****

**19s Programming Study**

**-기술면접 예상 질문-**

Update 21-07-16

목차

1. [**운영체제2**](#데이터베이스)

**1. 운영체제**

**1.1 프로세스와 스레드**

**● 프로세스(Process)**

- “컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램”을 뜻하며, 메모리에 올라와 실행되고 있는 프로그램의 인스턴스이다.

- 운영체제로부터 시스템 자원을 할당받는 작업의 단위로, 실행된 프로그램을 의미한다.

**\*\* 참고 \*\***

할당받는 시스템 자원의 예시 : CPU시간, 운영되기 위해 필요한 주소공간, 독립된 메모리 영역

**<특징>**

1) 프로세스는 각각 독립된 메모리 영역(Code,Data,Stack,Heap의 구조)을 할당받는다

2) 기본적으로 프로세스 당 최소 1개의 스레드(메인 스레드)를 갖는다.

3) 각 프로세스는 별도의 주소 공간에서 실행되며, 한 프로세스는 다른 프로세스의 변수나 자료구조에 접근불가

4) 한 프로세스가 다른 프로세스의 자원에 접근하려면 프로세스 간 통신(IPC)를 사용해야한다. 직접 접근 불가.

**● 스레드(Thread)**

- “프로세스 내에서 실행되는 여러 흐름의 단위”를 뜻하며, 프로세스의 특정한 수행경로이다.

- 프로세스가 할당 받은 자원을 이용하는 실행 단위

**<특징>**

1) 스레드는 프로세스 내에서 각각 Stack만 따로 할당받고, Code,Data,Heap 영역은 공유한다.

2) 스레드는 한 프로세스 내에서 동작되는 여러 실행의 흐름으로, 프로세스 내의 주소 공간이나 자원들을 다른 스레드와 공유하면서 실행된다.

3) 한 스레드가 프로세스 자원을 변경하면, 프로세스 내 다른 스레드도 그 변경 결과를 즉시 볼수 있다.

**● 멀티 프로세스(Multi Process)**

- 하나의 응용프로그램을 여러 개의 프로세스로 구성하여 각 프로세스가 하나의 작업을 처리하도록 하는 것.

**<장점>**

1) 여러 개의 자식 프로세스 중 하나에 문제가 발생하면 해당 자식 프로세스만 죽고 다른 영향이 없음.

**<단점>**

1) Context Switching과정에서 캐쉬 메모리 초기화 등 무거운 작업이 진행되고 많은 시간이 소모되는 오버헤드 발생. 각 프로세스는 독립된 메모리를 할당 받아 Context Switching 발생하면 캐쉬 리셋 후 다시 불러와야 한다.

2) 프로세스들은 서로 변수를 공유할 수 없어서 어렵고 복잡한 통신기법(IPC)를 이용해야 한다.

**\*\* 참고 \*\***

**Context Swiching이란?** 동작 중인 프로세스가 대기상태로 변경하면서 해당 프로세스의 상태(Context)를 보관하고, 대기하고 있던 다음 순서의 프로세스가 동작하면서 이전에 보관했던 프로세스의 상태를 복구하는 작업

**● 멀티 스레드(Multi Thread)**

- 하나의 응용프로그램을 여러 개의 스레드로 구성하고 각 스레드로 하여금 하나의 작업을 처리하도록 함.

- 많은 운영체제들이 멀티 프로세싱을 지원하고 있으나, 멀티 스레딩을 기본으로 함. (웹 서버가 대표적 예)

**<장점>**

1) 시스템 자원 소모 감소: 프로세스를 생성해 자원을 할당하는 시스템 콜이 줄어 자원의 효율성 증대

2) 시스템 처리량 증가: 스레드 간 데이터 교환이 간단하고, Context Switching이 빨라 처리비용이 감소

3) 간단한 통신 방법: 프로세스 내 Stack 영역을 제외한 모든 메모리를 공유하므로 프로그램 응답시간이 단축

**<단점>**

1) 디버깅이 까다로우며, 섬세한 설계가 필요하다

2) 다른 프로세스에서 스레드를 제어할 수 없다.

