1. CPU 스케줄링에서 '다음 CPU 버스트 길이 예측'은 왜 중요하며, 이를 위해 '지수 평균' 방식은 어떻게 작동하나요?

CPU 스케줄링에서 '다음 CPU 버스트 길이 예측'은 프로세스에게 CPU를 얼마나 할당할지 결정하는 데 중요한 역할을 합니다. 특히 최단 작업 우선(SJF) 알고리즘이나 그 선점형 버전인 최단 잔여 시간 우선(SRTF) 알고리즘과 같이 CPU 버스트 길이에 따라 우선순위가 결정되는 스케줄링 방식에서 더욱 중요합니다. CPU 버스트 길이를 정확히 예측하면 시스템의 평균 대기 시간을 최적화하고 CPU 활용률을 높일 수 있습니다.

'지수 평균(Exponential Averaging)'은 과거 CPU 버스트 길이를 바탕으로 다음 CPU 버스트 길이를 예측하는 방법입니다. 그 공식은 다음과 같습니다:

$ \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n $

여기서:

- τ_{n+1} : 다음 CPU 버스트의 예측값
- t_n : n번째 실제 CPU 버스트의 길이
- τ_n : n번째 CPU 버스트의 예측값 (이전 예측)
- α : 가중치 ($0 \leq \alpha \leq 1$)

α 값에 따라 예측 방식이 달라집니다:

- **$\alpha = 0$**: $\tau_{n+1} = \tau_n$. 최근 실제 버스트는 전혀 고려하지 않고, 이전 예측값만 계속 유지합니다. 즉, 최근 이력이 예측에 반영되지 않습니다.
- **$\alpha = 1$**: $\tau_{n+1} = t_n$. 오직 가장 최근의 실제 CPU 버스트 길이만을 다음 예측값으로 사용합니다. 즉, 과거의 모든 이력을 무시하고 직전 CPU 버스트만 중요하게 봅니다.
- **$0 < \alpha < 1$**: 실제 CPU 버스트(t_n)와 이전 예측(τ_n)을 적절히 혼합하여 예측합니다. 일반적으로 α는 1/2로 설정되는 경우가 많습니다. 이 경우, 공식을 확장하면 각 연속적인 항(과거의 실제 버스트들)이 이전 항보다 적은 가중치를 갖게 되어, 최근 이력에 더 큰 중요도를 부여하면서도 과거의 경향을 반영합니다.

이를 통해 시스템은 프로세스의 CPU 사용 패턴에 동적으로 적응하여 보다 효율적인 스케줄링 결정을 내릴 수 있습니다.

2. 운영체제에서 메모리 단편화(Fragmentation)는 무엇이며, 외부 단편화와 내부 단편화의 차이점 및 페이징 기법이 이 문제들을 어떻게 다루나요?

메모리 단편화는 주 메모리(RAM)를 효율적으로 사용하지 못하고 낭비되는 현상을 의미합니다. 이는 메모리 할당 및 해제 과정이 반복되면서 발생합니다.

- **외부 단편화(External Fragmentation)**: 총 사용 가능한 메모리 공간은 충분히 크지만, 이 공간들이 연속적이지 않고 여러 개의 작은 '홀(hole)'로 흩어져 있어, 요청된 크기의 연속적인 메모리 블록을 할당할 수 없는 문제입니다. 예를 들어, 10KB와 15KB의 빈 공간이 각각 있지만, 20KB의 연속된 메모리를 요구하는 프로세스는 할당받을 수 없는 상황입니다. 이는 가변 분할 방식에서 프로세스들이 메모리에 들락거릴 때 자주 발생합니다.
- **내부 단편화(Internal Fragmentation)**: 할당된 메모리 블록이 프로세스가 실제로 필요한 크기보다 커서, 할당된 블록 내부에 사용되지 않고 낭비되는 공간이 생기는 문제입니다. 예를 들어, 4KB 단위로 메모리를 할당하는데 프로세스가 3KB만 필요로 한다면, 1KB는 사용되지 않고 버려지게 됩니다.

페이징(Paging) 기법은 이러한 단편화 문제를 해결하기 위해 도입된 주 메모리 관리 기법입니다.

- **페이징의 원리**: 물리적 메모리(RAM)를 고정된 크기의 '프레임(Frame)'으로 나누고, 논리적 메모리(프로세스의 주소 공간)를 동일한 크기의 '페이지(Page)'로 나눕니다. 프로세스가 실행될 때, 각 페이지는 물리 메모리의 비어있는 아무 프레임에 로드될 수 있습니다. 이 매핑 정보는 '페이지 테이블(Page Table)'에 기록됩니다.
- **단편화 문제 해결**:
- **외부 단편화 해결**: 페이징은 물리 메모리를 고정된 크기의 프레임으로 나누고 페이지를 프레임에 불연속적으로 할당할 수 있기 때문에, 총 여유 공간이 충분하다면 연속적이지 않아도 할당이 가능하여 외부 단편화 문제를 **근본적으로 해결**합니다.
- **내부 단편화 발생**: 그러나 페이징은 페이지 크기 단위로 메모리를 할당하므로, 프로세스의 마지막 페이지가 페이지 크기만큼 딱 떨어지지 않을 경우, 마지막 프레임 내에 남는 공간이 발생하여 **내부 단편화**가 발생할 수 있습니다. 예를 들어, 페이지 크기가 4KB인데 프로세스가 12KB + 1바이트를 필요로 한다면, 마지막 1바이트를 위해 4KB 프레임 전체를 할당받게 되고, 이 중 약 4KB - 1바이트가 내부 단편화로 낭비됩니다. 일반적으로 내부 단편화의 평균 크기는 페이지 크기의 절반 정도로 예상됩니다.

결론적으로, 페이징은 외부 단편화 문제를 완벽히 해결하지만, 내부 단편화를 완전히 없애지는 못합니다. 하지만 내부 단편화의 낭비는 외부 단편화의 심각성보다는 일반적으로 덜 문제가 됩니다.

3. 페이지 테이블(Page Table)의 역할은 무엇이며, TLB(Translation Lookaside Buffer)는 어떤 문제를 해결하기 위해 사용되나요?

페이지 테이블(Page Table)은 페이징 기법에서 논리적 주소(프로세스가 사용하는 가상 주소)를 물리적 주소(실제 메모리 주소)로 변환하는 데 사용되는 핵심 데이터 구조입니다.

- **역할**: 논리 주소는 '페이지 번호(p)'와 '페이지 오프셋(d)'으로 구성됩니다. 페이지 테이블은 각 페이지 번호에 해당하는 물리적 메모리 내의 '프레임 번호(f)'를 저장합니다. CPU가 특정 논리 주소에 접근할 때, 페이지 번호를 사용하여 페이지 테이블에서 해당 프레임 번호를 찾고, 이 프레임 번호와 페이지 오프셋을 결합하여 실제 물리적 주소를 생성합니다. 즉, (논리 주소: p, d) -> (페이지 테이블 참조: f) -> (물리 주소: f, d)의 변환을 수행합니다. 페이지 테이블에는 유효-무효 비트, 보호 비트 등 페이지 접근 권한 및 상태 정보도 포함됩니다.

TLB(Translation Lookaside Buffer)는 페이지 테이블 접근으로 인한 성능 저하를 해결하기 위한 고속 캐시 메모리입니다.

- **문제점**: 페이지 테이블은 일반적으로 주 메모리(RAM)에 저장됩니다. 따라서 CPU가 어떤 데이터나 명령어를 접근할 때마다 논리 주소를 물리 주소로 변환하기 위해 최소 두 번의 주 메모리 접근이 필요하게 됩니다. 한 번은 페이지 테이블을 참조하기 위함이고, 다른 한 번은 실제 데이터가 있는 프레임에 접근하기 위함입니다. 이러한 이중 메모리 접근은 시스템 성능을 크게 저하시킵니다.
- **TLB의 해결책**: TLB는 CPU 내부에 위치한 작고 빠른 하드웨어 캐시입니다. 최근에 사용된 페이지 변환 정보(페이지 번호와 해당 프레임 번호의 매핑)를 저장합니다.
- **TLB 히트(Hit)**: CPU가 논리 주소를 변환할 때, 먼저 TLB를 검색합니다. 만약 TLB에 해당 페이지 변환 정보가 있다면(TLB 히트), 페이지 테이블을 주 메모리에서 참조할 필요 없이 TLB에서 바로 프레임 번호를 얻어 물리 주소를 생성합니다. 이 경우 메모리 접근은 한 번만 이루어지므로 변환 속도가 매우 빠릅니다.

- **TLB 미스(Miss)**: TLB에 원하는 정보가 없다면(TLB 미스), 주 메모리에 있는 페이지 테이블을 참조하여 프레임 번호를 얻고, 이 정보를 TLB에도 저장하여 다음에 다시 사용될 때 빠르게 접근할 수 있도록 합니다.
- **성능 영향**: TLB의 히트율(Hit Ratio)은 시스템 성능에 지대한 영향을 미칩니다. 히트율이 높을수록 메모리 접근 시간이 단축되어 전반적인 시스템 처리량이 향상됩니다.

4. 교착 상태(Deadlock)의 네 가지 필요 조건(Coffman Conditions)은 무엇이며, 이들을 해결하기 위한 접근 방식 세 가지를 설명해주세요.

교착 상태(Deadlock)가 발생하려면 다음 네 가지 조건이 동시에 충족되어야 합니다. 이를 코프만 조건(Coffman Conditions)이라고 합니다:

1. **상호 배제(Mutual Exclusion)**: 최소한 하나의 자원이 공유 불가능한 모드(non-sharable mode)로 점유되어야 합니다. 즉, 한 번에 하나의 프로세스만 그 자원을 사용할 수 있습니다. 뮤텍스 락과 같은 자원이 대표적입니다.
2. **점유와 대기(Hold and Wait)**: 자원을 이미 하나 이상 점유하고 있는 프로세스가 다른 프로세스가 점유하고 있는 추가적인 자원을 얻기 위해 대기해야 합니다.
3. **비선점(No Preemption)**: 자원은 강제로 빼앗을 수 없으며, 이를 점유하고 있는 프로세스가 작업을 완료하고 자발적으로 해제하기 전까지는 다른 프로세스가 자원을 사용할 수 없습니다.
4. **순환 대기(Circular Wait)**: 프로세스 집합 {P0, P1, ..., Pn}이 있을 때, P0는 P1이 점유하고 있는 자원을 대기하고, P1은 P2가 점유하고 있는 자원을 대기하는 식으로, 최종적으로 Pn은 P0가 점유하고 있는 자원을 대기하는 원형 체인이 형성되어야 합니다.

교착 상태를 처리하는 주요 접근 방식은 세 가지입니다:

1. **교착 상태 예방(Deadlock Prevention)**: 교착 상태의 네 가지 필요 조건 중 하나 이상을 애초에 발생하지 않도록 시스템을 설계하여 교착 상태를 원천적으로 차단하는 방법입니다.

- *예시*: "상호 배제"를 깨기 위해 모든 자원을 공유 가능하게 만들거나, "점유와 대기"를 깨기 위해 프로세스가 모든 필요한 자원을 한 번에 요청하고 할당받거나 아무것도 가지지 않은 상태에서만 요청하도록 강제합니다. "비선점"을 깨기 위해 추가 자원을 요청 시 현재 보유한 자원을 모두 해제하도록 하거나, "순환 대기"를 깨기 위해 자원 유형에 순서를 부여하여 오름차순으로만 요청하도록 합니다. 이 방법은 안전하지만, 자원 활용률이 낮아지거나 시스템 처리량/응답성이 저하될 수 있는 오버헤드가 있습니다.

1. **교착 상태 회피(Deadlock Avoidance)**: 시스템이 항상 '안전 상태(Safe State)'를 유지하도록 자원 할당을 동적으로 제어하는 방법입니다. '안전 상태'란 교착 상태 없이 모든 프로세스가 완료될 수 있는 순서(안전 순서)가 존재하는 상태를 의미합니다.

