' 라는 단위의 집합체로써 섹터 번호를 지정하여 데이터를 읽고 쓰게 된다.
        - 운영체제는 이를 파일이라는 논리적 단위로서 제공하여 데이터를 읽거나 쓸 수 있도록 제공한다. (=파일 시스템)
        - 윈도우 시스템은 파일 시스템으로 FAT 시스템 가 NTFS 의 종류가 있다.

1. Interpreter 와 compile 언어에 대해 설명하시오

- **인터프리터 언어**는 프로그래머가 작성한 소스코드(원시코드)를 바로 기계어로 변환하는 과정없이 한 줄씩 해석하여 명령어를 실행하는 언어이다. R, Python, Ruby와 같은 언어들이 대표적이다.

인터프리터가 직접 한 줄씩 읽고서 기계어로 따로 변환하지 않기 때문에 빌드 시간이 없다. Runtime 상황에서는 한 줄씩 실시간으로 실행하기 때문에 컴파일 언어에 비해 속도가 느립니다.

실행 속도가 느리지만, 코드 변경 시에 빌드 과정이 없이 바로 프로그램을 실행하는 장점이 있습니다.

- **컴파일 언어**는 프로그래머가 작성한 코드를 모두 한꺼번에 기계어로 변환한 후에 기계(JVM 같은 가상 머신)에 넣고 기계어 코드를 실행합니다. C와 Java가 대표적이다.

  이런 변환 과정을 빌드 과정이라고 하고, 이 때문에 인터프리터 언어에 비해 실행에 시간이 소요가 됩니다.

  하지만, 런타임 상황에서는 이미 기계어로 모든 소스코드가 변환 되어 있기 때문에 빠르게 실행할 수 있습니다.

> **런타임**이란? 컴퓨터 프로그램이 실행되고 있는 동안의 동작.
>
> 즉, 컴퓨터 내에서 프로그래밍이 기동 되면, 그것이 바로 프로그램의 런타임이다.
>
> 어떻게 보면, 프로그래밍 언어가 구동되는 환경이라고 이해할 수도 있다.
>
> 예로, JavaScript 라면 Web Browser에서 작동하는 JavaScript 측면이 있고,
>
> Node.js라는 환경에서 구동되는 측면도 있다. 여기에서의 Browser와 Node.js를 런타임이라고 볼 수 있다.

2. Process 와 Thread에 대해 설명하고, 둘의 차이를 설명하시오

- **프로세스**는 운영체제가 메모리 등의 필요한 자원을 할당한 실행중인 프로그램을 뜻한다.
- 즉, 프로그램이 실행 중인 상태로 특정 메모리 공간에 프로그램의 코드가 적재되고 CPU가 해당 명령어를 하나씩 수행하고 있는 상태를 의미한다.

  이때, 각각의 프로세스는 서로 메모리 공간을 독자적으로 갖기 때문에 서로 메모리 공간을 공유하지 못한다. 따라서, 다른 프로세스의 메모리에 접근하려면 IPC(Inter Process Communication)과 같은 방식이 필요하다.

> **Multiprocessing** : 여러 개의 프로세스를 동시에 처리하는 것, Task를 실행하는 core (CPU core, processor) 가 2개 이상인 경우 (동시에 여러가지 일을 수행할 수 있을 때)

> **Multitasking** : 동시간에 여러 개의 프로그램을 띄우는 것, 하나의 core가 시분할(time slicing) 기법을 이용해서 여러 개의 Task를 마치 동시에 수행되는 것처럼 보이게 하는 기법

- **스레드**는 프로세스 내에서 실행되는 각각의 일을 말한다. 결국, 프로세스 내에서 실행되는 여러 개의 스레드가 하나의 프로세스를 이루게 되는 것이다. (즉, 운영체제에서 프로세서 시간을 할당하는 기본 단위)

  스레드는, 프로세스 내에서 그 프로세스의 자원을 이용해서 실제로 작업을 한다. 따라서, 스레드가 포함된 프로세스가 운영체제로부터 자원을 할당 받으면 그 자원을 사용하여 작업을 처리한다. 즉, 쓰레드는 같은 프로세스에 있는 자원과 상태를 공유

  각 스레드는 독자적인 Stack메모리를 갖는다. 스레드는 메모리를 공유하기 때문에, 동기화(synchronize) 및 데드락의 (Deadlock) 문제가 발생 할 수 있다.
- 하나의 프로세스 안에서 여러 개의 루틴을 동시에 수행하여서 수행 능력을 향상시키려고 할 때 쓰레드를 사용한다.

> **Multithread** : 프로세스가 둘 이상의 스레드를 가지고 일을 수행하는 것, 하나의 Task를 여러 개의 sub Task로 분할해서 동시간에 실행되는 것처럼 수행하는 기법, 여러 개의 스레드를 하나의 프로세스 내에서 수행하는 것

> 예로, 1부터 1000까지 더하는 작업의 경우, 단일 스레드(single thread)는 순차적으로 일을 처리하므로, 1부터 1000까지 더하는 작업이 진행될 동안, 이와 독립적은 다음 작업을 실행할 수가 없다. 따라서, 더하는 작업을 multithread방식으로 구현하면, 1~1000까지 더하는 작업과 동시에 다음 작업을 진행할 수 있다.