3) 하나의 스레드가 문제가 생기면 전체 프로세스가 영향을 받는다.

4) 멀티 스레드의 경우, 전역변수 등을 함께 사용하므로 자원 공유의 문제가 발생할 수 있음. (동기화 issue)

= 멀티 프로세스 사용 예시: 각 프로세스가 독립적으로 작업을 수행할 수 있는 경우   
= 멀티 스레드 사용 예시: 서로 정보를 공유함으로써 얻는 이득이 큰 경우

**1.2 스케줄링**

**● 프로세스를 스케줄링하기 위한 queue**

1) Job Queue: 현재 시스템 내에 있는 모든 프로세스의 집합

2) Ready Queue: 현재 메모리 내에 있으면서 CPU를 잡아서 실행되기를 기다리는 프로세스의 집합

3) Device Queue: Device I/O작업을 대기하고 있는 프로세스의 집합

**● 스케줄러의 종류**

1) 장기 스케줄러

- 대용량 메모리에 임시로 저장된 프로세스 중 어떤 것에 메모리를 할당하여 Ready Queue로 보낼지 결정.

- 메모리와 디스크 사이의 스케줄링을 담당하며, 프로세스에 메모리를 할당함.

- 프로세스의 상태 : New -> Ready

2) 단기 스케줄러

- CPU와 메모리 사이의 스케줄링 담당

- Ready Queue에 존재하는 프로세스 중 어떤 프로세스를 running 시킬지 결정

- 프로세스 상태: Ready –> Running ->Waiting -> Ready

3) 중기 스케줄러

- 여유공간 마련을 위해 프로세스를 통째로 메모리에서 디스크로 보냄.

- 현 시스템에서 메모리에 너무 많은 프로그램이 올라가는 것을 조절하는 스케줄러

- 프로세스 상태: Ready -> Suspended

**● CPU 스케줄러**

1) 선점형 스케줄러: 한 프로세스가 CPU를 점유 중일 때, 우선 순위가 높은 프로세스가 CPU를 차지할 수 있으나, 선점이 일어날 경우 오버헤드가 발생하며 처리시간 예측이 힘들다.

**- SRT(Shortest Remaining Time) 스케줄링:** 짧은 시간 순서대로 프로세스를 수행한다. 남은 처리 시간이 더 짧은 프로세스가 Ready 큐에 들어오면 그 프로세스가 바로 선점됨. 아래에 소개할 SJF의 선점 버전이라고 할 수 있다.

**- 라운드로빈(Round-Robin)스케줄링:** 각 프로세스는 같은 크기의 CPU 시간을 할당 받고 선입선출에 의해 행된다. 할당시간이 너무 크면 선입선출과 다를 바가 없어지고, 너무 작으면 오버헤드가 너무 커진다.

**- 다단계 큐(Multi-level Queue) 스케줄링:** Ready큐를 여러 개 사용하는 기법. 각각의 큐는 자신의 스케줄링 알고리즘을 수행하며, 큐와 큐 사이에도 우선순위를 부여한다.

**- 다단계 피드백 큐 스케줄링:** 다단계 큐와 비슷하나 프로세스들이 큐를 이동할 수 있다.

2) 비선점형 스케줄러: 프로세스가 작업이 완료 된 후, 스스로 CPU를 놓아주는 시점에만 스케줄링이 일어난다.

**-**  **HRN(Highest response ratio next) 스케줄링:** 긴 작업과 짧은 작업간의 지나친 불평등을 어느 정도 보완한 기법. 수행시간의 길이와 대기 시간을 모두 고려해 우선순위를 정한다.