- *예시*: **은행가 알고리즘(Banker's Algorithm)**이 대표적입니다. 이 알고리즘은 각 프로세스의 최대 자원 요구량을 미리 알고 있다고 가정하고, 새로운 자원 할당 요청이 들어올 때마다 시스템이 안전 상태로 유지될 수 있는지 검사합니다. 만약 할당 후 시스템이 불안전 상태가 될 가능성이 있다면, 해당 요청을 거부하고 프로세스를 대기시킵니다. 이 방법은 예방보다 덜 제한적이지만, 모든 프로세스의 최대 자원 요구량을 미리 알아야 한다는 제약이 있습니다.

1. **교착 상태 탐지 및 복구(Deadlock Detection & Recovery)**: 교착 상태 발생을 허용하되, 주기적으로 시스템을 검사하여 교착 상태가 발생했는지 탐지하고, 발생 시 이를 해결하여 시스템을 복구하는 방법입니다.

- *탐지*: 단일 인스턴스 자원의 경우 '대기 그래프(Wait-for Graph)'에서 사이클을 검사하거나, 다중 인스턴스 자원의 경우 은행가 알고리즘과 유사한 탐지 알고리즘을 사용합니다.
- *복구*: 교착 상태가 탐지되면, 이를 해결하기 위해 교착 상태에 연루된 프로세스를 강제로 종료(abort)시키거나, 자원을 선점(preemption)하여 다른 프로세스에게 할당하는 방식으로 시스템을 정상화합니다. 이 과정에서 어떤 프로세스를 희생시킬지(victim selection) 결정해야 하며, 이는 보통 비용(Cost)을 최소화하는 방향으로 이루어집니다.

대부분의 현대 운영체제(Windows, Linux 등)는 교착 상태 발생 빈도가 매우 낮고 예방/회피의 오버헤드가 크기 때문에, 성능상의 이유로 교착 상태를 '무시하는(Ignorance)' 접근법을 사용하는 경우가 많습니다.

5. 생산자-소비자 문제에서 동기화 문제점들은 무엇이며, 이를 해결하기 위해 뮤텍스와 세마포어는 어떻게 함께 사용되나요?

생산자-소비자 문제(Producer-Consumer Problem)는 유한한 크기의 버퍼를 공유하여 데이터를 생산하는 생산자 스레드와 소비하는 소비자 스레드 간의 동기화를 다루는 고전적인 문제입니다. 여기서 발생할 수 있는 주요 동기화 문제점들은 다음과 같습니다:

1. **경쟁 상태(Race Condition)**: 여러 생산자나 소비자가 동시에 버퍼의 공유 변수(예: count, in/out 인덱스)나 버퍼 자체에 접근하여 수정하려 할 때 발생합니다. 이로 인해 버퍼의 데이터가 손상되거나, 포인터가 잘못되거나, 아이템 수가 부정확해질 수 있습니다.
2. **버퍼 오버플로우(Buffer Overflow)**: 생산자가 가득 찬 버퍼에 데이터를 추가하려고 시도하는 경우입니다. 적절한 동기화가 없으면 기존 데이터를 덮어쓰거나 오류가 발생할 수 있습니다.
3. **버퍼 언더플로우(Buffer Underflow)**: 소비자가 텅 빈 버퍼에서 데이터를 추출하려고 시도하는 경우입니다. 동기화가 없으면 유효하지 않은 데이터를 읽거나 오류가 발생할 수 있습니다.
4. **바쁜 대기(Busy Waiting)**: 생산자가 버퍼가 가득 찼을 때, 또는 소비자가 버퍼가 비었을 때 단순히 반복문을 돌며 버퍼 상태를 계속 확인하는 방식입니다. 이는 CPU 자원을 심각하게 낭비합니다.
5. **교착 상태(Deadlock)**: 잘못된 잠금 및 신호 순서로 인해 생산자와 소비자가 서로를 무한정 기다리는 상태에 빠질 수 있습니다.

뮤텍스와 세마포어를 함께 사용한 해결 방법: 생산자-소비자 문제를 해결하기 위해 일반적으로 세 개의 동기화 프리미티브를 사용합니다:

1. **mutex (뮤텍스)**

- **종류**: 이진 세마포어 또는 뮤텍스 객체.
- **초기값**: 1 (열림 상태).
- **역할**: 버퍼 자체(데이터 배열, in/out 인덱스, count 변수 등 모든 공유 자원)에 대한 **상호 배제**적 접근을 보장합니다. 즉, 한 번에 하나의 스레드만 버퍼를 조작할 수 있도록 하여 경쟁 상태를 방지합니다.
- **wait/signal 위치**: 생산자와 소비자 모두 버퍼에 아이템을 추가하거나 제거하는 실제 작업(임계 구역) **직전에 wait(mutex)를 호출**하고, 작업 **직후에 signal(mutex)를 호출**합니다.

1. **empty (계수 세마포어)**

- **종류**: 계수 세마포어 (Counting Semaphore).
- **초기값**: 버퍼의 최대 크기 N.
- **역할**: 버퍼 내 사용 가능한 (비어있는) 슬롯의 수를 나타냅니다. 생산자는 아이템을 추가하기 전에 이 세마포어 값이 0보다 큰지 확인하여 **오버플로우를 방지**합니다.
- **wait/signal 위치**:

- **생산자**: 아이템을 생산한 후, mutex를 획득하기 **전에 wait(empty)를 호출**합니다. 이는 버퍼에 빈 공간이 생길 때까지 생산자를 대기(블록)시킵니다.
- **소비자**: 아이템을 소비한 후, mutex를 해제한 **후에 signal(empty)를 호출**합니다. 이는 버퍼에 빈 공간이 하나 생겼음을 알립니다 (생산자를 깨울 수 있음).

1. **full (계수 세마포어)**

- **종류**: 계수 세마포어.
- **초기값**: 0.
- **역할**: 버퍼 내 채워진 (소비 가능한 아이템이 있는) 슬롯의 수를 나타냅니다. 소비자는 아이템을 제거하기 전에 이 세마포어 값이 0보다 큰지 확인하여 **언더플로우를 방지**합니다.
- **wait/signal 위치**:
- **소비자**: 아이템을 소비하기 위해, mutex를 획득하기 **전에 wait(full)을 호출**합니다. 이는 버퍼에 소비할 아이템이 생길 때까지 소비자를 대기(블록)시킵니다.
- **생산자**: 아이템을 버퍼에 추가한 후, mutex를 해제한 **후에 signal(full)을 호출**합니다. 이는 버퍼에 새 아이템이 하나 추가되었음을 알립니다 (소비자를 깨울 수 있음).

주의사항: empty와 full 세마포어에 대한 wait 연산은 반드시 mutex 획득 전에 이루어져야 합니다. 만약 mutex를 획득한 상태에서 empty나 full 세마포어에서 대기하게 되면, 다른 스레드가 mutex를 획득할 수 없어 해당 세마포어를 signal 해주지 못하므로 교착 상태가 발생할 수 있습니다. 이는 잠금의 범위를 적절히 설정하는 '세분화된 잠금(fine-grained locking)'의 중요성을 보여줍니다.

6. 뮤텍스(Mutex)와 세마포어(Semaphore)의 주요 차이점은 무엇인가요?

뮤텍스와 세마포어는 모두 동기화를 위한 중요한 프리미티브이지만, 다음과 같은 주요 차이점이 있습니다:

1. **주 사용 목적**:

- **뮤텍스**: 주로 **상호 배제(Mutual Exclusion)**를 위해 사용됩니다. 즉, 하나의 공유 자원 또는 임계 구역에 한 번에 하나의 스레드만 접근하도록 제어하는 것이 목적입니다. '잠금(lock)'의 개념에 가깝습니다.
- **세마포어**: 뮤텍스보다 일반적인 동기화 도구로, **자원의 가용 개수를 관리**하거나 **스레드 간의 실행 순서를 제어**하고 신호를 전달하는 데 사용됩니다.

1. **소유권 개념**:

- **뮤텍스**: 일반적으로 **소유권(ownership)** 개념이 있습니다. 즉, 뮤텍스를 잠근(획득한) 스레드만이 해당 뮤텍스를 해제(반환)할 수 있습니다.
- **세마포어**: 명시적인 소유권 개념이 없습니다. 한 스레드가 wait (P) 연산을 통해 세마포어 값을 감소시키고, 다른 스레드가 signal (V) 연산을 통해 값을 증가시켜 대기 중인 스레드를 깨울 수 있습니다.

1. **자원 개수 관리**:

- **뮤텍스**: 관리하는 자원의 상태는 잠김(locked) 또는 열림(unlocked)의 두 가지뿐이므로, 개념적으로 **하나의 자원**만을 관리합니다 (이진 세마포어와 유사).
- **세마포어**: 내부적으로 정수형 카운터 값을 가지며, 이 값은 0 이상의 **가용 자원의 개수**를 나타낼 수 있습니다 (계수 세마포어). 이진 세마포어는 0 또는 1의 값만 가집니다.

1. **signal (또는 unlock/release) 연산의 주체**:

- **뮤텍스**: 소유권 개념 때문에, **락을 획득한 스레드만이** 해당 락을 해제할 수 있습니다.

- **세마포어**: 락을 획득(세마포어 값을 감소)한 스레드가 아니더라도 **다른 스레드가 signal 연산을 통해** 세마포어 값을 증가시키고 대기 중인 스레드를 깨울 수 있습니다.

대표적인 사용 사례:

- **뮤텍스**: 공유 변수나 공유 자료구조(예: 링크드 리스트)에 대한 접근을 직렬화하여 경쟁 상태를 방지하는 **임계 구역 보호**.
- **세마포어**: 생산자-소비자 문제에서 유한 버퍼의 빈 공간 수(empty)와 채워진 아이템 수(full)를 관리하고, 생산자와 소비자 간의 작업을 조율하는 데 사용. 또는 동시에 접근 가능한 리소스의 개수를 제한하는 경우(예: 데이터베이스 커넥션 풀의 커넥션 수 제한).

7. 멀티코어 환경에서 캐시 일관성(Cache Coherence) 문제가 발생하는 이유와 MESI 프로토콜은 어떻게 이를 해결하나요?

캐시 일관성 문제 발생 이유: 멀티코어 프로세서에서 각 코어는 자신만의 독립적인 로컬 캐시(L1, L2 캐시 등)를 가집니다. 여러 코어가 동일한 주 메모리 위치에 저장된 데이터를 각자의 캐시에 복사본으로 가지고 있을 때, 한 코어가 자신의 캐시에 있는 데이터 복사본을 수정하게 되면, 다른 코어의 캐시에 있는 동일 데이터의 복사본은 더 이상 최신 값이 아니게 되어 '오래된 데이터(stale data)'가 됩니다. 이러한 데이터 불일치 상태가 발생하면 프로그램이 잘못된 데이터를 읽어 오작동할 수 있는데, 이를 캐시 일관성 문제라고 합니다.

MESI 프로토콜을 통한 해결: MESI(Modified, Exclusive, Shared, Invalid) 프로토콜은 스누핑(snooping) 기반의 캐시 일관성 프로토콜 중 하나로, 각 캐시 라인의 상태를 4가지로 정의하여 데이터 일관성을 유지합니다.