- 결국 **process 와 thread의 가장 중요한 차이**는 자원을 공유하는 방식에 있다.

  프로세스는 운영체제로부터 자원을 할당 받는다. 따라서, 각 작업(Task)마다 운영체제로부터 자원을 할당 받기 위해 시스템 콜(call)을 하는 부담이 존재한다. 이 상황에서, 멀티스레드는 시스템 콜(call)을 한번만 해도 되기 때문에 효율적이다. 한 process 내에서 자원을 공유하는 것이기 때문이다.

  또한, IPC 방식보다는 스레드 간 통신이 덜 복잡하고 시스템 자원 사용이 더 적다. 따라서 통신의 부담이 더 적다. 하지만, 멀티스레드 방식은 동기화 작업을 따로 해줘야 하기 때문에 제어가 어렵다는 단점이 존재한다.

  멀티 프로세스 구조에서 여러 개의 자식 프로세스 중 하나에 문제가 발생하면 자식 프로세스 하나만 죽는다 해서 다른 곳에 영향을 끼치지는 않는다.

  멀티 스레드 구조에서는 자식 스레드 중 하나에 문제가 생긴 경우에는 전체 프로세스가 영향을 받게 된다.(ex : thread I/O)

> 포트(port)란? 서버 내에서 프로세스를 구분하는 번호
>
> - 서버는 http 요청을 대기하는 것 외에도 데이터베이스 통신 및 FTP 요청을 처리하기도 하는 등 다양한 작업을 한다. 이럴 때, 서버는 프로세스에 포트를 다르게 할당하여 들어오는 요청을 구분한다.
> - 유명한 포트 번호로는 21(FTP), 80(HTTP), 443(HTTPS), 3306(MYSQL) 가 있다.
> - 80번 포트를 사용하면 url 주소에서 포트를 생략할 수 있다.
>
> 출처 : 책 Node.js 교과서

3. Deadlock과 Starvation에 대해서 설명하시오

- 데드락은 프로세스가 자원을 얻지 못해서, 다음 작업을 처리하지 못하는 생태입니다. 이는 시스템적으로 한정된 자원을 여러 곳에서 동시에 사용하려고 할 때 발생합니다.

  주로, 멀티프로세스나 멀티스레드 환경에서 여러 프로세스나 스레드들이 한정된 자원을 공유하여 사용하기 때문에 발생합니다.

  예로, A라는 프로세스가 B가 가진 자원이 있어야 동작이 가능해서 B의 작업이 끝나서 자원을 넘겨주기를 기다리고 있다. 근데, 동시에 B라는 프로세스 역시 A가 가진 자원이 있어야 해서 A의 작업이 끝나기를 기다리고 있는 상태면, 무한 교착 상태에 빠진다. (A와 B 작업 모두 끝이 날 수 없다.) -> 서로 상대방의 작업 끝나기만을 기다리는 상태

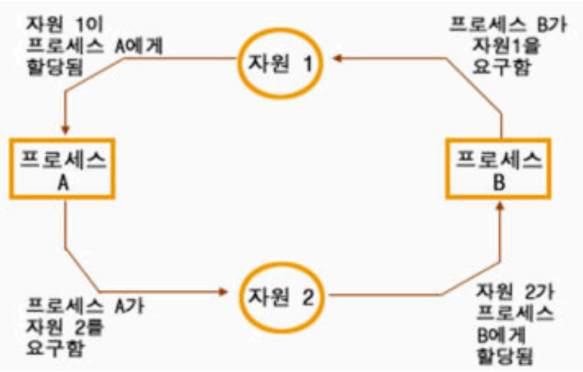


그림 9-3. 교착 상태

- 데드락 발생 조건 4가지

  1) **상호 배제 (Mutual Exclusion, Mutex)** : 공유 불가능한 자원의 동시 사용을 피하기 위해 사용되는 로직, 임계 구역(critical section)으로 불리는 코드 영역에 의해 구현된다. 즉, 공유 자원을 어느 시점에서 오직 한 개의 프로세스만이 사용할 수 있도록 하는 조건

  2) **점유 대기 (Hold and Wait)** : 프로세스가 할당된 자원을 가진 상태에서 다른 자원을 요청하고 기다리는 것

  3) **비선점 (No preemption)** : 프로세스가 어떤 자원의 사용을 끝낼 때까지 다른 프로세스가 그 자원을 뺏을 수 없는 것

  4) **순환 대기 (Circular wait)** : 각 프로세스가 순환적으로 다음 프로세스가 요구하는 자원을 가지고 있는 형태

> 기아상태(starvation)은 특정 프로세스의 우선순위가 낮아서 원하는 자원을 계속 할당 받지 못하는 상태를 말한다.

> 교착상태는 여러 프로세스가 동일 자원의 점유를 요청할 때 발생하고, 기아상태는 여러 프로세스가 부족한 자원을 점유하기 위해 경쟁할 때, 특정 프로세스는 영원히 자원이 할당이 안되는 경우를 의미한다.