**- SJF(Shortest Job First) 스케줄링:** 큐 안에 있는 프로세스 중 수행시간이 짧은 것을 먼저 수행. 평균 대기 시간을 감소시킨다.

**- 우선순위(priority) 스케줄링:** 프로세스에게 우선순위를 정적, 혹은 동적으로 부여하여 우선순위가 높은 순서대로 처리한다. 동적으로 부여할 경우, 구현이 복잡하고 오버헤드가 많다는 단점이 있으나, 시스템의 응답속도를 증가시킨다.

**- 기한부(Deadline) 스케줄링:** 작업을 명시된 시간이나 기한 내에 완료하도록 계획.

**- FIFO 스케줄링:** 프로세스들은 Ready큐에 도착한 순서대로 CPU를 할당 받는다. 작업 완료 시간을 예측하기 매우 용이하다. 하지만 덜 중요한 작업이 중요한 작업을 기다리게 할 수도 있다.

**1.3 동기화와 데드락**

**● 동기화(Synchronization)**

- 협력하는 프로세스 사이에서 실행 순서 규칙을 정하여 공유 자원의 일관성을 보장하는 것을 뜻함.

- 공유 자원을 사용하는 상황에서, 경쟁 조건이 발생하면 공유 자원을 신뢰할 수 없게 만들 수 있어, 이를 방지하기 위해 프로세스들이 공유 자원을 사용할 때 특정 규칙을 만드는 것.

\*\* 경쟁 조건(Race Condition)

- 여러 프로세스(또는 스레드)가 공유 자원에 동시에 접근할 때, 공유 자원에 대한 접근 순서에 따라 실행결과가 달라질 수 있는 상황 (여러 프로세스들이 공유 자원을 두고 경쟁하는 형태)

**● 임계영역**

- 멀티 스레드 프로그램에서 공유 자원이 참조 가능한 코드 영역을 말한다. 즉, 경쟁 조건이 발생할 수 있는 프로그램 코드 부분이다.

**<임계영역 문제를 해결하기 위한 3가지>**

1)상호배제(**Mut**ual **Ex**clusion): 하나의 프로세스가 임계 영역에 들어가 있으면 다른 프로세스는 들어갈 수 없도록 막아주는 것.

2)진행(Process): 임계영역에 들어간 프로세스가 없다면, 어느 프로세스가 들어갈 것인지 적절히 선택해줘야 함. 즉, 임계 영역안에 있는 프로세스 외에는, 다른 프로세스가 임계 영역에 진입하는 것을 방해하면 안된다.

3)한정 대기(Bounded Waiting): 교착상태를 방지하기 위해, 한 번 들어갔다 나온 프로세스는 다음에 들어갈 때 제한을 준다.

**● 뮤텍스(Mutex), 세마포어(Semaphore), 모니터(Monitor)**

1) 뮤텍스

- 공유자원에 여러 스레드가 접근하려고 할 때, Lock을 통해 한 번에 하나의 스레드만 접근하도록 허용하는 것.

- 임계영역은 한 프로세스 내의 스레드에만 사용 가능하지만, 뮤텍스는 서로 다른 프로세스에 속한 스레드에도 적용 가능.

2) 세마포어

- 한정된 자원을 여러 스레드가 사용하려고 할 때 접근을 제한한다.

- 카운트 기능으로 여러 개의 스레드가 임계영역에 접근이 가능하다. 즉, 동기화 대상이 하나 이상.

3) 모니터

- 상호배제 하게끔 만들어진 추상화된 데이터 형태로, 키의 획득과 사용 후 해제를 모두 처리해준다.

- 하나의 프로세스 내에서 다른 스레드 간의 동기화에 사용된다.

**● 교착상태(Deadlock)**

- 두 개 이상의 작업이 하나씩 자원을 소지(Hold)하고 있으면서 상대방이 가진 자원을 서로 원하고(Need) 있는 상태.

- 어떤 작업도 실행되지 못하고 계속 서로 상대방의 작업이 끝나기를 바라는 무한정 대기상태.