- **Modified (M)**:
- **상태**: 해당 캐시 라인이 현재 캐시에만 존재하며, 메인 메모리의 내용과 다르게 **수정된(dirty)** 상태입니다.
- **특징**: 다른 코어가 이 데이터를 읽으려면 먼저 이 캐시의 데이터가 메인 메모리에 반영(write-back)되어야 합니다.
- **Exclusive (E)**:
- **상태**: 해당 캐시 라인이 현재 캐시에만 존재하며, 메인 메모리의 내용과 **동일한(clean)** 상태입니다.
- **특징**: 다른 코어의 개입 없이 자유롭게 읽거나 쓸 수 있습니다 (쓰기 시 Modified 상태로 변경).
- **Shared (S)**:
- **상태**: 해당 캐시 라인이 여러 캐시에 존재할 수 있으며, 메인 메모리의 내용과 **동일한(clean)** 상태입니다.
- **특징**: 읽기는 자유롭지만, 쓰기를 하려면 다른 캐시의 복사본을 무효화(Invalidate)하고 Modified 상태로 변경해야 합니다.
- **Invalid (I)**:
- **상태**: 해당 캐시 라인이 유효하지 않거나 사용되지 않는 상태입니다.
- **특징**: 이 상태의 데이터는 사용할 수 없으며, 접근 시 메인 메모리 또는 다른 캐시에서 최신 데이터를 가져와야 합니다.

'Shared' 상태 데이터에 대한 쓰기 연산 시 일관성 유지 과정: 한 코어(예: Core A)가 특정 메모리 위치에 쓰기 연산을 수행하려 하고, 다른 코어(예: Core B)의 캐시에 해당 데이터가 'Shared (S)' 상태로 존재한다고 가정합니다.

1. **독점적 소유권 요청**: Core A는 쓰기 작업을 수행하기 위해 해당 캐시 라인에 대한 독점적인 소유권을 가져야 합니다.

2. **무효화 메시지 전송**: Core A는 공유 버스를 통해 다른 모든 코어에게 해당 캐시 라인을 무효화하라는 메시지(예: BusUpgr 또는 RFO - Request For Ownership)를 보냅니다.
3. **스누핑 및 상태 변경**: Core B는 이 메시지를 스누핑(snooping)하여 감지하고, 자신의 캐시에 있는 해당 라인을 'Invalid (I)' 상태로 변경합니다. (모든 공유된 복사본이 무효화됨)
4. **쓰기 및 상태 변경**: 모든 다른 코어의 복사본이 무효화되면, Core A는 해당 캐시 라인의 상태를 'Modified (M)'으로 변경하고 안전하게 쓰기 작업을 수행합니다.

이러한 과정을 통해 다른 코어가 오래된 데이터를 사용하는 것을 방지하고 데이터 일관성을 유지합니다. ** 메모리 배리어(Memory Barrier)**는 현대 CPU와 컴파일러가 성능 최적화를 위해 명령어 실행 순서를 재배치하거나 쓰기 버퍼(Write Buffer) 등을 사용하는 것에 대해, 특정 지점을 기준으로 이전의 메모리 연산들이 이후의 연산들보다 먼저 완료되거나 다른 코어에 관찰되도록 강제하여 동기화 로직이 올바르게 동작하고 데이터 일관성을 보장하는 데 필요합니다.

8. 스핀락(Spinlock)은 무엇이며, 단일 코어 시스템에서 사용 시 발생할 수 있는 문제점은 무엇인가요?

스핀락(Spinlock)은 운영체제에서 동기화를 위해 사용되는 락(lock)의 한 종류입니다.

- **개념**: 스핀락은 임계 구역(Critical Section)에 진입하려는 스레드가 락이 잠겨 있을 때, CPU를 점유한 채(CPU를 양보하지 않고) 반복적으로 락의 상태를 확인하며 대기(busy-waiting)하는 방식입니다. 마치 문이 열릴 때까지 계속해서 문고리를 흔들어 보는 것과 같습니다.
- **작동 방식**: Test-and-Set이나 Compare-and-Swap과 같은 하드웨어 수준의 원자적(atomic) 명령어를 사용하여 락의 상태를 검사하고 변경합니다.

단일 코어 시스템에서 스핀락 사용 시 문제점: 스핀락은 멀티코어 환경, 특히 락을 보유한 스레드가 다른 코어에서 빠르게 락을 해제할 수 있는 매우 짧은 임계 구역에 효과적일 수 있습니다. 문맥 교환(Context Switch) 오버헤드가 발생하지 않는다는 장점이 있기 때문입니다.

하지만 **단일 코어 시스템에서는 스핀락의 사용이 심각한 성능 저하를 초래할 수 있습니다.**

- **CPU 낭비 및 진행 방해**: 단일 코어 시스템에서 스핀락을 획득하려는 스레드가 락을 점유한 채 busy-waiting을 하게 되면, 그 스레드는 CPU를 놓지 않고 계속 반복적으로 락이 해제되기를 기다립니다. 이 과정에서 **정작 락을 보유하고 있는 스레드(만약 CPU가 그 스레드에게 할당되어야 한다면)**가 CPU를 할당받지 못하게 됩니다. 결과적으로 락을 해제해야 할 스레드가 실행되지 못하여 락이 영원히 해제되지 않거나, 시스템 전체의 진행이 극도로 지연되는 '교착 상태와 유사한 상황'을 유발할 수 있습니다.
- **비효율성**: 문맥 교환 오버헤드는 피할 수 있지만, 락을 기다리는 동안 CPU 사이클을 낭비하게 되어 전체 시스템의 처리량이 오히려 감소합니다.

따라서 단일 코어 시스템에서는 스핀락 대신 **슬립락(sleep lock)** 방식이 더 적합합니다. 슬립락(예: 뮤텍스)은 락을 획득하지 못하면 해당 스레드를 대기 큐에 넣고 CPU를 양보(block)하여 다른 스레드가 실행될 수 있도록 합니다. 이는 CPU 자원을 효율적으로 사용하여 전반적인 시스템 성능을 향상시킵니다.

1. CPU 스케줄링

- **CPU-I/O 버스트 사이클**: 프로세스 실행은 CPU 실행과 I/O 대기 구간이 번갈아 나타나는 사이클로 구성됩니다. 대부분의 프로세스는 짧은 CPU 버스트가 많고 긴 CPU 버스트가 적은 패턴을 보입니다.
- **CPU 스케줄러**: 준비 큐에서 프로세스를 선택하여 CPU 코어를 할당합니다.
- **디스패처**: 스케줄러가 선택한 프로세스에게 실제로 CPU 제어권을 전달하는 모듈로, 컨텍스트 스위칭, 모드 전환, 프로그램 카운터 점프 등을 수행합니다.
- **스케줄링 기준**: CPU 활용률, 처리량, 반환 시간, 대기 시간, 응답 시간 등.
- **스케줄링 알고리즘**:
- **선입선출 (FCFS)**: 가장 단순한 비선점형 알고리즘. 호송 효과(convoy effect) 발생 가능.
- **최단 작업 우선 (SJF)**: 평균 대기 시간이 최적인 비선점형 알고리즘. 선점형 버전은 SRTF(Shortest Remaining Time First)라고 합니다.
- **우선순위 스케줄링**: 우선순위가 높은 프로세스 먼저 실행. 기아 현상 방지를 위해 에이징 기법 사용 가능.
- **라운드 로빈 (RR)**: 시분할 시스템의 핵심 알고리즘. 각 프로세스에 동일한 시간 할당량을 부여하여 순환 실행. 컨텍스트 스위치 오버헤드 발생.
- **다단계 큐 스케줄링**: 프로세스 유형별로 별도의 큐를 사용하며, 각 큐마다 다른 스케줄링 알고리즘 적용 가능.
- **다단계 피드백 큐**: 프로세스 큐 간 이동 가능. 실행 특성에 따라 동적으로 우선순위 조정.
- **CPU 버스트 길이 예측 (지수 평균)**:
- $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
- t_n: n번째 실제 CPU 버스트 길이
- τ_n: n번째 예측된 CPU 버스트 길이
- α: 가중치 (일반적으로 0과 1 사이, 흔히 1/2로 설정).
- $\alpha = 0$: 최근 이력 무시, $\tau_{n+1} = \tau_n$.
- $\alpha = 1$: 오직 실제 마지막 CPU 버스트만 고려, $\tau_{n+1} = \alpha t_n$.
- 확장 시 각 연속 항은 이전 항보다 적은 가중치를 가짐.

2. 동기화 도구

- **경합 조건 (Race Condition)**: 여러 프로세스가 동시에 공유 데이터에 접근할 때, 실행 순서에 따라 결과가 달라지는 상황.
- **임계 구역 (Critical Section)**: n개의 프로세스가 공통 변수를 변경하거나 테이블을 업데이트하는 등의 작업을 수행하는 코드 세그먼트.
- **해결책 요구사항**: 상호 배제, 진행, 유한 대기.
- **하드웨어 지원**: Test-and-Set, Compare-and-Swap (CAS) 명령어.
- **뮤텍스 락 (Mutex Lock)**: 가장 간단한 동기화 도구. acquire()와 release() 함수 제공. 바쁜 대기 (busy-waiting)를 사용하므로 스핀락(Spinlock)이라고도 불림.
- **세마포어 (Semaphore)**: 정수 변수 S와 wait() (P), signal() (V) 두 원자적 연산으로 구성.
- **종류**: 카운팅 세마포어, 이진 세마포어.
- **바쁜 대기 없는 구현**: 각 세마포어마다 대기 큐 유지, block 및 wakeup 연산 사용.
- **모니터 (Monitor)**: 프로세스 동기화를 위한 고수준 추상화 메커니즘. 조건 변수(condition variable)와 함께 사용.
- **고전적인 동기화 문제**: 유한 버퍼 문제, 읽기-쓰기 문제, 식사하는 철학자 문제.
- **식사하는 철학자 문제**: 모니터 구현 예시 (DiningPhilosophers 모니터 내 pickup, putdown, test 함수, state 배열, self 조건 변수).

3. 교착상태 (Deadlock)

- **교착상태의 4가지 필요조건 (Coffman Conditions)**: 이 네 가지 조건이 모두 동시에 만족될 때 교착상태 발생 가능.

1. **상호 배제 (Mutual Exclusion)**: 한 번에 하나의 프로세스만 자원 사용 가능.
2. **점유와 대기 (Hold and Wait)**: 자원을 보유한 채로 다른 자원을 대기.
3. **비선점 (No Preemption)**: 자원을 강제로 빼앗을 수 없음.
4. **순환 대기 (Circular Wait)**: 프로세스들이 원형으로 서로의 자원을 대기.

- **자원 할당 그래프 (Resource-Allocation Graph)**: 프로세스와 자원 간의 관계를 나타내는 방향 그래프. 요청 간선과 할당 간선으로 구성.
- 사이클이 없으면 교착상태가 없고, 사이클이 있으면 교착상태 가능성이 있습니다 (모든 인스턴스가 하나인 자원 유형의 경우 사이클이 곧 교착상태를 의미).
- **교착상태 처리 방법**:
- **예방 (Prevention)**: 4가지 필요조건 중 하나를 무효화하여 원천 차단 (점유와 대기 무효화, 비선점 무효화, 순환 대기 무효화).
- **회피 (Avoidance)**: 시스템이 안전 상태를 유지하도록 동적으로 자원 할당을 제어.
- **은행가 알고리즘 (Banker's Algorithm)**: 각 프로세스의 최대 자원 요구량을 미리 알고 있다고 가정하고, 자원 할당 요청 시 안전성 검사. (Available, Max, Allocation, Need 행렬 사용).
- **탐지 및 복구 (Detection & Recovery)**: 교착상태 발생을 허용하되 주기적으로 탐지하여 복구.
- **탐지**: 대기 그래프의 사이클 검사 또는 다중 인스턴스용 탐지 알고리즘 사용.
- **복구**: 프로세스 종료 또는 자원 선점을 통해 교착상태 해결.
- **실제 시스템에서의 적용**: 대부분의 현대 OS는 성능상의 이유로 교착상태를 무시하는 접근법 사용.
- **라이블락 (Livelock)**: 스레드가 바쁘게 상태를 변경하며 계속 실행되지만, 실제 유용한 작업을 진행하지 못하는 상황. 데드락과 달리 스레드들이 멈춰있지 않고 활성 상태.