교착상태와 무한연기의 비교

구분	교착상태	무한연기
내용	여러 개의 프로세스가 아무 일도 못하고 어떤 특정 사건을 기다리며 무기한 연기되어 있는 상태	특정 프로세스가 자원을 할당 받기 위해 무한정 대기
원인	상호배제, 점유와 대기, 비선점, 환형대기	자원의 편중된 분배정책으로 발생
해결책	예방, 회피, 발견, 복구	노화기법(Aging)으로 해결

표 9-8. 교착 상태와 무한 연기 비교

4. Deadlock 의 예방(Prevention)법에 대해 설명하시오

교착상태 예방(Prevention)

교착상태 발생 조건 중 1개만 부정하는 방법으로 교착상태의 발생 가능성을 사전에 모두 제거하도록 시스템을 조절한다. 가장 명료하고 널리 사용되는 방법이나 자원의 낭비가 발생한다.

구분	특징	단점
점유와 대기조건의 부정	✓ 프로세스는 필요한 모든 자원을 한꺼번에 요청하고, 시스템은 요청된 자원들을 전부 할당하든지 전혀 할당하지 않는 방식	✓ 자원 낭비와 비용 증가 ✓ 자원 공유 불가능 ✓ 무한연기 발생 가능
비선점 조건의 부정	✓ 어떤 자원을 가지고 있는 프로세스가 더 이상 자원 할당 요구가 없으면 가지고 있던 자원을 반납하고, 필요 시 다시 자원을 요구하는 방법	✓ 비용 증가 ✓ 일부 자원은 선점 불가 경우 발생 ✓ 무한연기 발생 가능 ✓
환형대기 조건의 부정	✓ 모든 프로세스에게 각 자원의 유형별로 할당순서(고유번호)를 부여하는 방법 ✓ 시스템 설치 시 모든 자원에게 고유번호 부여 ✓ 자원 요청 시 번호 증가순으로 요청	✓ 새로운 자원 추가 시 재구성 ✓ 프로세스 실행 중 예상한 순서와 다른 요구조건 발생하는 경우 긴 시간 동안 자원 낭비 ✓ 급한 프로그램 발생시 자원 할당의 어려움
상호배제 조건의 부정	✓ 공유할 수 없는 자원을 사용할 때 성립. 기억장치, CPU 등	✓ 비 공유 전제

표 9-10. 교착 상태 예방 방법

5. Deadlock 의 회피(Avoidance)법에 대해 설명하시오

> 교착상태의 발생 가능성을 인정하고 교착생태가 발생할 때 이를 적절히 피해가는 방법

- 대표적인 방법 : Banker's Algorithm
  - 운영체제는 자원의 상태를 감시한다. 사용자 프로세스는 사전에 자기작업에서 필요한 자원의 수를 제시한다. 운영체제는 사용자 프로세스로부터 자원의 요청이 있으면 모든 프로세스가 일정 기간 내에 성공적으로 끝날 수 있는 안정상태인지를 면밀하게 분석한다.
  - 운영체제는 안전 상태를 유지할 수 있는 요구만을 수락하고, 불안전 상태를 초래할 사용자의 요구는 나중에 만족될 수 있을 때까지 계속 거절한다.
  - 할당할 자원량이 일정량 존재해야하며 최대 자원 요구량을 미리 알아야 한다. 일정한 수의 사용자 프로세스가 존재할 때만 적용 가능하고 프로세스들은 유일한 시간 내에 할당된 자원을 반납해야 하는 문제점이 있다.

6. 그 외 Deadlock 관련 알고리즘
   - 교착상태 발견 (Detection)
     - 일단 교착상태가 발생하도록 허용하고, 교착상태가 발생하면 그에 관련된 프로세스와 자원을 조사하여 결정해 내는 방법
   - 교착상태 복구 (Recovery)
     - 교착상태에 빠진 프로세스를 식별하고 그 프로세스를 다시 시작하거나 되돌림으로써 해결하는 방법
7. Thread 들이 자원을 어떻게 공유하는지 설명하시오

- 멀티스레드 방식에서 만나게 되는 문제는 동기화(Synchronization)다. 왜냐하면, 스레드는 별도로 실행되는 하나의 실행이므로 동시에 여러 스레드가 진행되면 스레드들이 공유하는 자원이 동기화된 상태로 공유될 수 없기 때문이다. 만약, 각 스레드들이 서로가 침범하는 영역이 없다면 동기화 문제를 고려하지 않아도 된다.

  프로세스들 간에는 이미 OS가 별도의 프로그램으로 잘 동작하도록 알아서 처리를 해주고 있다. 왜냐하면, 프로세스는 자신만의 메모리 영역이 존재하고, 다른 프로세스의 메모리 영역을 침범할 수가 없기 때문이다. 하지만, 한 프로세스 안에서 실행되는 여러 개의 스레드들은 서로 공유할 수 있는 메모리 부분이 있다. 그래서 동기화 문제가 발생한다.