**<교착상태 발생 조건>**

1) 상호배제(Mutual Exclusion): 한 번에 한 프로세스만이 자원을 사용하는 것.

2) 점유대기(Hold and Wait): 하나 이상의 자원을 소지한 프로세스가 다른 프로세스가 가진 자원을 추가로 얻기 위한 대기

3) 비선점(No preemption): 자원을 강제적으로 뺏을 수 없고, 사용중인 프로세스가 종료된 후 반환되어야 사용 가능.

4) 순환대기(Circular wait): 대기 프로세스의 집합이 순환 형태로 자원을 대기 (A->B, B->C, … , N->A)

\*\* 각 조건은 서로 완전히 독립적이지 않으며, 위 조건을 모두 만족해야 교착상태가 발생한다.

**● 교착상태(Deadlock) 해결법**

**1) 데드락 예방**

-데드락의 발생조건 4가지 중 하나라도 발생하지 않도록 예방하는 것.

1-1) 상호 배제 방지: 한 번에 여러 프로세스가 공유 자원을 사용할 수 있게 한다.

1-2) 점유 대기 방지: 프로세스 실행에 필요한 모든 자원을 한꺼번에 요구하고 허용할 때까지 작업을 보류해서, 나중에 또 다른 자원을 점유하기 위한 대기 조건을 성립하지 않도록 한다.

1-3) 비선점 방지: 이미 다른 프로세스에게 할당된 자원이 선점권이 없다고 가정할 때, 높은 우선순위의 프로세스가 해당 자원을 선점할 수 있도록 한다.

1-4) 순환대기 방지: 자원을 순환 형태로 대기하지 않도록 일정한 한쪽 방향으로만 자원을 요구할 수 있도록 한다.

\*\*단점: 자원의 낭비가 심하고 시스템 처리율이 감소될 수 있음.

**2) 데드락 회피**

- 자원할당 상태(사용 가능한 자원 수, 할당된 자원 수 등)를 검사해 교착상태가 발생하면 피해가도록 한다.

- 불안전상태이면 교착상태가 일어날 수 있으므로, 안전상태가 되도록 회피하는 알고리즘을 수행한다.

2-1) 자원할당그래프 알고리즘: 교착상태 회피를 위해서 자원할당그래프에 예약 간선을 추가한다. 시스템에서 자원이 반드시 미리 예약되어 그래프에 예약 간선이 모두 표시되어야 한다.

2-2) 은행원 알고리즘: 필요한 자원의 최대 개수를 미리 신고해 프로세스가 자원을 요청했을 때, 요청을 수락하여도 안전상태에 머무를 수 있는지 판단 후 요청을 수락한다. 만약 안전상태가 아니게 된다면 다른 프로세스가 끝날 때까지 요청 수락 대기한다.

**3) 데드락 탐지**

- Allocation, Request, Available 등으로 시스템에 데드락이 발생했는지 여부를 탐색한다. 현재 시스템의 자원할당 상태를 가지고 파악한다.

**4) 데드락 회복**

4-1) 교착 상태가 일어난 프로세스를 종료: 교착상태의 프로세스를 모두 중지하거나, 교착상태가 제거될 때까지 한 프로세스씩 중지한다.

4-2) 프로세스에 할당된 자원을 선점: 교착상태의 프로세스가 점유하고 있는 자원을 선점해 다른 프로세스로 할당하고, 해당 프로세스는 일시 정지시킨다. 우선 순위가 낮은 프로세스 위주로 자원을 선점 할당한다.

**1.4 메모리 관리 전략**

- 각 프로세스들은 독립된 메모리 공간을 갖고, 운영체제 또는 다른 프로세스의 메모리 공간에 접근할 수 없는 제한이 걸려있음. 단지, 운영체제만이 운영체제 메모리 영역과 사용자 메모리 영역의 접근에 제약을 받지 않음.