4. 주 메모리 관리

- **메모리 계층 구조**: CPU 레지스터, 캐시, 주 메모리, 비휘발성 메모리(하드 디스크, 마그네틱 테이프, 광 디스크) 순으로 구성. 속도와 용량이 반비례.
- **메모리 보호**:
- **베이스-리밋 레지스터**: 베이스 레지스터는 프로세스의 시작 주소를, 리밋 레지스터는 프로세스 크기를 저장하여 메모리 접근을 제한.
- **MMU (Memory Management Unit)**: 논리 주소를 물리 주소로 변환하는 하드웨어 장치. 모든 메모리 접근을 검사하여 보호 위반 시 트랩 발생.
- **논리 주소와 물리 주소**:
- **논리 주소 (Logical Address)**: 프로그램이 생성하는 가상 주소.
- **물리 주소 (Physical Address)**: 실제 메모리 하드웨어의 주소.
- **동적 로딩 및 연결**:
- **동적 로딩**: 프로그램 실행 중 필요한 루틴만 메모리에 로드하여 효율적 메모리 활용.
- **동적 연결**: 실행 시점에 라이브러리와 연결하여 실행 파일 크기 줄이고 공유 라이브러리 사용.
- **연속 메모리 할당**: 주 메모리를 운영체제 영역과 사용자 프로세스 영역으로 분할. 가변 분할 방식에서 홀(빈 공간)과 단편화(Fragmentation) 발생.
- **외부 단편화 (External Fragmentation)**: 총 여유 공간은 충분하지만 연속되지 않아 할당 불가.
- **내부 단편화 (Internal Fragmentation)**: 할당된 공간이 요청보다 커서 일부가 낭비됨.

- **동적 저장소 할당 알고리즘**: First Fit, Best Fit, Worst Fit.
- **압축 (Compaction)**: 외부 단편화 해결을 위해 메모리 재배치 (비용 높음).
- **페이징 (Paging)**: 물리 메모리를 고정 크기 프레임으로, 논리 메모리를 같은 크기의 페이지로 나누어 외부 단편화 해결.
- **페이지 (Page)**: 논리 메모리(프로세스의 주소 공간)를 고정된 작은 블록으로 나눈 것.
- **프레임 (Frame)**: 물리 메모리를 고정된 작은 블록으로 나눈 것. 페이지와 동일한 크기.
- **페이지 테이블 (Page Table)**: 논리 주소(페이지 번호와 오프셋)를 물리 주소(프레임 번호와 오프셋)로 변환하는 데 사용되는 데이터 구조. 각 프로세스마다 하나씩 가짐.
- 페이지 번호(p)와 페이지 오프셋(d)으로 구성.
- 항목은 프레임 번호 외에도 유효/무효 비트(Valid/Invalid bit), 보호 비트(Protection bit), 더티 비트(Dirty bit), 참조 비트(Reference bit) 등을 포함.
- **TLB (Translation Lookaside Buffer)**: 페이지 테이블 접근 횟수를 줄이기 위한 고속 캐시. 최근 사용된 페이지 변환 정보를 저장. TLB 히트율이 성능에 큰 영향.
- **페이지 테이블 구조**:
- **계층적 페이지 테이블 (Hierarchical Paging)**: 페이지 테이블의 크기 문제를 해결하기 위해 여러 단계로 나눔 (예: 2단계 페이징).
- **해시 페이지 테이블 (Hashed Page Table)**: 희소 주소 공간에서 효과적. 가상 페이지 번호를 해시하여 물리 프레임 찾음.
- **역 페이지 테이블 (Inverted Page Table)**: 물리 메모리 기준으로 구성되어 시스템당 하나만 존재. 각 프레임이 어떤 프로세스의 어떤 페이지에 할당되었는지 기록.
- **스와핑 (Swapping)**: 메모리 부족 시 프로세스를 일시적으로 보조 저장소(하드 디스크)로 이동시키는 기법.

퀴즈 (단답형)

1. CPU 버스트 길이 예측에 사용되는 지수 평균 공식에서, α 값이 0일 때와 1일 때 예측 결과가 어떻게 달라지는지 설명하시오.
2. 뮤텍스 락과 스핀락은 어떤 관계를 가지며, 스핀락이 단일 코어 시스템에서 사용될 때 발생할 수 있는 주요 문제점은 무엇인지 서술하시오.
3. 교착상태 발생의 4가지 필요조건 중 '순환 대기'는 무엇을 의미하는지 간략하게 설명하시오.
4. 은행가 알고리즘에서 Need 행렬은 어떤 정보를 나타내며, 이 정보는 어떻게 계산되는지 설명하시오.
5. 페이징 시스템에서 '외부 단편화'는 발생하지 않지만 '내부 단편화'가 발생할 수 있는 주된 이유는 무엇인지 서술하시오.
6. 페이지 테이블 엔트리에 포함되는 'Valid/Invalid bit'의 역할은 무엇이며, 이 비트가 'Invalid'일 때 어떤 상황이 발생하는지 설명하시오.
7. TLB(Translation Lookaside Buffer)는 페이징 시스템의 성능 향상을 위해 사용됩니다. TLB가 없다면 메모리 접근 시 어떤 비효율이 발생하는지 서술하시오.
8. 식사하는 철학자 문제에서 모니터(DiningPhilosophers)를 사용한 해결책에서, test(i) 함수는 어떤 조건을 확인하여 철학자 i가 식사를 할 수 있도록 허용하는지 설명하시오.
9. 자원 할당 그래프에서 사이클의 존재가 반드시 교착상태를 의미하지 않는 경우의 예시를 자원 유형의 인스턴스 개수와 연관하여 설명하시오.
10. 읽기-쓰기 락(Read-Write Lock)이 뮤텍스보다 성능 이점을 가질 수 있는 주된 시나리오는 무엇이며, 어떤 상황에서 기아(starvation) 문제가 발생할 수 있는지 설명하시오.

퀴즈 정답

1. CPU 버스트 길이 예측에 사용되는 지수 평균 공식에서, α 값이 0일 때는 예측값 τ_{n+1}이 이전 예측값 τ_n과 동일해져 최근 실제 이력이 전혀 반영되지 않습니다. 반면, α 값이 1일 때는 예측값 τ_{n+1}이 오직 직전 실제 CPU 버스트 길이 t_n에만 의존하게 됩니다.
2. 뮤텍스 락은 동기화 도구의 일종이며, 스핀락은 뮤텍스 락의 한 구현 방식입니다. 스핀락은 임계 구역 진입이 불가능할 때 CPU를 점유한 채 반복적으로 조건을 확인하며 대기하는데, 단일 코어 시스템에서는 대기하는 스레드가 CPU를 계속 점유하여 잠금을 보유한 다른 스레드의 실행을 방해하여 시스템 전체 성능을 저하시킬 수 있습니다.
3. '순환 대기'는 교착상태 발생의 4가지 필요조건 중 하나로, 프로세스들이 원형으로 서로가 보유하고 있는 자원을 대기하는 상황을 의미합니다. 즉, P0는 P1의 자원을, P1은 P2의 자원을, ..., Pn은 P0의 자원을 기다리는 형태의 순환 의존 관계가 형성됩니다.
4. 은행가 알고리즘에서 Need 행렬은 각 프로세스가 작업을 완료하기 위해 추가적으로 필요한 각 자원 유형의 인스턴스 수를 나타냅니다. 이 값은 프로세스의 Max 요구량에서 현재 Allocation된 자원량을 뺀 값으로 계산됩니다 (Need[i][j] = Max[i][j] - Allocation[i][j]).
5. 페이징 시스템은 물리 메모리를 고정된 크기의 '프레임'으로, 논리 메모리를 동일한 크기의 '페이지'로 나누어 할당하기 때문에 연속적인 빈 공간을 찾을 필요가 없어 외부 단편화가 발생하지 않습니다. 그러나 프로세스의 마지막 페이지가 프레임 크기보다 작을 경우, 할당된 프레임의 일부 공간이 사용되지 않고 낭비되는데, 이를 내부 단편화라고 합니다.
6. 페이지 테이블 엔트리의 'Valid/Invalid bit'는 해당 페이지가 현재 물리 메모리에 적재되어 유효한지(Valid), 아니면 유효하지 않은 상태(Invalid)인지를 나타냅니다. 이 비트가 'Invalid'일 경우, 해당 페이지가 메모리에 없거나 불법적인 주소 공간 접근임을 의미하며, 이는 일반적으로 페이지 폴트(Page Fault) 트랩을 발생시킵니다.
7. TLB가 없다면, CPU가 논리 주소를 물리 주소로 변환하기 위해 매번 페이지 테이블에 접근해야 합니다. 페이지 테이블은 메인 메모리에 저장되어 있으므로, 하나의 데이터/명령어 접근을 위해 메모리 접근이 최소 두 번(페이지 테이블 접근 + 실제 데이터 접근) 이루어져 시스템 성능이 크게 저하됩니다.
8. 식사하는 철학자 문제의 모니터 해결책에서 test(i) 함수는 철학자 i가 젓가락을 들고 식사를 시작할 수 있는지 확인합니다. 구체적으로는 철학자 i가 'HUNGRY' 상태이고, 양 옆의 이웃((i+4)%5와 (i+1)%5)이 'EATING' 상태가 아닐 때, 철학자 i의 상태를 'EATING'으로 변경하고 대기 중인 self[i] 조건 변수를 signal하여 식사를 허용합니다.
9. 자원 할당 그래프에서 사이클이 존재하더라도, 해당 사이클에 포함된 자원 유형이 여러 인스턴스를 가질 경우 교착상태가 아닐 수 있습니다. 예를 들어, 자원 유형이 여러 개의 동일한 자원 인스턴스를 가지고 있고, 한 프로세스가 그 중 하나의 인스턴스를 보유하고 있다면, 다른 프로세스가 그 유형의 다른 가용한 인스턴스를 얻어 작업을 완료하고 자원을 반납함으로써 사이클이 해소될 수 있습니다.
10. 읽기-쓰기 락은 읽기 작업이 쓰기 작업보다 훨씬 빈번한(read-heavy) 시나리오에서 여러 읽기 스레드가 동시에 락을 획득하여 데이터를 병렬적으로 읽을 수 있으므로 뮤텍스보다 성능 이점을 가집니다. 그러나 읽기 스레드 우선 정책을 사용할 경우, 읽기 요청이 지속적으로 들어오면 쓰기 스레드는 계속해서 락을 획득하지 못하고 '기아(starvation)' 상태에 빠질 수 있습니다.

에세이 형식 문제 (답변 미포함)

1. CPU 스케줄링 알고리즘 중 SJF와 RR의 특징을 비교하고, 각각이 어떤 시스템 환경 또는 프로세스 특성에서 효율적인지 설명하시오. 또한, 두 알고리즘의 단점과 이를 보완하기 위한 방안에 대해 논하시오.

2. 생산자-소비자 문제 해결을 위해 세마포어와 뮤텍스를 함께 사용하는 방법을 자세히 설명하고, 각 동기화 프리미티브의 역할과 초기값 설정의 중요성을 강조하시오. 또한, 잘못된 동기화 구현으로 인해 발생할 수 있는 교착상태 또는 라이블락의 시나리오를 제시하시오.
3. 교착상태의 4가지 필요조건을 각각 정의하고, 각 조건이 교착상태 발생에 어떻게 기여하는지 설명하시오. 이어서, 각 조건을 무효화하여 교착상태를 예방하는 구체적인 방법들을 제시하고, 각 방법의 시스템 성능 및 효율성에 미치는 영향에 대해 논하시오.
4. 페이징 기법의 등장 배경(연속 메모리 할당의 문제점)과 페이징의 기본 개념(페이지, 프레임, 페이지 테이블)을 설명하시오. 특히, TLB(Translation Lookaside Buffer)가 페이징 시스템의 성능에 미치는 영향과 계층적 페이징이 필요한 이유에 대해 기술하시오.
5. 은행가 알고리즘의 동작 원리를 Available, Max, Allocation, Need 행렬을 사용하여 설명하고, 이 알고리즘이 교착상태를 회피하는 데 어떻게 기여하는지 논하시오. 또한, 은행가 알고리즘이 실제 운영체제에 적용되기 어려운 이유를 현실적인 제약사항과 연관하여 설명하시오.