> Ex) 10만원이 있는 한 계좌에서 A가 10만원을 뺏고, ATM기기가 계좌 잔고를 0원으로 바꾸려고 할 때, interrupt 가 걸린 상태에서, B가 다른 기기에서 동일 계좌로부터 10만원을 빼려고 할 때, 잔고 부족으로 안되어야 하는데 동기화 작용을 안하면 10만원이 남은 줄 알고 돈이 빠진다.
>
> Race Condition : 이렇게 두 개 이상의 스레드들이 하나의 자원을 이용하려 경쟁하는 상황을 뜻

- 동기화 문제는 스레드들간의 공유 자원에 접근하는 것을 한번에 한 스레드만 가능하게 함으로써(상호 배제, Mutual Exclusion) 해결할 수 있다.

> 한 번에 하나의 스레드만 임계 영역(critical section)에 있는 객체에 접근할 수 있도록 객체에 락(lock)을 걸어서 데이터의 일관성을 유지

> **lock 방식** 뿐 아니라 동기화 문제를 해결하기 위한 방법으로 **semaphore, monitor** 등이 있다.

5. 유저모드에서의 동기화 기능 객체들
   - Critical Section
     - 유저 레벨 동기화 방법 중 유일하게 커널 객체를 바로 사용하지 않으며, 내부 구조가 단순하기 때문에 동기화 처리를 하는 데 있어 속도가 빠르다는 장점이 있으나 동일한 process 내에서만 사용할 수 있다는 제약이 있다.
     - 프로세스 하나에 포함된 여러 개의 스레드가 공유 리소스에 접근할 때 베타적 제어를 하기 위한 구조이다. (동기화하기를 원하는 구역의 시작과 끝에 API 함수를 호출하는 방식)
       EnterCriticalSection()
       ..// 파일쓰기 등의 동기화 필요한 작업
       LeaveCriticalSection()

       > 베타적 제어? 임의로 정의된 구역 내를 접근할 때에 한 쓰레드가 소유권을 받게 되고 그 쓰레드가 구역을 통과할 때 소유권을 반환
   - 동기화 객체
     - 동기화를 위한 객체로 뮤텍스, 세마포, 이벤트와 같은 객체가 있고, 프로세스, 쓰레드, 파일 들까지도 동기화를 위한 커널 객체를 포함하고 있다.
     - 커널 객체의 경우는 Signaled 와 Nonsignaled의 2가지 상태 중 하나로 존재하며, 동기화 객체가 Signaled될 때까지 이 커널 객체를 사용하려는 쓰레드는 대기하고 있게 된다.
     - 크리티컬 섹션과의 큰 차이는 동기화 객체는 모두 프로세스 간의 동기화에 사용할 수 있다는 점이다.

동기화 객체	용도
뮤텍스	리소스 배타적 제어
세마포어	리소스를 동시에 사용할 수 있는 개수 제어
이벤트	다른 쓰레드에 이벤트 통지

표 9-3. 동기화 객체의 종류

- 뮤텍스 (Mutex)
  - 최초에는 Signaled 상태로 생성되고, WaitForSingleObject()와 같은 대기 함수를 호출함으로써 NonSignaled 상태로 변경 되어 다른 쓰레드가 접근할 수 없도록 한다. (프로세스 간의 배타적 제어)
  - 즉, 뮤텍스는 어느 쓰레드에서도 소유되지 않을 때 Signaled 상태가 되고 소유되어 있을 때는 NonSignaled 상태가 된다.
  - 뮤텍스를 원래의 Signaled 상태로 바꾸어주는 ReleaseMutex() 함수를 사용함으로써 대기하고 있던 쓰레드가 다시 실행하게 되는 것이다.
  - 뮤텍스 vs 크리티컬 섹션
    - CriticalSection 의 경우 한 쓰레드가 잘못된 연산을 하거나 강제 종료된다면 무한 대기를 할 수 밖에 없는데 뮤텍스의 경우에는 그것에 대해서 무한정으로 기다리는 일이 없도록 강제적으로 조치를 취한다. WAIT_ABANDONED 값을 전달하여 정상적인 방법이 아닌 포기된 것임을 알려준다. 같은 쓰레드가 같은 뮤텍스를 중복 호출하더라도 데드락(DeadLock)현상이 발생하지 않게 한다.
- 이벤트 (Event)
  - 어떠한 사건에 대하여 알리기 위한 용도로 사용되는 동기화 객체
  - 이벤트 객체는 뮤텍스나 세마포와는 달리 소유권이나 카운터 같은 리소스 접근 제어 속성이 없다. 대신 객체의 시그널 상태를 프로그램에서 자유롭게 제어할 수 있다.
  - 즉, 대기용 API를 호출해서 정지한 쓰레드에 대한 이벤트 객체를 시그널 상태로 바꾸는 방법으로 실행을 재개할 수 있다.

