운영체제(OS) 핵심 개념 정리

I. CPU 스케줄링 (CPU Scheduling)

1. 기본 개념

- **CPU-I/O 버스트 사이클**: 프로세스 실행은 CPU 사용(CPU 버스트)과 입출력 대기(I/O 버스트)가 반복되는 형태로 구성됩니다.
- **CPU 스케줄러**: 준비 큐(Ready Queue)에 있는 프로세스 중 하나를 선택하여 CPU를 할당합니다.
- **디스패처 (Dispatcher)**: 스케줄러가 선택한 프로세스에게 CPU 제어권을 넘겨주는 모듈입니다. (문맥 교환, 사용자 모드 전환 등 수행)

2. 스케줄링 성능 평가 기준

- **CPU 활용률 (Utilization)**: CPU를 얼마나 바쁘게 사용하는가 (최대화 목표)
- **처리량 (Throughput)**: 단위 시간당 완료되는 프로세스의 수 (최대화 목표)
- **반환 시간 (Turnaround Time)**: 프로세스 시작부터 완료까지 걸린 총 시간 (최소화 목표)
- **대기 시간 (Waiting Time)**: 프로세스가 준비 큐에서 대기한 총 시간 (최소화 목표)
- **응답 시간 (Response Time)**: 요청 후 첫 번째 응답이 올 때까지의 시간 (최소화 목표)

3. 주요 스케줄링 알고리즘

- **선입선출 (FCFS, First-Come, First-Served)**
  - **특징**: 가장 간단한 비선점형 방식. 먼저 온 요청을 먼저 처리합니다.
  - **문제점**: **호송 효과(Convoy Effect)** 발생 가능. (긴 프로세스가 짧은 프로세스들을 기다리게 함)
- **최단 작업 우선 (SJF, Shortest-Job-First)**
  - **특징**: CPU 버스트 길이가 가장 짧은 프로세스에게 CPU를 할당합니다. 평균 대기 시간을 최소화하는 최적의 알고리즘입니다.
  - **종류**:
    - **비선점형 SJF**: 현재 프로세스가 끝나야 다음 프로세스로 넘어갑니다.
    - **선점형 SJF (SRTF, Shortest-Remaining-Time-First)**: 새로 도착한 프로세스의 버스트 시간이 현재 실행 중인 프로세스의 남은 시간보다 짧으면 CPU를 선점합니다.
- **라운드 로빈 (RR, Round Robin)**
  - **특징**: 시분할 시스템의 핵심. 각 프로세스에 동일한 **시간 할당량(Time Quantum)**을 부여하고, 시간이 지나면 준비 큐의 맨 뒤로 보냅니다.
  - **장점**: 응답 시간이 빠르고 공평합니다.
  - **단점**: 시간 할당량(q)의 크기는 중요한 트레이드오프 관계를 가집니다. 시간 할당량이 작으면 문맥 교환이 잦아져 오버헤드가 증가하지만, 사용자는 빠른 응답 시간을 경험하게 됩니다. 반면, 시간 할당량이 크면 오버헤드는 줄어들지만 FCFS처럼 동작하여 긴급한 작업의 응답 시간이 길어질 수 있습니다. 따라서 시스템의 목적에 맞는 적절한 q 값을 설정하는 것이 중요합니다.
- **우선순위 스케줄링 (Priority Scheduling)**

    - **특징**: 우선순위가 가장 높은 프로세스에게 CPU를 할당합니다.
    - **문제점**: **기아 상태(Starvation)** 발생 가능. (낮은 우선순위 프로세스가 영원히 실행되지 못함)
    - **해결책**: **에이징(Aging)** 기법을 통해 오래 대기한 프로세스의 우선순위를 점진적으로 높여줍니다.
- **다단계 큐 / 다단계 피드백 큐**
    - **다단계 큐**: 프로세스를 종류별로 여러 큐에 나누어 담고, 각 큐는 독립적인 스케줄링 알고리즘을 가집니다. (큐 간 이동 불가)
    - **다단계 피드백 큐**: 다단계 큐에서 큐 간 이동을 허용하여, 프로세스의 실행 특성에 따라 동적으로 우선순위를 조정합니다. (가장 유연한 방식)

4. 다음 CPU 버스트 길이 예측: 지수 평균 (Exponential Averaging)

- **중요성**: SJF/SRTF 알고리즘의 성능은 CPU 버스트 길이를 얼마나 정확하게 예측하는지에 따라 달라집니다.
- **공식**:$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
    - τ_{n+1}: 다음 CPU 버스트의 예측값
    - t_n: n번째 실제 CPU 버스트의 길이
    - τ_n: n번째 CPU 버스트의 예측값
    - α: 가중치 ($0 \le \alpha \le 1$). 최근 이력의 중요도를 결정합니다.
        - $\alpha=0$: 과거 예측값만 사용. ($\tau_{n+1}=\tau_n$)
        - $\alpha=1$: 가장 최근 실제 값만 사용. ($\tau_{n+1}=t_n$)
        - $\alpha=1/2$: 과거 예측과 최근 실제 값을 동일한 비중으로 반영.