용어집 (Glossary)

- **CPU 버스트 (CPU Burst)**: 프로세스가 CPU를 사용하여 연산을 수행하는 기간.
- **I/O 버스트 (I/O Burst)**: 프로세스가 I/O 작업을 수행하고 I/O 완료를 기다리는 기간.
- **지수 평균 (Exponential Averaging)**: 과거의 값과 현재의 실제 값을 가중 평균하여 다음 값을 예측하는 방법. CPU 버스트 길이 예측에 사용.
- **Shortest-Remaining-Time-First (SRTF)**: 최단 잔여 시간 우선 스케줄링. SJF의 선점형 버전으로, 남은 CPU 버스트 시간이 가장 짧은 프로세스를 먼저 실행.
- **알파 (α)**: 지수 평균 공식에서 현재 실제 값에 부여하는 가중치. 0에서 1 사이의 값.
- **컨텍스트 스위치 (Context Switch)**: CPU가 하나의 프로세스에서 다른 프로세스로 제어권을 전환하는 과정.
- **디스패처 (Dispatcher)**: CPU 스케줄러에 의해 선택된 프로세스에게 CPU 제어권을 넘기는 모듈.
- **호송 효과 (Convoy Effect)**: FCFS 스케줄링에서 긴 프로세스가 먼저 도착하여 CPU를 점유하면, 뒤에 있는 짧은 프로세스들이 오랫동안 기다리게 되는 현상.
- **에이징 (Aging)**: 우선순위 스케줄링에서 기아 현상을 방지하기 위해 오랫동안 대기한 프로세스의 우선순위를 점진적으로 높이는 기법.
- **경합 조건 (Race Condition)**: 여러 프로세스나 스레드가 공유 데이터에 동시에 접근하여 조작할 때, 실행 순서에 따라 결과가 달라지는 상황.
- **임계 구역 (Critical Section)**: 공유 데이터에 접근하는 코드 부분으로, 한 번에 하나의 프로세스/스레드만 접근하도록 제어해야 하는 영역.
- **상호 배제 (Mutual Exclusion)**: 임계 구역 문제 해결을 위한 첫 번째 요구사항. 한 프로세스가 임계 구역에 있을 때 다른 프로세스는 진입할 수 없어야 함.
- **진행 (Progress)**: 임계 구역이 비어있고 진입 요청이 있으면, 다음 진입 프로세스 선택이 무한히 연기되지 않아야 함.
- **유한 대기 (Bounded Waiting)**: 프로세스가 임계 구역 진입 요청 후, 진입이 허가될 때까지 다른 프로세스들의 진입 횟수에 상한이 존재해야 함.
- **Test-and-Set 명령어**: 특정 메모리 위치의 현재 값을 반환하고, 새 값을 true로 설정하는 원자적 하드웨어 명령어. 뮤텍스 구현에 사용.
- **Compare-and-Swap (CAS)**: 특정 메모리 위치의 현재 값이 예상 값과 일치하는 경우에만 새로운 값으로 원자적으로 변경하는 하드웨어 명령어. 락-없는 알고리즘에 사용.
- **뮤텍스 락 (Mutex Lock)**: 상호 배제를 위한 간단한 동기화 도구. 잠김/열림 상태를 가짐.

- **스핀락 (Spinlock)**: 뮤텍스 락의 한 구현 방식으로, 잠금을 획득할 수 없을 때 CPU를 점유한 채 반복적으로 확인하며 대기하는 바쁜 대기(busy-waiting) 방식.
- **세마포어 (Semaphore)**: 정수형 변수와 wait(P)/signal(V) 원자적 연산을 통해 자원의 가용 개수를 관리하거나 스레드 간 실행 순서를 제어하는 동기화 도구.
- **카운팅 세마포어 (Counting Semaphore)**: 여러 개의 자원 인스턴스를 관리할 수 있는 세마포어.
- **이진 세마포어 (Binary Semaphore)**: 0 또는 1의 값만 가지는 세마포어. 뮤텍스처럼 상호 배제에 사용될 수 있음.
- **모니터 (Monitor)**: 고수준 언어에서 제공하는 동기화 메커니즘. 공유 데이터와 해당 데이터를 조작하는 프로시저들을 하나의 단위로 묶어 상호 배제를 자동으로 보장.
- **조건 변수 (Condition Variable)**: 모니터 내부에서 스레드를 대기시키거나 깨울 때 사용하는 메커니즘. 특정 조건이 만족될 때까지 스레드를 블록.
- **유한 버퍼 문제 (Bounded-Buffer Problem)**: 생산자와 소비자가 고정 크기 버퍼를 공유하는 고전적인 동기화 문제.
- **읽기-쓰기 락 (Read-Write Lock)**: 여러 읽기 작업은 동시에 허용하지만, 쓰기 작업은 배타적으로 수행되는 락.
- **식사하는 철학자 문제 (Dining-Philosophers Problem)**: 교착상태 발생 가능성을 보여주는 고전적인 동기화 문제 예시.
- **점유와 대기 (Hold and Wait)**: 교착상태 필요조건 중 하나. 한 자원을 점유한 채 다른 자원을 기다리는 상황.
- **비선점 (No Preemption)**: 교착상태 필요조건 중 하나. 자원을 강제로 빼앗을 수 없고, 오직 자발적으로만 해제될 수 있는 상황.
- **순환 대기 (Circular Wait)**: 교착상태 필요조건 중 하나. 프로세스들이 순환 형태로 서로가 가진 자원을 대기하는 상황.
- **자원 할당 그래프 (Resource-Allocation Graph)**: 프로세스와 자원 간의 요청 및 할당 관계를 나타내는 방향 그래프.
- **안전 상태 (Safe State)**: 시스템이 교착상태에 빠지지 않고 모든 프로세스가 작업을 완료할 수 있는 순서가 존재하는 상태.
- **안전 순서 (Safe Sequence)**: 시스템이 안전 상태에 있음을 보장하는 프로세스의 실행 순서.
- **은행가 알고리즘 (Banker's Algorithm)**: 교착상태 회피를 위한 알고리즘. 자원 할당 전 안전 상태 여부를 검사하여 할당 여부 결정.
- **Available 행렬 (Available Matrix)**: 현재 사용 가능한 각 자원 유형의 인스턴스 개수를 나타내는 벡터.
- **Max 행렬 (Max Matrix)**: 각 프로세스가 최대로 요구할 수 있는 각 자원 유형의 인스턴스 개수를 나타내는 행렬.
- **Allocation 행렬 (Allocation Matrix)**: 각 프로세스에 현재 할당된 각 자원 유형의 인스턴스 개수를 나타내는 행렬.
- **Need 행렬 (Need Matrix)**: 각 프로세스가 작업을 완료하기 위해 추가적으로 필요한 각 자원 유형의 인스턴스 개수를 나타내는 행렬. (Need = Max - Allocation)
- **라이블락 (Livelock)**: 스레드가 바쁘게 상태를 변경하며 계속 실행되지만, 실제 유용한 작업을 진행하지 못하고 무한히 반복되는 상황.
- **논리 주소 (Logical Address)**: CPU가 생성하는 주소. 가상 주소라고도 함.
- **물리 주소 (Physical Address)**: 메모리 장치에 실제로 존재하는 주소.
- **MMU (Memory Management Unit)**: CPU가 생성하는 논리 주소를 물리 주소로 변환하는 하드웨어 장치.

- **페이지 (Page)**: 논리 메모리(프로세스의 주소 공간)를 고정된 크기로 나눈 블록.
- **프레임 (Frame)**: 물리 메모리를 페이지와 동일한 크기로 나눈 블록.
- **페이지 테이블 (Page Table)**: 논리 주소의 페이지 번호를 해당 페이지가 적재된 물리 메모리의 프레임 번호로 매핑하는 테이블.
- **페이지 오프셋 (Page Offset)**: 페이지 또는 프레임 내에서의 상대적인 위치를 나타내는 주소 부분.
- **TLB (Translation Lookaside Buffer)**: 페이지 테이블 접근 속도를 높이기 위한 고속 하드웨어 캐시. 최근 사용된 페이지-프레임 매핑 정보를 저장.
- **히트율 (Hit Ratio)**: TLB에서 원하는 페이지 변환 정보를 찾을 확률.
- **내부 단편화 (Internal Fragmentation)**: 할당된 메모리 블록이 프로세스가 실제로 사용하는 공간보다 커서, 블록 내부에 사용되지 않고 낭비되는 공간.
- **외부 단편화 (External Fragmentation)**: 총 메모리 공간은 충분하지만, 연속된 가용 공간이 없어 새로운 프로세스에 할당할 수 없는 현상.
- **계층적 페이징 (Hierarchical Paging)**: 페이지 테이블의 크기가 너무 커질 때, 페이지 테이블 자체를 여러 단계의 페이지 테이블로 구성하여 관리하는 방식.
- **역 페이지 테이블 (Inverted Page Table)**: 물리 메모리 프레임당 하나의 엔트리를 가지며, 해당 프레임에 어떤 프로세스의 어떤 페이지가 적재되어 있는지를 기록하는 페이지 테이블. 시스템 전체에 하나만 존재.
- **스와핑 (Swapping)**: 메모리 공간이 부족할 때, 현재 사용되지 않는 프로세스(또는 페이지)를 주 메모리에서 보조 저장소(하드 디스크)로 이동시키고, 필요할 때 다시 주 메모리로 불러오는 기법.

I. CPU 스케줄링 및 버스트 예측 (시기 미상)

- **CPU-I/O 버스트 사이클 개념 정립:** 프로세스 실행이 CPU 실행과 I/O 대기 구간이 번갈아 나타나는 사이클로 구성되며, 대부분의 프로세스가 짧은 CPU 버스트가 많고 긴 CPU 버스트가 적은 패턴을 보인다는 모델이 제시됨.
- **CPU 스케줄러 기능 정의:** 준비 큐에서 프로세스를 선택하여 CPU 코어를 할당하는 역할 정의.
- **디스패처 역할 정의:** 스케줄러가 선택한 프로세스에게 실제로 CPU 제어권을 전달하는 모듈로, 컨텍스트 스위칭, 모드 전환, 프로그램 카운터 점프 등을 수행하는 기능 정의.
- **스케줄링 성능 지표 정의:** CPU 활용률, 처리량, 반환 시간, 대기 시간, 응답 시간 등 핵심 성능 지표 제시.
- **다양한 스케줄링 알고리즘 등장:선입선출 (FCFS):** 가장 단순한 비선점형 알고리즘으로, 호송 효과 (convoy effect) 문제 지적.
- **최단 작업 우선 (SJF):** 평균 대기 시간이 최적인 알고리즘으로, 선점형 버전은 SRTF(Shortest Remaining Time First)로 불림.
- **우선순위 스케줄링:** 기아 현상 방지를 위해 에이징 기법 사용 가능 제시.
- **라운드 로빈 (RR):** 시분할 시스템의 핵심 알고리즘으로, 공정한 CPU 시간 분배를 보장하나 컨텍스트 스위치 오버헤드 발생.
- **다단계 큐 스케줄링:** 프로세스 유형별로 별도 큐를 사용하며 각 큐마다 다른 알고리즘 적용 가능.
- **다단계 피드백 큐:** 프로세스 큐 간 이동이 가능한 고급 알고리즘으로, 동적 우선순위 조정 가능.
- **CPU 버스트 예측 기법 (지수 평균) 도입:**다음 CPU 버스트 길이를 예측하는 공식 $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ 제시.
- α 값을 통해 최근 이력의 가중치를 조절하는 방식 설명:
- $\alpha = 0$: 최근 이력을 반영하지 않고 이전 예측 값만 사용 ($\tau_{n+1} = \tau_n$).
- $\alpha = 1$: 오직 실제 마지막 CPU 버스트만 반영 ($\tau_{n+1} = \alpha t_n$).
- α가 일반적으로 1/2로 설정되는 경우가 흔하다고 언급됨.
- **최단-남은-시간-우선 (SRTF) 개념 정립:** 최단 작업 우선(SJF)의 선점형 버전으로 정의.