6. Semaphore(세마포)에 대해서 설명하시오

- '깃발' 이라는 뜻으로, 옛날 기찻길에서 빨간색 깃발이면 멈추고, 파란색이면 지나가도 되는 표시를 했던 걸 말한다. s/w 적으로는 임계 영역(critical section)이 기찻길이고, 공유 자원의 사용 가능 여부를 semaphore가 나타낸다. 즉, 세마포는 공유 자원의 개수를 나타내는 변수이다.
- 뮤텍스 vs 세마포
  - 뮤텍스에서는 한 번에 하나의 쓰레드만이 뮤텍스로 보호하는 자원에 접근할 수 있었던 것에 비해 세마포는 사용자가 지정한 개수만큼 이 동기화 객체로 보호하는 자원에 접근할 수 있다.
  - 세마포가 초기화되었을 때에는 뮤텍스와 마찬가지로 Signaled 상태로 되어 있으며, 세마포어의 자원이 하나 사용될 때마다 사용 가능한 자원의 개수는 1씩 감소하게 된다. 이 개수가 1이상일 때까지는 Signaled 상태로 계속 유지되고, 이 값이 0이 되면 세마포의 상태는 NonSignaled가 되어 이 이후에 접근하는 쓰레드들은 대기하게 된다.
- 세마포를 사용하는 경우는 동시에 몇 개의 접근만 가능하게 하고자 할 때 사용한다. 동시 접근이 가능한 시스템의 일정 성능을 확보해야 하는 경우 과도하게 쓰레드가 많이 생성되면 효율이 떨어지는 경우에 그 개수를 제한할 수 있다.
- **Binary semaphore** : 공유 자원이 1개인 경우, 자원을 사용할 수 없는 경우는 0, 사용할 수 있는 경우는 1의 값을 가진다. (Lock 과 비슷하나 값이 반대이다. Lock은 사용할 수 없는 경우에 1을 가진다.)
- **Counting semaphore** : 2이상의 값도 가질 수 있는 경우이다. 예를 들어, 서버에 프린트 기기가 5개가 연결되어 있을 때, 공유 자원인 프린터는 5개로 초기화 된다. 이 경우에서, 사용자가 동시에 사용하는 프린터 기기가 5개가 넘으면 그 다음 프린터 요청은 세마포가 0이니까 보류된다. 이는 후에 누군가가 프린터를 다 쓰고 반환할 때 까지 기다렸다가 처리 된다.

> 자원 & 반납 :

- wait() [세마포를 -1 하는 함수, 세마포가 0이면 기다리는 함수]
- signal() [세마포를 +1 하는 함수, 세마포 사용했으니 돌려놓는 함수]

세마포어 특징

특징	설명
Bus Traffic 효율	Bus Lock 처럼 반복 검사과정이 없어서 BUS 교통량 적음
소프트웨어적인 overhead	P,V 연산 프로그램 코드가 길고 SW 오버헤드 있음
Signal 처리	Wakeup signal 위한 OS 기법 필요
긴 처리에 유리	공유자원의 사용시간이 긴 경우에 세마포어 유리
Starvation 가능	세마포어 안에서 Starvation 과 무한 정지 상태가 될 수 있음. 대기 queue 에서 wakeup 하는 알고리즘이 LIFO 의 경우 심각해서 적절 알고리즘이 필요
그외 문제점	유한 버퍼 생산자 소비자 문제 원형 데드락 발생 가능 문제

표 9-5. Semaphore 의 특징

6. thread를 사용하는 이유는 무엇인지 설명하시오

- 시스템 작업을 효율적으로 관리하기 위해서다. 특히, 멀티 프로세스 작업을 멀티 스레드로 실행하면 자원을 할당할 필요 없이 한번 프로세스에 자원을 할당하면 스레드들 간 이 자원을 공유하여 사용하므로 프로세스를 Context Switching 하는 것 보다 오버헤드가 작다. 그리고, 여러 스레드 간 통신 비용이 프로세스간의 통신비용보다 작다.

7. 선점형과 비선점형에 대해 각각 설명하시오

- 선점형은 뺏는거, 비선점형은 안뺏고 이미 처리중이던거 끝날때까지 기다리기
- 선점형은 다른 프로세스(B)가 CPU를 사용하고 있는 프로세스(A) 대신 자신이 CPU를 점령 할수 있다.
- 대표적인 **비선점형**(다른 프로세스를 끝낼수 없는..) 스케쥴링은
    - FIFO : 대기 큐에 먼저 들어온 작업순으로 CPU를 할당
    - SJF(Shot Job First) : 소요시간이 짧은 작업순으로 할당 (긴 작업은 무한연기(Starvation) 가능성이 있음, 또한 준비 큐에 있는 프로세스의 CPU 요구시간에 대한 정확한 정보를 얻기 어려움)
    - HRN : 우선순위와 대기 시간에 따라 작업을 할당, 대기중인 프로세스 중 현재 응답률 (response ratio)이 가장 높은 것을 선택, 긴 작업과 짧은 작업간의 지나친 불평등을 어느 정도 보완한 기법 (응답률 = (대기시간 + 서비스 시간)/서비스 시간)
- **선점형**(다른 프로세스를 종료시키고 다른 프로세스가 실행되는) 스케쥴링은
    - Round Robin 방식
        - 시간 단위가 설정되어 각 시간동안 프로세스를 실행하고 시간이 지나면 다음 프로세스로 전환되게 된다.
          전체적인 응답 속도가 빨라질수도 있으나, 시간 단위마다 프로세스를 전환할때 문맥 전환에 따른 오버헤드가 발생할 수 있기 때문에 적당한 시간 단위를 설정하여야 한다.
    - SRT(Short Remaining Time) 스케줄링