Ⅱ. 프로세스 동기화 (Process Synchronization)

1. 동기화 문제

- **경쟁 상태 (Race Condition)**: 여러 프로세스가 동시에 공유 데이터에 접근하여 조작할 때, 실행 순서에 따라 결과가 달라지는 현상.
- **임계 구역 (Critical Section)**: 공유 데이터에 접근하는 코드 영역.
- **임계 구역 문제 해결 조건**:
    1. **상호 배제 (Mutual Exclusion)**: 한 번에 하나의 프로세스만 임계 구역에 진입 가능해야 합니다.
    2. **진행 (Progress)**: 임계 구역에 들어갈 프로세스를 결정하는 과정이 무한히 연기되어서는 안 됩니다.
    3. **유한 대기 (Bounded Waiting)**: 어떤 프로세스도 무한정 임계 구역 진입을 기다려서는 안 됩니다.

2. 동기화 도구

- **뮤텍스 (Mutex)**
    - **목적**: **상호 배제**를 위한 잠금(lock) 메커니즘.
    - **소유권**: 락을 획득한 스레드만이 해제할 수 있습니다.
    - **동작**: acquire() (락 획득)와 release() (락 해제) 연산으로 임계 구역을 보호합니다.

- **세마포어 (Semaphore)**
  - **목적**: 자원의 가용 개수를 관리하거나, 프로세스 실행 순서를 제어합니다.
  - **종류**:
    - **카운팅 세마포어**: 가용한 자원의 개수를 나타내는 정수 값을 가집니다.
    - **이진 세마포어**: 0 또는 1의 값을 가지며, 뮤텍스처럼 동작할 수 있습니다.
  - **동작**: wait() (P 연산: 자원 획득 시도 및 카운터 감소)와 signal() (V 연산: 자원 반납 및 카운터 증가) 연산으로 동기화를 수행합니다.
  - **뮤텍스와의 차이점**: 뮤텍스는 '소유권' 개념이 있어 락을 건 스레드만이 해제할 수 있는 반면, 세마포어는 소유권 개념이 없어 어떤 스레드든 signal 연산을 할 수 있습니다. 또한 뮤텍스는 주로 상호 배제를 위한 '잠금'이 목적인 반면, 세마포어는 여러 개의 자원을 계수하거나 실행 순서를 제어하는 '신호' 메커니즘으로 더 폭넓게 사용됩니다.
- **스핀락 (Spinlock)**
  - **개념**: 락을 얻을 수 없을 때, CPU를 양보하지 않고 **바쁜 대기(busy-waiting)**하며 락 상태를 반복적으로 확인하는 방식.
  - **단일 코어 시스템 문제**: 단일 코어에서는 락을 기다리는 스레드가 CPU를 점유하여, 정작 락을 해제해야 할 스레드가 실행되지 못하게 만들어 시스템 성능을 심각하게 저하시킵니다.

3. 고전적인 동기화 문제

- **생산자-소비자 문제 (Bounded-Buffer Problem)**: 유한한 크기의 버퍼를 공유하는 생산자와 소비자 간의 동기화 문제. mutex, empty, full 세 개의 세마포어를 사용하여 해결합니다.
- **읽기-쓰기 문제 (Readers-Writers Problem)**: 여러 독자는 동시에 데이터에 접근 가능하지만, 쓰기 작업은 배타적으로 이루어져야 하는 문제.
- **식사하는 철학자 문제 (Dining-Philosophers Problem)**: 교착 상태의 발생 가능성을 보여주는 대표적인 예시.

Ⅲ. 교착 상태 (Deadlock)

1. 교착 상태란?

두 개 이상의 프로세스가 서로 상대방이 가진 자원을 기다리며 작업을 진행하지 못하는 **무한 대기 상태**를 의미합니다.

2. 교착 상태 발생의 4가지 필요 조건 (Coffman Conditions)

아래 네 가지 조건이 **모두** 충족될 때 교착 상태가 발생할 수 있습니다.

1. **상호 배제 (Mutual Exclusion)**: 한 자원은 한 번에 하나의 프로세스만 사용 가능해야 합니다.
2. **점유와 대기 (Hold and Wait)**: 프로세스가 자원을 보유한 상태에서 다른 자원을 추가로 기다립니다.
3. **비선점 (No Preemption)**: 다른 프로세스의 자원을 강제로 빼앗을 수 없습니다.
4. **순환 대기 (Circular Wait)**: 프로세스들이 원형으로 서로의 자원을 대기합니다. (예: P0 → P1 → P2 → P0)