**● 메모리 관리 용어**

1) 스와핑(Swapping)

- 메모리의 관리를 위해 사용되는 기법. 표준 Swapping방식으로는 라운드로빈과 같은 스케줄링의 다중 프로그래밍 환경에서 CPU 할당 시간이 끝난 프로세스의 메모리를 보조 기억장치로 내보내고 다른 프로세스의 메모리를 불러들일 수 있다.

- 이 과정을 Swap이라고 하며, 주 기억장치로 불러오는 과정을 Swap-in, 보조 기억장치로 내려보내는 과정을 Swap-out이라고 한다.

- Swap에는 큰 디스크 전송시간이 필요하기 때문에 현재에는 메모리 공간이 부족할 때 Swapping이 일어난다.

2) 단편화(Fragmentation)

- 프로세스들이 메모리에 적재되고 제거되는 일이 반복되다 보면, 프로세스들이 차지하는 메모리 틈 사이에 사용하지 못할 만큼의 작은 자유공간이 늘어나게 되는데 이를 단편화라고 한다.

- 외부 단편화: 메모리 공간 중 사용하지 못하게 되는 일부분. 물리 메모리(RAM)에서 사이사이 남는 공간을 모두 합치면 충분한 공간이 되는 부분들이 분산되어 있을 때 발생.

- 내부 단편화: 프로세스가 사용하는 메모리 공간에 포함된 남는 부분. 예를 들어 메모리 분할 자유공간이 10,000B가 있고, A프로세스가 9,998B를 사용하게 되면 2B의 남는 공간이 생기는데 이를 내부 단편화라 한다.

3) 압축

- 외부 단편화를 해소하기 위해 프로세스가 사용하는 공간들을 한쪽으로 몰아, 자유공간을 확보하는 방법론.

- 작업효율이 좋지 않다는 단점이 있다.

**● 페이징과 세그멘테이션**

1) 페이징(Paging)

- 하나의 긴 프로세스가 사용하는 메모리 공간이 연속적이어야 한다는 제약을 없애는 메모리 관리 방법.

- 외부 단편화와 압축 작업을 해소하기 위해 생긴 방법론으로, 물리 메모리는 Frame이라는 고정 크기로 분리되어 있고, 논리 메모리는 페이지라 불리는 고정 크기의 블록으로 분리된다.

- 하나의 프로세스가 사용하는 공간은 여러 개의 페이지로 나뉘어서 관리되고, 개별 페이지는 순서에 상관없이 물리 메모리에 있는 Frame에 맵핑되어 저장된다.

<장점>

- 논리 메모리는 물리 메모리에 저장될 때, 연속되어 저장될 필요가 없고 물리 메모리의 남는 프레임에 적절히 배치됨으로 외부 단편화를 해결할 수 있음.

<단점>

- 내부 단편화 문제의 비중이 늘어나게 된다. (페이지 크기가 1,024B이고 A프로세스가 3,172B의 메모리를 요구한다면 4개의 페이지 프레임이 필요하게 되고 결론적으로 마지막 페이지 프레임의 924B의 여유공간이 남게 되는 내부 단편화 발생)

2) 세그멘테이션(Segmentation)

- Paging에서처럼 논리 메모리와 물리 메모리를 같은 크기의 블록이 아닌, 서로 다른 크기의 논리적 단위인 세그먼트로 분할한다.

- 사용자가 두 개의 주소로 지정(세그먼트 번호, 한계)하며, 세그먼트 테이블에는 각 세그먼트의 기준(세그먼트 시작 물리 주소)와 한계(세그먼트 길이)를 저장한다.

<단점>

- 서로 다른 크기의 세그먼트들이 메모리에 적재되고 제거되는 일이 반복되다 보면, 자유 공간들이 많은 수의 작은 조각들로 나누어져 못 쓰게 될 수 있음. 즉, 외부 단편화가 발생할 수 있음.