        - 남은 수행 시간이 가장 짧은 것을 먼저 수행하는 방식, 긴 작업은 SJF 보다 대기 시간이 길다.
    - 다단계 큐(Multilevel Queue) 스케줄링
        - 작업들을 여러 종류의 그룹으로 나누어 여러 개의 큐를 이용하는 기법
    - 다단계 큐(Multilevel Feedback Queue) 스케줄링
        - 여러 개의 대기큐를 두고 각 큐마다 시간 할당량을 달리하는 스케줄링 방식, 입출력 위주와 CPU 위주인 프로세스의 특성에 따라 서로 다른 CPU의 타임 슬라이스를 부여하여 스케줄링함, 짧은 작업에 유리하며 입출력 위주의 작업에 우선권을 부여

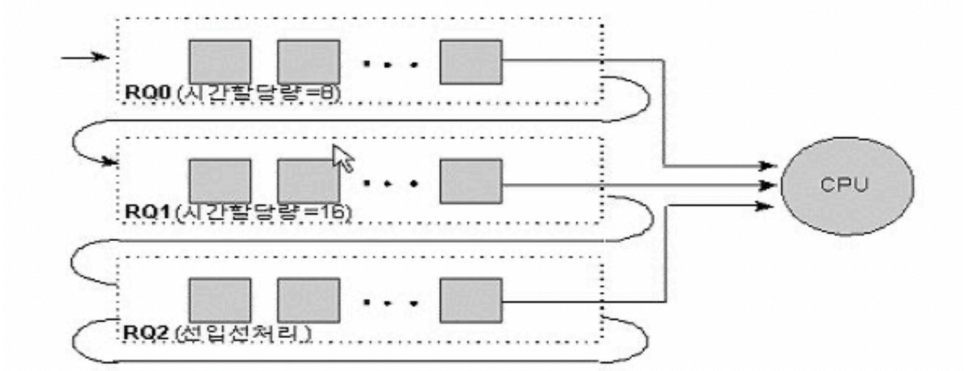


그림 8-8. 다단계 피드백 큐 스케쥴링 형태

8. VM이란 무엇인가?

9. static binding이랑 dynamic binding이랑 설명해봐요

바인딩이란 프로그램 구성 요소의 성격을 결정해주는 것 ex) 변수의 데이터 타입이 무엇인지 정해지는 것

정적 바인딩은 컴파일 시간에 성격이 결정되는 것이고
동적 바인딩은 실행시간에 성격이 결정되는 것이다.

- C언어 컴파일 시간에 변수의 데이터 타입이 결정
- Python(Interpreter 언어) 런타임에 값에 따라 변수의 데이터 타입이 결정

10. activation record 뭐야?
11. scoping은 뭐야? 그걸 왜할까?

    변수의 영역을 정의하는 것 (global, local ..) -> 메모리 공간을 효율적으로 사용하기 위해서 -> 필요 없는 데이터는 가비지 콜렉터가 지우고

12. operation mode는 뭐지?

    명령어를 나타내는 비트에서 어떤 동작이 수행되는지를 나타내는 시그널?

13. 커널모드랑 사용자모드 차이점이 뭐야?
    - 커널에서 중요한 자원을 관리하기 때문에, 사용자가 그 중요한 자원에 접근하지 못하도록 모드를 2가지로 나눈 것입니다.
    - **유저모드**
      : 유저(사용자)가 접근할 수 있는 영역을 제한적으로 두고, 프로그램의 자원에 함부로 침범하지 못하는 모드 입니다.
      : 우리는 여기서 코드를 작성하고, 프로세스를 실행하는 등의 행동을 할 수 있습니다.
      : 간단하게 유저 어플리케이션 코드가 유저모드에서 실행된다. 라고 말할 수 있습니다.

- **커널모드**

  : 모든 자원(드라이버, 메모리, CPU 등)에 접근, 명령을 할 수 있습니다.
  : 유저모드와는 비교가 안되게 컴퓨터 내부에서 모든 짓? 을 할 수 있다고 생각하면 되겠습니다

14. 커널모드가 왜 있어야 될까? 그냥 사용자모드만 있으면 안되나? 내가 사용잔데 내 맘대로 내 PC 쓰는거지 커널모드가 왜 있어야되는데?

    유저가 함부로 건드려서는 안되는 자원 관련 데이터들이 있기 때문에 사용자가 마음대로 프로그램 자원에 손을 댈 수 없게 해야한다. 따라서 프로그램의 자원에 접근하는 모드와 사용자 전용 모드를 나누어서 관리하는 것이다.

15. page 참조하려는데 page 없으면 무슨 상황이 발생해?
    page fault랑 인터럽트랑 같은거야?
    page fault가 발생한 이후에 운영체제가 해주는 일을 자세히 순서대로 기술해봐.

16. First-fit, best-fit, worst-fit 에 대해서 설명하시오.

    메모리에 process 를 할당할때, contiguous allocation을 하는 경우. 즉, 한 프로세스는 연속적으로 메모리에 할당이 되는 것 -> 프로세스마다 필요한 메모리 영역의 크기가 다르다. 이 과정에서 프로세스를 메모리에 어떻게 할당하는가를 정의한 방법들이다.