II. 동시성 및 동기화 도구의 발전 (시기 미상)

- **동시성과 데이터 일관성 문제 인식:** 여러 프로세스가 공유 데이터에 동시 접근할 때 데이터 불일치가 발생할 수 있다는 점 인식.
- **경합 조건(Race Condition) 정의:** 실행 순서에 따라 결과가 달라지는 상황으로, counter++ 연산 예시 제시.
- **임계 구역(Critical Section) 문제 정의:** 여러 프로세스가 공유 변수를 변경하는 코드 세그먼트이며, 한 프로세스 실행 중에는 다른 프로세스 진입 불가를 요구.
- **임계 구역 해결책 요구사항 제시:** 상호 배제, 진행, 유한 대기 조건 정의.
- **피터슨의 해결책 제안:** 두 프로세스를 위한 소프트웨어 기반 해결책으로 turn 변수와 flag 배열 사용 제시, 현대 아키텍처에서의 한계점 언급.
- **하드웨어 지원 동기화 명령어 등장:Test-and-Set 명령어:** 원자적으로 값을 반환하고 새 값을 true로 설정하여 뮤텍스 락 구현 가능성 제시.
- **Compare-and-Swap (CAS) 명령어:** 예상 값과 일치하는 경우에만 원자적으로 값을 변경하는 하드웨어 연산으로, 락 없는(lock-free) 자료구조 구현의 핵심 역할.
- **뮤텍스 락 개발:** 가장 간단한 동기화 도구로 acquire()와 release() 함수 제공, 바쁜 대기(busy-waiting) 문제로 인해 스핀락으로도 불림.
- **세마포어 개념 도입:** 정수 변수 S와 원자적 연산 wait() (P), signal() (V)로 구성, 카운팅 세마포어와 이진 세마포어로 구분.
- 바쁜 대기 없는 구현을 위해 대기 큐 및 block, wakeup 연산 사용.

- 임계 구역 문제 및 실행 순서 동기화에 활용.
- **모니터 도입:** 프로세스 동기화를 위한 고수준 추상화 메커니즘으로, 내부 변수는 프로시저 내에서만 접근 가능하며 한 번에 하나의 프로세스만 활성화.
- x.wait()와 x.signal() 연산을 통해 프로세스 중단 및 재시작 제어.
- **고전적인 동기화 문제 제시 및 해결 노력:유한 버퍼 문제(생산자-소비자 문제):** 고정 크기 버퍼 공유 문제로, mutex, full, empty 세마포어를 함께 사용하여 해결 방안 제시.
- **읽기-쓰기 문제:** 다수의 읽기/쓰기 프로세스가 데이터베이스 공유 시 읽기는 동시, 쓰기는 독점 요구. 읽기-쓰기 락 개념 등장.
- **식사하는 철학자 문제:** 교착상태 방지를 위한 다양한 해결책 논의. 모니터를 사용한 해결책 제시.

III. 교착상태(Deadlock) 및 관리 전략 (시기 미상)

- **교착상태 정의:** 여러 프로세스(또는 스레드)가 서로가 점유하고 있는 자원을 무한정 기다리는 상황.
- **교착상태의 4가지 필요조건(Coffman Conditions) 정립:상호 배제 (Mutual Exclusion):** 한 번에 하나의 프로세스만 자원 사용 가능.
- **점유와 대기 (Hold and Wait):** 자원을 보유한 채로 다른 자원을 대기.
- **비선점 (No Preemption):** 자원을 강제로 빼앗을 수 없음.
- **순환 대기 (Circular Wait):** 프로세스들이 원형으로 서로의 자원을 대기.
- **자원 할당 그래프(Resource-Allocation Graph) 도입:** 프로세스와 자원 간의 관계를 나타내는 방향 그래프로, 사이클 유무로 교착상태 가능성 판단. (사이클이 없으면 교착상태 없음, 사이클이 있으면 교착상태 가능성 있음.)
- **교착상태 처리 방법론 제시:예방 (Prevention):** 네 가지 필요조건 중 하나를 무효화하여 교착상태 원천 차단. (예: 점유와 대기 무효화, 비선점 무효화, 순환 대기 무효화)
- **회피 (Avoidance):** 시스템이 안전 상태를 유지하도록 동적으로 자원 할당을 제어.
- **뱅커스 알고리즘(Banker's Algorithm):** 각 프로세스의 최대 자원 요구량을 미리 알고 있다고 가정하고, 자원 할당 요청 시 안전성 검사. (Available, Max, Allocation, Need 행렬 사용)
- **탐지 및 복구 (Detection & Recovery):** 교착상태 발생을 허용하되 주기적으로 탐지하여 복구.
- **탐지:** 대기 그래프의 사이클 검사 또는 다중 인스턴스용 탐지 알고리즘 사용.
- **복구:** 프로세스 종료 또는 자원 선점을 통해 교착상태 해결.
- **실제 시스템에서의 적용:** 대부분의 현대 운영체제(Windows, Linux, macOS)는 성능상의 이유로 교착상태를 무시하는 접근법 사용.
- **라이블락(Livelock) 개념 정의:** 살아있으면서 더 이상 진행이 안 되는 상태로, 특정 작업이 반복되지만 앞으로 나아가지 못하는 상황.

IV. 주 메모리 관리 및 가상 메모리 (시기 미상)

- **메모리 계층 구조 이해:** CPU 레지스터, 캐시, 주 메모리, 보조 저장소 순으로 구성되며 속도와 용량이 반비례.
- **메모리 보호 메커니즘 도입:** 베이스 레지스터와 리밋 레지스터를 사용하여 프로세스의 시작 주소와 크기를 저장하고 MMU가 메모리 접근 검사.
- **주소 바인딩 시점 구분:** 컴파일 시간, 로드 시간, 실행 시간 바인딩 개념.
- **논리 주소와 물리 주소 개념 정립:논리 주소:** 프로그램이 생성하는 가상 주소.
- **물리 주소:** 실제 메모리 하드웨어의 주소.
- **MMU(Memory Management Unit):** 논리 주소를 물리 주소로 변환하는 하드웨어.
- **동적 로딩 및 연결 기법 도입:동적 로딩:** 프로그램 실행 중 필요한 루틴만 메모리에 로드하여 효율성 증대.

- **동적 연결:** 실행 시점에 라이브러리와 연결하여 실행 파일 크기 줄이고 공유 라이브러리 사용 가능.
- **연속 메모리 할당 방식 논의:**메모리 분할: 운영체제 영역과 사용자 프로세스 영역 분할, 가변 분할 방식에서 홀(빈 공간)과 할당된 영역의 동적 변화.
- 동적 저장소 할당 알고리즘: First Fit, Best Fit, Worst Fit 제시.
- 단편화 문제 인식:
- **외부 단편화:** 총 여유 공간은 충분하나 연속되지 않아 할당 불가.
- **내부 단편화:** 할당된 공간이 요청보다 커서 일부 낭비.
- 압축: 외부 단편화 해결을 위한 메모리 재배치(높은 비용).
- **페이징(Paging) 개념 도입 및 발전:페이징 개념:** 물리 메모리를 고정 크기 프레임으로, 논리 메모리를 같은 크기의 페이지로 나누어 외부 단편화 해결. 페이지 테이블을 통해 논리 주소를 물리 주소로 변환.
- **주소 변환:** 논리 주소는 페이지 번호(p)와 페이지 오프셋(d)으로 구성되며, 페이지 테이블에서 해당 프레임 번호를 찾아 물리 주소 계산.
- **TLB(Translation Lookaside Buffer) 도입:** 페이지 테이블 접근 횟수를 줄이기 위한 고속 캐시로, 최근 사용된 페이지 변환 정보 저장. TLB 히트율이 성능에 큰 영향.
- **메모리 보호:** 페이지 테이블 항목에 유효-무효 비트, 보호 비트 포함하여 페이지별 접근 권한 제어. 공유 페이지를 통한 메모리 효율성 증대.
- **계층적 페이지 테이블:** 페이지 테이블 크기 문제 해결을 위해 다단계 페이지 테이블 사용. 64비트 시스템에서 3단계 이상 페이징 필요성 언급.
- **해시 페이지 테이블:** 희소 주소 공간에서 효과적으로, 가상 페이지 번호를 해시하여 물리 프레임 찾음.
- **역 페이지 테이블:** 물리 메모리 기준으로 구성되어 시스템당 하나만 존재, 메모리 절약.
- **스와핑(Swapping) 개념 도입:** 메모리 부족 시 프로세스를 일시적으로 보조 저장소로 이동시키는 기법.

V. 캐시 일관성 (시기 미상)

- **멀티코어 환경에서 캐시 일관성 문제 발생 이유 설명:** 각 코어의 로컬 캐시 사용으로 인한 데이터 불일치 (stale data) 문제 발생.
- **MESI 캐시 일관성 프로토콜 도입:** 캐시 라인의 4가지 상태 정의:
- **Modified (M):** 캐시에만 존재하며 메인 메모리와 내용이 다름 (dirty).
- **Exclusive (E):** 캐시에만 존재하며 메인 메모리와 내용이 동일 (clean). 쓰기 시 M 상태로 변경.
- **Shared (S):** 여러 캐시에 존재하며 메인 메모리와 내용이 동일 (clean). 쓰기 시 다른 캐시 복사본 무효화 및 M 상태로 변경.
- **Invalid (I):** 유효하지 않거나 사용되지 않음.
- **'Shared' 상태 데이터에 대한 쓰기 연산 시 일관성 유지 과정 설명:** BusUpgr 또는 RFO 메시지를 통한 스누핑 및 Invalid 상태로 변경 과정 언급.
- **메모리 배리어(Memory Barrier) 필요성 언급:** CPU 및 컴파일러의 성능 최적화로 인한 명령어 재배치 문제 해결 및 데이터 일관성 보장 역할.

VI. 락 없는/대기 없는 알고리즘 (시기 미상)

- **비차단(Non-Blocking) 동기화 방식 개념 도입:락 없는(Lock-Free) 알고리즘:** 시스템 전체적으로 항상 진행이 이루어짐을 보장하나 개별 스레드는 기아 상태 가능.
- **대기 없는(Wait-Free) 알고리즘:** 모든 스레드가 각자의 연산을 유한한 단계 내에 완료함을 보장하여 기아 상태 발생 없음 (가장 강력한 진행 보장).
- **전통적인 락 기반 동기화 방식 대비 장점 제시:** 교착상태/라이블락 회피, 성능/확장성 향상, 우선순위 역전 문제 해결, 인터럽트/시그널 핸들러 안전성.

- **구현 시 주요 도전 과제 제시:** 복잡성, 원자적 연산 의존성, ABA 문제, 메모리 관리 어려움, 성능 보장 어려움.

주요 인물 (출처에 명시된 인물이 없음. 개념 및 알고리즘 위주로 설명되어 있음.)

- **시스템 운영자 (Operator):** 교착상태 발생 시 개입하여 특정 스레드나 프로세스를 종료시키거나 자원을 선점하여 시스템을 복구하는 주체.
- **생산자 (Producer):** 유한 버퍼 문제에서 버퍼에 아이템을 추가하는 스레드/프로세스.
- **소비자 (Consumer):** 유한 버퍼 문제에서 버퍼에서 아이템을 추출하는 스레드/프로세스.
- **철학자 (Philosopher):** 식사하는 철학자 문제에서 젓가락(자원)을 공유하며 식사하려는 주체. 교착상태 발생의 전형적인 예시.
- **뱅커(Banker):** 뱅커스 알고리즘의 비유적 표현에서, 고객(프로세스)에게 자원(대출)을 할당할 때 시스템(은행)의 안전성을 보장하는 역할을 하는 주체.
- **스레드 (Thread) / 프로세스 (Process):** CPU 시간, 메모리, 자원 등을 할당받아 실행되는 프로그램의 단위. 스케줄링, 동기화, 교착상태, 메모리 관리 등 모든 운영체제 개념의 중심 대상.