3. 교착 상태 처리 방법

- **교착 상태 예방 (Prevention)**

    - **방법**: 4가지 필요 조건 중 하나를 원천적으로 차단합니다.
    - **예시**: 자원에 순서를 부여하여 오름차순으로만 요청하게 함으로써 **순환 대기**를 방지.
    - **단점**: 자원 활용률이 낮아지고 시스템 성능이 저하될 수 있습니다.
- **교착 상태 회피 (Avoidance)**
    - **방법**: 시스템이 항상 **안전 상태(Safe State)**를 유지하도록 자원 할당을 동적으로 제어합니다. (안전 상태: 교착 없이 모든 프로세스가 완료될 수 있는 순서가 존재하는 상태)
    - **대표 알고리즘**: **은행가 알고리즘 (Banker's Algorithm)**. 이 알고리즘은 각 프로세스의 최대 자원 요구량(Max), 현재 할당된 자원(Allocation), 추가로 필요한 자원(Need), 그리고 시스템에 남아있는 가용 자원(Available) 정보를 바탕으로, 자원 할당 요청이 들어왔을 때 모든 프로세스가 안전하게 종료될 수 있는 순서(Safe Sequence)가 존재하는지 시뮬레이션하여 교착 상태를 회피합니다.
    - **단점**: 최대 자원 요구량을 미리 알아야 하는 제약이 있습니다.
- **교착 상태 탐지 및 복구 (Detection & Recovery)**
    - **방법**: 교착 상태 발생을 허용하되, 주기적으로 탐지하여 복구합니다.
    - **탐지**: **대기 그래프(Wait-for Graph)**에서 사이클을 찾습니다.
    - **복구**: 교착 상태에 빠진 프로세스를 강제 종료하거나, 자원을 선점합니다.
    - **특징**: 이 접근법은 교착 상태가 드물지만 발생 시 복구가 중요한 데이터베이스 시스템 등에서 사용됩니다. 반면, Windows나 Linux 같은 대부분의 범용 운영체제는 성능 저하를 우려하여 교착 상태를 의도적으로 '무시'하는 전략을 택하는 경우가 많으며, 프로그래머가 동기화 문제를 직접 해결하도록 기대합니다.

Ⅳ. 메모리 관리 (Main Memory)

1. 기본 개념

- **MMU (Memory Management Unit)**: 논리 주소(프로그램이 사용하는 가상 주소)를 물리 주소(실제 메모리 주소)로 변환하는 하드웨어 장치.
- **주소 바인딩**: 프로그램의 주소가 메모리 주소에 연결되는 시점 (컴파일, 로드, 실행 시간).
- **단편화 (Fragmentation)**: 메모리 공간이 낭비되는 현상.
    - **외부 단편화 (External)**: 메모리 공간은 충분하지만, 연속적이지 않아 할당할 수 없는 문제.
    - **내부 단편화 (Internal)**: 할당된 메모리 블록이 실제 필요한 크기보다 커서 내부의 일부가 낭비되는 문제.

2. 메모리 관리 기법

- **연속 메모리 할당 (Contiguous Allocation)**
    - **방식**: 각 프로세스가 메모리의 단일 연속된 공간에 위치합니다.
    - **문제점**: **외부 단편화**가 심하게 발생할 수 있습니다.
    - **해결책**: **압축(Compaction)**을 통해 흩어진 빈 공간을 하나로 모을 수 있으나 비용이 비쌉니다.
- **페이징 (Paging)**
    - **개념**:
        - **페이지(Page)**: 프로세스의 논리 주소 공간을 고정된 크기로 나눈 블록.
        - **프레임(Frame)**: 물리 메모리를 페이지와 동일한 크기로 나눈 블록.

        - 프로세스의 페이지들이 물리 메모리의 프레임에 **비연속적으로** 할당됩니다.
    - **장점**: **외부 단편화를 근본적으로 해결**합니다.
    - **단점**: 프로세스의 마지막 페이지에서 **내부 단편화**가 발생할 수 있습니다.
    - **페이지 테이블 (Page Table)**: 논리 주소(페이지 번호)를 물리 주소(프레임 번호)로 변환하기 위한 매핑 테이블.

3. 페이징 성능 및 구조 최적화

- **TLB (Translation Lookaside Buffer)**
    - **개념**: 페이지 테이블의 일부를 캐싱하는 고속 하드웨어 캐시.
    - **역할**: 페이지 테이블을 참조하기 위한 메모리 접근 횟수를 줄여 주소 변환 속도를 향상시킵니다.
    - **성능**: TLB **히트율(Hit Ratio)**이 높을수록 시스템 성능이 크게 향상됩니다.
- **페이지 테이블 구조**
    - **계층적 페이징 (Hierarchical Paging)**: 거대한 페이지 테이블 자체를 페이징하여 크기 문제를 해결합니다. (예: 2단계, 3단계 페이징)
    - **역 페이지 테이블 (Inverted Page Table)**: 물리 프레임 기준으로 테이블을 구성하여 시스템 전체에 하나만 유지합니다.

Ⅹ. 멀티코어 환경과 캐시 일관성

- **문제 발생 이유**: 각 코어가 자신만의 로컬 캐시를 가지므로, 여러 코어가 동일한 데이터를 각자 캐시에 복사해 둔 상태에서 한 코어가 데이터를 수정하면 다른 코어의 캐시 데이터와 불일치가 발생합니다 (Stale Data).
- **MESI 프로토콜**: 캐시 일관성을 유지하기 위한 대표적인 프로토콜. 각 캐시 라인의 상태를 **Modified, Exclusive, Shared, Invalid** 네 가지로 정의하여 데이터의 일관성을 관리합니다.