    그렇기 때문에 External fragmentation(외부 파편화)이 발생한다.

    > External Fragmentation?
    >
    > 현재 메모리에 총 남아있는 영역이 넉넉히 있음에도 연속적으로 프로세스를 담을 수 있는 공간이 없는 경우 (남은 영역의 총 합 > 프로세스 사이즈 but 연속된 영역의 최대 사이즈 < 프로세스 사이즈)
    >
    > Compaction으로 해결할 수 있진 않을까?
    >
    > compaction은 현재 할당된 프로세스들을 옮겨서 빈공간의 영역을 키우는 것인데 옮기는 과정에서 이를 복사하여 저장할 임시의 데이터 공간이 필요한데 그렇게 되면 secondary storage(즉, 하드디스크)같은 것을 도입해야하므로 좋은 방법이 아니다.

    - **First-fit** 은 프로세스가 위치할 곳을 메모리 처음부터 탐색하는데 제일 먼저 들어갈 수 있는 공간이 나오면 거기에 위치시키는 것
    - **Best-fit** 은 프로세스가 그 위치에 들어가면 남는 메모리 영역이 가장 적은 최적의 공간에 위치시키는 것
    - **Worst-fit** 은 프로세스가 위치하면 가장 많은 메모리 영역이 남게되는 곳에 위치시키는 것

17. 멀티 프로그래밍이란?

    각 프로세스가 독립적 메모리 영역을 갖는 것 -> 각 프로세스들이 서로 메모리를 침범하면 안됨

    이런 경우 context switch 순간에 어떤 일이 일어나나? -> 현재 진행중인 프로세스 A의 시작점(base)와 한계 크기(limit)값들을 저정하고, 프로세스 B를 위한 새로운 값을 base와 limit으로 로드 한다.

18. MMU 란?

    Memory Management Unit으로 execution time에 진행되는 run time address binding을 도움을 주는 것 -> CPU코어 안에 탑재되어 가상 주소를 실제 메로리 주소로 변환해주는 장치

    이 과정에서 memory protection을 해주는데 주소 변환을 위해 제공된 base,limit을 이용해 실제 주소를 반환하는 과정에서 cpu가 요청한 주소가 해당 영역을 넘어간다 (실제로 할당될 수 없는 주소이다)라는 경우에 memory protection fault가 발생해 interupt를 걸어줍니다.(cpu가 운영체제가 이를 처리하도록 요청)

    위와 같은 방식을 이용하려면 각 프로세스 등의 메모리 할당을 cotiguous allocation 방식으로 진행해야 한다. 따라서 contiguous allocation은 MMU가 매우 간단하게 설계될 수 있는 장점이 있다. cpu가 요청한 논리 주소가 그 프로세스의 할당 범위보다 넘냐 안넘냐를 base, limit을 가지고만 판단하면 되기 때문이다. 하지만, external fragmentation을 해결하지 못한다는 단점이 있기 때문에 현재의 시스템에서는 잘 쓰이지 않는다.

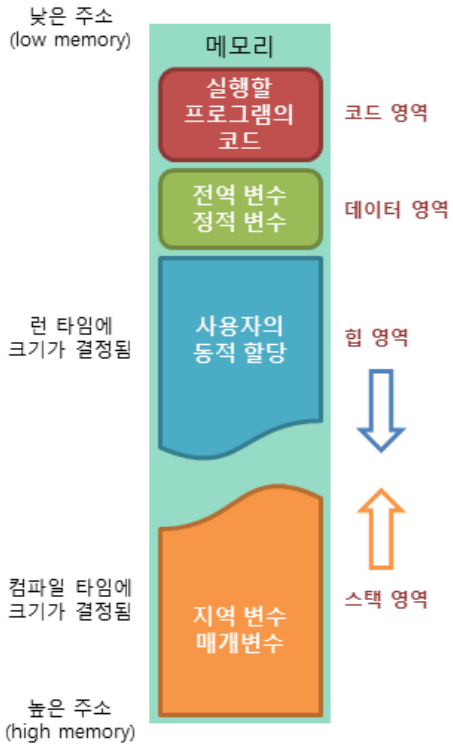
19. 우선순위 스케쥴링

> 여러 개의 어플리케이션이 동시에 수행을 요구하고 CPU의 사용을 요구할 때 어떻게 할 것인지를 반영하여 적절하게 CPU 사용을 할 수 있도록 시간을 배분해주는 것

- 윈도우가 사용하고 있는 방식 : `Round-Robbin` (선점형 우선순위)
  - 수행중인 스레드보다 우선 순위가 높은 스레드가 오면 그 수행을 멈추고 자리를 비켜줘야 하는 방식
  - 윈도우는 기본적으로 스레드가 CPU 사용을 대기하고 있고, 그 줄은 여러 줄로 서 있는데, 그 줄의 의미는 우선순위이다. 이 우선순위에 따라서 CPU가 수행을 하게 된다. 이때 수행이 계속 안되는 현상이 생길 수 있는데, 이를 '기아현상(Starvation)' 이라고 부른다.