1. 운영체제가 교착 상태(Deadlock)를 처리하는 주요 방법에는 어떤 것들이 있으며, 각 방법의 특징은 무엇인가요?

운영체제가 교착 상태를 처리하는 주요 방법은 크게 세 가지로 나눌 수 있습니다:

1. **교착 상태 예방(Deadlock Prevention):** 교착 상태가 발생하기 위한 네 가지 필수 조건(상호 배제, 점유와 대기, 비선점, 순환 대기) 중 하나 이상을 무효화하여 교착 상태를 원천적으로 차단하는 방식입니다. 예를 들어, 프로세스가 실행을 시작하기 전에 필요한 모든 자원을 요청하거나, 자원을 보유하지 않은 상태에서만 새로운 자원을 요청하도록 강제하여 '점유와 대기' 조건을 위반할 수 있습니다. 각 자원 유형에 총 순서를 부여하고 오름차순으로만 요청하도록 하여 '순환 대기'를 방지하는 것이 가장 일반적인 예방 방법입니다. 이 방법은 안전하지만, 자원 활용률이 낮아지고 성능 저하를 초래할 수 있습니다.
2. **교착 상태 회피(Deadlock Avoidance):** 시스템이 자원 할당 상태에 대한 사전 정보(예: 각 프로세스의 최대 자원 요구량)를 사용하여, 자원 할당 요청 시 시스템이 '안전 상태(Safe State)'를 유지하는지 동적으로 검사하는 방식입니다. 안전 상태는 모든 프로세스가 작업을 완료할 수 있는 순서가 존재하는 상태를 의미합니다. '은행가 알고리즘(Banker's Algorithm)'이 대표적인 회피 기법으로, 자원 유형당 여러 인스턴스가 있을 때 사용됩니다. 이 방법은 예방보다는 유연하지만, 프로세스의 최대 자원 요구량을 미리 알아야 한다는 제약이 있습니다.
3. **교착 상태 탐지 및 복구(Deadlock Detection and Recovery):** 시스템이 교착 상태에 진입하는 것을 허용하되, 주기적으로 교착 상태 발생 여부를 탐지하고, 발생 시 복구하는 방식입니다. 탐지 알고리즘은 '대기 그래프(wait-for graph)'를 사용하여 사이클 유무를 확인합니다. 복구 방법으로는 교착 상태에 연루된 프로세스를 종료시키거나(모두 중단 또는 하나씩 중단), 자원을 선점(preemption)하여 희생자를 선택하고 롤백시키는 방법이 있습니다. 대부분의 현대 운영체제(Windows, Linux, macOS)는 성능 오버헤드 때문에 교착 상태를 무시하고 이 탐지 및 복구 방법을 사용합니다.

2. 프로세스 동기화에서 "경합 조건(Race Condition)"이란 무엇이며, 이를 해결하기 위한 "임계 구역(Critical Section)" 문제와 그 요구사항은 무엇인가요?

경합 조건(Race Condition)은 여러 프로세스나 스레드가 동시에 공유 데이터에 접근하여 값을 변경할 때, 접근 순서에 따라 실행 결과가 달라지는 상황을 말합니다. 예를 들어, count++ 연산이 여러 기계어 명령어로 나뉘어 실행될 때, 중간에 다른 스레드가 count 값을 변경하면 예상치 못한 최종 결과가 나올 수 있습니다.

이를 해결하기 위한 **임계 구역(Critical Section) 문제**는 여러 프로세스가 공유 자원에 접근하는 코드 세그먼트인 임계 구역을 가질 때, 한 프로세스가 임계 구역에서 실행 중일 때 다른 어떤 프로세스도 자신의 임계 구역에 진입할 수 없도록 보장하는 프로토콜을 설계하는 것입니다. 각 프로세스는 임계 구역에 들어가기 위해 진입 구역에서 허가를 요청하고, 임계 구역을 마친 후에는 퇴출 구역을 거쳐 나머지 구역으로 이동합니다.

임계 구역 해결책을 위한 세 가지 주요 요구사항은 다음과 같습니다:

1. **상호 배제(Mutual Exclusion):** 프로세스 Pi가 임계 구역에서 실행 중이면, 다른 어떤 프로세스도 자신의 임계 구역에 들어갈 수 없어야 합니다.
2. **진행(Progress):** 임계 구역에서 실행 중인 프로세스가 없고, 임계 구역에 들어가고자 하는 프로세스가 있다면, 다음에 임계 구역에 들어갈 프로세스를 선택하는 과정이 무기한 연기되어서는 안 됩니다.
3. **유한 대기(Bounded Waiting):** 프로세스가 임계 구역 진입을 요청한 후, 다른 프로세스가 자신의 임계 구역에 들어갈 수 있는 횟수에 상한이 존재해야 합니다. 즉, 어떤 프로세스도 무한히 기다리지 않아야 합니다.

3. 뮤텍스(Mutex)와 세마포어(Semaphore)는 동기화 도구로서 어떻게 다르며, 각각의 주요 특징은 무엇인가요?

뮤텍스와 세마포어는 모두 공유 자원에 대한 동시 접근을 제어하는 동기화 도구이지만, 다음과 같은 주요 차이점을 가집니다:

특징뮤텍스 (Mutex)세마포어 (Semaphore)**개념**상호 배제(Mutual Exclusion)를 위한 잠금 메커니즘신호를 기반으로 한 동기화 도구 (카운터 값)**소유권**락을 획득한 스레드만이 락을 해제할 수 있는 명시적인 소유권 개념이 있습니다.명시적인 소유권 개념이 없습니다. 어떤 스레드든 signal(V) 연산을 통해 값을 증가시키고 대기 중인 스레드를 깨울 수 있습니다.**자원 관리**잠김(locked) 또는 열림(unlocked) 두 가지 상태만 가지므로, 개념적으로 하나의 자원만을 관리합니다 (이진 세마포어와 유사).내부적으로 정수형 카운터 값을 가집니다. 이는 0 이상의 가용 자원의 개수를 나타낼 수 있습니다. 이진 세마포어(Binary Semaphore)는 0 또는 1의 값만 가집니다.**용도**임계 구역 보호와 같이 하나의 스레드만 특정 코드 블록에 접근하도록 할 때 주로 사용됩니다.여러 개의 공유 자원에 대한 접근을 제어하거나, 스레드 간의 신호 전달(생산자-소비자 문제 등)에 사용됩니다.**조건 변수**조건 변수(Condition Variable)와 함께 사용되어 복잡한 스레드 간 동기화(조건에 따른 대기 및 깨움)를 구현합니다. wait()는 항상 연관된 뮤텍스와 함께 사용되어야 합니다.그 자체로 동기화 프리미티브로 사용될 수 있으며, wait(S) 연산이 항상 외부의 별도 뮤텍스와 함께 사용되어야 하는 것은 아닙니다.간단히 말해, 뮤텍스는 '하나의 문'을 잠그고 여는 열쇠와 같아서 한 번에 한 명만 들어갈 수 있도록 하는 반면, 세마포어는 '여러 개의 자원'이 있을 때 몇 명이 들어갈 수 있는지 세어주는 카운터와 같습니다.

4. 컴퓨터 시스템에서 가상 메모리(Virtual Memory)와 페이징(Paging)의 개념은 무엇이며, 왜 필요한가요?

가상 메모리(Virtual Memory)는 실제 물리 메모리보다 훨씬 큰 논리적 주소 공간을 프로세스에 제공하는 메모리 관리 기법입니다. 이를 통해 개발자는 실제 물리 메모리 크기에 구애받지 않고 큰 프로그램을 작성할 수 있으며, 여러 프로세스가 동시에 실행되는 멀티프로그래밍 환경에서 각 프로세스가 독립적인 주소 공간을 가지는 것처럼 보이게 합니다.

페이징(Paging)은 가상 메모리를 구현하는 가장 일반적인 방법 중 하나입니다. 이는 프로세스의 논리 주소 공간을 고정된 크기의 **페이지(Page)**로 나누고, 물리 메모리 공간을 동일한 크기의 **프레임(Frame)**으로 나눕니다. 프로세스의 페이지는 물리 메모리의 어떤 빈 프레임이든 비연속적으로 할당될 수 있습니다. 가상 주소는 페이지 번호와 페이지 오프셋으로 구성되며, **메모리 관리 장치(MMU)**와 **페이지 테이블(Page Table)**을 통해 물리 주소로 변환됩니다. 페이지 테이블은 각 가상 페이지가 물리 메모리의 어떤 프레임에 매핑되어 있는지를 기록합니다.

페이징이 필요한 주된 이유는 다음과 같습니다:

- **외부 단편화(External Fragmentation) 해결:** 연속 메모리 할당 방식에서 발생하는 외부 단편화(총 메모리 공간은 충분하지만 연속적인 공간이 없어 할당할 수 없는 문제)를 근본적으로 해결합니다. 페이지 단위로 메모리를 할당함으로써 프로세스가 비연속적인 물리 메모리 공간을 사용할 수 있습니다.
- **메모리 효율성 증대:** 프로그램의 전체 코드가 메모리에 로드될 필요 없이, 필요한 부분만 페이지 단위로 로드하여 메모리 공간을 효율적으로 활용할 수 있습니다.
- **프로세스 간 보호 및 격리:** 각 프로세스는 자신만의 페이지 테이블을 가지므로, 다른 프로세스의 메모리 영역을 침범하는 것을 방지하여 시스템의 안정성을 높입니다.
- **공유 메모리 구현 용이:** 여러 프로세스가 동일한 코드(예: 텍스트 편집기)나 데이터를 공유할 때, 해당 페이지를 물리 메모리의 동일한 프레임에 매핑하여 효율적으로 메모리를 공유할 수 있습니다.

5. 페이지 테이블(Page Table)의 크기 문제는 어떻게 발생하며, 이를 해결하기 위한 일반적인 접근 방식은 무엇인가요?

페이지 테이블은 가상 페이지 번호를 물리 프레임 번호로 매핑하는 데 사용되는 데이터 구조입니다. 가상 주소 공간이 커지거나 페이지 크기가 작아질수록 페이지 테이블의 크기가 기하급수적으로 커지는 문제가 발생합니다. 예를 들어, 32비트 주소 공간에서 페이지 크기가 4KB(2^12 바이트)라면, 페이지 번호는 20비트(32-12)가 됩니다. 각 페이지 테이블 항목이 4바이트라고 가정하면, 하나의 프로세스 페이지 테이블만으로도 2^20 * 4 바이트 = 4MB의 메모리가 필요하게 됩니다. 64비트 시스템에서는 이 문제가 훨씬 심각해져, 하나의 페이지 테이블만으로도 테라바이트 단위의 메모리가 필요할 수 있습니다.