20. 메모리의 구조

> 프로그램이 실행되기 위해서는 먼저 프로그램이 메모리에 로드(load) 되어야 하고 프로그램에서 사용되는 변수들을 저장할 메모리도 필요하다.
>
> => 따라서, 컴퓨터의 운영체제는 프로그램의 실행을 위해 다양한 메모리 공간을 제공한다.



1. 코드(code) 영역

   실행할 프로그램의 코드가 저장되는 영역, CPU는 코드 영역에 저장된 명령어를 하나씩 가져가서 처리하게 됨

2. 데이터(data) 영역
   프로그램의 전역 변수와 정적(static) 변수가 저장되는 영역, 프로그램의 시작과 함께 할당되면, 프로그램이 종료되면 소멸

3. 스택(stack) 영역
   함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역, 함수의 호출과 함께 할당되며, 함수의 호출이 완료되면 소멸, 스택 영역에 저장되는 함수의 호출 정보를 스택 프레임(stack frame) 이라고 함, 스택 영역은 LIFO 방식으로 메모리의 높은 주소에서 낮은 주소의 방향으로 할당됨

4. 힙(heap) 영역
   사용자가 직접 관리할 수 있고 관리 해야만 하는 메모리 영역, 사용자에 의해 메모리 공간이 동적으로 할당되고 해제됨, 메모리의 낮은 주소에서 높은 주소의 방향으로 할당됨

21. 메모리 동적 할당 (dynamic allocation)

> 데이터 영역과 스택 영역에 할당되는 메모리의 크기는 컴파일 타임(compile time)에 미리 된다.
>
> 하지만, 힙 영역의 크기는 프로그램이 실행되는 도중인 런 타임(run time)에 사용자가 직접 결정하게 된다.
>
> 이렇게 런 타임에 메모리를 할당받는 것을 메모리의 동적 할당(dynamic allocation)이라고 한다.

- c언어에서 메모리 동적 할당을 위해 malloc() 함수 사용
  - 프로그램이 실행 중일 때 사용자가 직접 힙 영역에 메모리를 할당할 수 있게 해주는 함수

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - malloc() 함수는 인수로 할당받고자 하는 메모리의 크기를 바이트 단위로 전달받습니다.

    이 함수는 전달받은 메모리 크기에 맞고, 아직 할당되지 않은 적당한 블록을 찾습니다.

    이렇게 찾은 블록의 첫 번째 바이트를 가리키는 주소값을 반환합니다.
  - 힙 영역에 할당할 수 있는 적당한 블록이 없을 때에는 널 포인터를 반환합니다.

    주소값을 반환받기 때문에 힙 영역에 할당된 메모리 공간으로 접근하려면 포인터를 사용해야 합니다.
- 힙 영역에 할당받은 메모리 공간을 다시 운영체제로 반환하기 위해 free() 함수 사용
  - 데이터 영역이나 스택 영역에 할당되는 메모리의 크기는 컴파일 타임에 결정되어, 프로그램이 실행되는 내내 고정됩니다.

    하지만 메모리의 동적 할당으로 힙 영역에 생성되는 메모리의 크기는 런 타임 내내 변화됩니다.

    따라서 free() 함수를 사용하여 다 사용한 메모리를 해제해 주지 않으면, 메모리가 부족해지는 현상이 발생할 수 있습니다.

    이처럼 사용이 끝난 메모리를 해제하지 않아서 메모리가 부족해지는 현상을 메모리 누수(memory leak)라고 합니다.

```
#include <stdlib.h>
void free(void *ptr);
```

  - free() 함수는 인수로 해제하고자 하는 메모리 공간을 가리키는 포인터를 전달받습니다.

    인수의 타입이 void형 포인터로 선언되어 있으므로, 어떠한 타입의 포인터라도 인수로 전달될 수 있습니다.

```
ptr_arr = (int*) malloc(arr_len * sizeof(int)); // 메모리의 동적 할당

if (ptr_arr == NULL) // 메모리의 동적 할당이 실패할 경우
{
    printf("메모리의 동적 할당에 실패했습니다.\n");
    exit(1);
}

printf("동적으로 할당받은 메모리의 초깃값은 다음과 같습니다.\n");

for (i = 0; i < arr_len; i++)
```

```
{
    printf("%d ", ptr_arr[i]);
}

free(ptr_arr);        // 동적으로 할당된 메모리의 반환
```

참고

1. https://jhkang-tech.tistory.com/13
2. https://asfirstalways.tistory.com/99

   Node.js 교과서 [조현영, 길벗]
3. https://gbsb.tistory.com/312

   https://jwprogramming.tistory.com/12
4. https://12bme.tistory.com/68

   https://jhnyang.tistory.com/35
5. https://jhnyang.tistory.com/36

   https://jhnyang.tistory.com/101
6. https://www.crocus.co.kr/1510
7. https://secretroute.tistory.com/entry/140819
8. https://blockdmask.tistory.com/69
9. https://jhnyang.tistory.com/264
10. https://jhnyang.tistory.com/247