이를 해결하기 위한 일반적인 접근 방식은 다음과 같습니다:

1. **다단계 페이지 테이블(Multilevel Page Table):** 페이지 테이블 자체를 페이징하는 방식입니다. 예를 들어 2단계 페이징에서 가상 주소의 페이지 번호는 다시 '외부 페이지 테이블의 인덱스(p1)'와 '내부 페이지 테이블의 페이지 내 변위(p2)'로 나뉩니다. 이렇게 하면 전체 페이지 테이블을 한 번에 메모리에 로드할 필요 없이, 필요한 부분만 로드하여 메모리 사용량을 줄일 수 있습니다. 하지만 이는 물리 메모리 접근 횟수를 증가시켜 성능 저하를 초래할 수 있습니다 (예: 3단계 페이징 시 하나의 데이터 접근을 위해 4번의 메모리 접근이 필요할 수 있음).
2. **해시 페이지 테이블(Hashed Page Tables):** 32비트보다 큰 주소 공간에서 흔히 사용됩니다. 가상 페이지 번호를 해시 함수를 통해 페이지 테이블의 특정 위치로 매핑하고, 충돌이 발생할 경우 체인 방식으로 연결합니다. 각 항목은 가상 페이지 번호, 매핑된 물리 프레임 값, 그리고 다음 요소에 대한 포인터를 포함합니다. 희소 주소 공간(메모리 참조가 비연속적인 경우)에 특히 유용합니다.
3. **역 페이지 테이블(Inverted Page Table):** 각 프로세스가 페이지 테이블을 가지는 대신, 시스템의 모든 물리 페이지를 추적하는 단일 페이지 테이블을 유지합니다. 즉, 물리 메모리의 실제 프레임 하나당 하나의 항목이 있습니다. 이 항목은 해당 물리 메모리 위치에 저장된 페이지의 가상 주소와 소유 프로세스 정보를 포함합니다. 페이지 테이블을 저장하는 데 필요한 메모리는 감소하지만, 페이지 참조 시 테이블을 검색하는 시간이 증가할 수 있습니다. 검색 시간을 줄이기 위해 해시 테이블이나 TLB(Translation Lookaside Buffer)를 함께 사용합니다.

6. TLB(Translation Lookaside Buffer)는 무엇이며, 페이징 시스템의 성능에 어떻게 기여하나요?

TLB(Translation Lookaside Buffer)는 가상 주소를 물리 주소로 변환하는 페이지 테이블의 접근 속도를 높이기 위한 CPU 내의 고속 캐시 메모리입니다. 페이지 테이블은 일반적으로 주 메모리에 저장되어 있어, 가상 주소 하나를 물리 주소로 변환하기 위해 페이지 테이블 접근(주 메모리 접근)과 실제 데이터/명령어 접근(주 메모리 접근)으로 총 두 번의 주 메모리 접근이 필요하여 시스템 성능 저하를 초래할 수 있습니다.

TLB는 이러한 성능 저하를 완화합니다. CPU가 가상 주소에 접근할 때 가장 먼저 TLB를 확인합니다.

- **TLB 히트(Hit):** 원하는 페이지 변환 정보(가상 페이지 번호와 해당 물리 프레임 번호)가 TLB에 존재하면, MMU는 페이지 테이블에 접근할 필요 없이 TLB에서 직접 물리 주소를 얻어 빠르게 데이터/명령어에 접근할 수 있습니다. 이는 메모리 접근 시간을 크게 단축시킵니다.
- **TLB 미스(Miss):** 원하는 정보가 TLB에 없으면, MMU는 주 메모리에 있는 페이지 테이블에 접근하여 해당 정보를 찾아옵니다. 이 정보는 TLB에 저장되어 다음에 같은 주소를 접근할 때 활용될 수 있도록 합니다.

TLB의 **히트율(Hit Rate)**은 시스템 성능에 매우 큰 영향을 미칩니다. 히트율이 높을수록 페이지 테이블 접근을 위한 주 메모리 접근 횟수가 줄어들어 전체 시스템의 메모리 접근 속도가 향상됩니다. TLB는 자주 사용되는 페이지 매핑 정보를 캐시함으로써 페이징 시스템의 효율성을 극대화합니다.

7. 식사하는 철학자 문제(Dining-Philosophers Problem)는 무엇이며, 이 문제가 발생하는 이유와 몇 가지 해결책은 무엇인가요?

식사하는 철학자 문제(Dining-Philosophers Problem)는 동시성 프로그래밍에서 교착 상태와 기아 문제를 설명하기 위해 고안된 고전적인 문제입니다. N명의 철학자가 둥근 테이블에 앉아 생각하고 먹는 것을 반복하며, 각 철학자 사이에는 젓가락이 하나씩 놓여 있습니다. 철학자가 식사를 하려면 양쪽에 있는 젓가락 두 개를 모두 집어야 합니다. 젓가락은 공유 자원에 해당합니다.

문제가 발생하는 이유: 가장 큰 문제는 **교착 상태(Deadlock)**입니다. 다섯 명의 철학자가 모두 동시에 배가 고파져 각자 자신의 왼쪽 젓가락을 동시에 집어들었다고 가정해 봅시다. 이 경우 모든 철학자는 왼쪽 젓가락을 점유한 채로, 자신의 오른쪽에 있는 젓가락(이미 옆 철학자의 왼손에 잡혀 있는)이 놓이기를 무한히 기다리게 됩니다. 이로 인해 모든 철학자가 아무것도 하지 못하고 굶어 죽는 교착 상태에 빠지게 됩니다. 이는 '순환 대기(Circular Wait)' 조건과 '점유와 대기(Hold and Wait)' 조건이 동시에 충족될 때 발생합니다.

해결책: 교착 상태를 피하기 위한 몇 가지 접근 방식이 있습니다:

1. **철학자 수 제한:** 테이블에 동시에 앉을 수 있는 철학자의 수를 최대 N-1명으로 제한합니다. (예: 5명이라면 4명만 앉게 함) 이렇게 하면 항상 최소 한 명의 철학자가 두 젓가락을 모두 집을 수 있어 교착 상태가 방지됩니다.
2. **원자적 젓가락 집기:** 철학자가 두 젓가락이 모두 사용 가능한 경우에만 젓가락을 집을 수 있도록 허용합니다. 이를 위해서는 젓가락을 집는 행위 자체가 임계 구역 안에서 이루어져야 하며, 젓가락을 집기 전에 두 젓가락의 상태를 모두 확인해야 합니다.
3. **비대칭 해결책:** 홀수 번호 철학자는 먼저 왼쪽 젓가락을 집고 오른쪽 젓가락을 집는 반면, 짝수 번호 철학자는 먼저 오른쪽 젓가락을 집고 왼쪽 젓가락을 집도록 합니다. 이렇게 하면 젓가락 획득 순서에 차이를 두어 순환 대기 조건을 깨뜨립니다.
4. **모니터(Monitor) 사용:** 철학자의 상태(THINKING, HUNGRY, EATING)를 관리하고, 이웃이 식사 중이 아닌 경우에만 철학자가 식사를 시작할 수 있도록 하는 고수준 동기화 메커니즘인 모니터를 사용할 수 있습니다. 모니터 내부에서 조건 변수를 사용하여 철학자들이 젓가락을 얻을 수 있을 때까지 대기하도록 합니다.

8. 현대 멀티코어 시스템에서 명령어 재배열(Instruction Reordering)이 동기화 문제에 어떤 영향을 미치며, 이를 완화하기 위한 기술적 접근 방식은 무엇인가요?

현대 CPU와 컴파일러는 성능 최적화를 위해 명령어 실행 순서를 재배열(Instruction Reordering)하거나, 쓰기 버퍼(Write Buffer) 및 무효화 큐(Invalidate Queue)와 같은 하드웨어 최적화 기법을 사용합니다. 이러한 최적화는 단일 스레드 프로그램에서는 문제가 되지 않지만, **멀티코어 환경에서 공유 데이터에 접근하는 경우 심각한 동기화 문제를 야기**할 수 있습니다.

예를 들어, 스레드 2가 x = 100; flag = true;를 실행하고 스레드 1이 while (!flag); print x;를 실행한다고 가정해 봅시다. 예상되는 출력은 100입니다. 그러나 명령어 재배열로 인해 flag = true;가 x = 100;보다 먼저 실행될 수 있습니다. 만약 이 상황에서 스레드 1이 flag를 확인하고 임계 구역에 진입하여 x를 읽을 때 x = 100;이 아직 실행되지 않았다면, 스레드 1은 0을 출력할 수 있습니다. 피터슨 해결책과 같은 고전적인 소프트웨어

기반 동기화 알고리즘도 명령어 재배열의 영향으로 인해 두 프로세스가 동시에 임계 구역에 들어가는 등 올바르게 작동하지 않을 수 있습니다.

이를 완화하고 데이터 일관성을 보장하기 위한 기술적 접근 방식은 다음과 같습니다:

1. **메모리 배리어(Memory Barrier) 또는 펜스(Fence):** 컴파일러나 CPU에게 특정 지점에서 이전의 모든 메모리 연산들이 이후의 연산들보다 먼저 완료되거나 다른 코어에 관찰되도록 강제하는 명령어입니다. 이는 명령어 재배열을 막고 쓰기 버퍼의 내용을 강제로 플러시하거나 무효화 큐의 요청을 처리하여 최신 메모리 상태를 반영하게 합니다.
2. **원자적 연산(Atomic Operations):** Test-and-Set이나 Compare-and-Swap (CAS)과 같은 하드웨어 수준에서 제공되는 원자적 명령어입니다. 이 명령어들은 읽기, 수정, 쓰기 작업을 단일하고 분리할 수 없는 작업으로 수행하여, 다중 코어가 동시에 접근해도 데이터 손상을 방지합니다. 락(Lock) 기반 동기화 도구(뮤텍스, 세마포어)의 기본적인 구현에 사용됩니다.
3. **락 기반 동기화 도구(Lock-based Synchronization Tools):** 뮤텍스, 세마포어, 모니터 등은 내부적으로 원자적 연산이나 메모리 배리어를 사용하여 공유 자원에 대한 배타적 접근을 보장함으로써 경합 조건을 방지합니다. 락을 획득한 스레드만이 임계 구역에 진입할 수 있도록 하여 데이터 일관성을 유지합니다.
4. **락-프리(Lock-Free) 및 웨이트-프리(Wait-Free) 알고리즘:** 락을 사용하지 않고 동시성을 관리하는 고급 알고리즘입니다. 락 경쟁으로 인한 직렬화 구간을 줄여 성능을 향상시키고 교착 상태나 라이브락의 위험을 피할 수 있습니다. 하지만 설계 및 검증이 매우 복잡하며, CAS와 같은 원자적 연산에 크게 의존합니다.

---
1. 메모리 관리의 배경 및 문제점
운영 체제는 메인 메모리를 효율적으로 관리하여 여러 프로세스가 동시에 원활하게 실행될 수 있도록 지원합니다. 초기 메모리 관리 방식에는 여러 한계점이 있었습니다.
•
메모리 관리 장치 (MMU: Memory Management Unit)
◦
MMU는 메모리 관리를 하드웨어적으로 지원하는 장치입니다
.
◦
단순한 MMU는 베이스 레지스터(Base Register)와 리밋 레지스터(Limit Register)를 포함합니다
.
◦
베이스 레지스터는 한 프로세스의 물리적 메모리 시작 위치를 지정하며
, 페이징 시스템에서는 프레임의 시작 주소를 의미합니다
.
◦
리밋 레지스터는 프로세스가 할당받을 수 있는 메모리 공간의 크기를 제한합니다
.
◦
이 레지스터들은 다른 프로세스와의 메모리 영역 구분을 통해 서로 침범할 수 없도록 하는 안전장치 역할을 합니다
.
•
프로세스의 주소 공간 및 연속 할당
◦
프로세스(프로그램)는 일반적으로 0번지부터 실행 파일의 크기(파일 사이즈)만큼 상대적인 논리적 번지로 커집니다
.
◦
물리적 메모리 할당 시, 프로세스는 진입점(시작 위치)으로부터 연속적인 메모리 공간을 할당받았습니다
.
◦
각 프로세스의 크기가 다르기 때문에 메모리 할당이 가변적으로 이루어집니다
.
•
외부 단편화 (External Fragmentation)
◦
시간이 지나면서 프로세스가 메모리를 반납하면 그 자리에 '홀(Hole)'이라는 빈 공간이 생깁니다
.
◦
새로운 프로세스가 이 홀에 들어갈 수 있지만, 홀의 크기가 너무 작아 다른 프로세스가 사용하기 어려울 경우 메모리 단편화가 발생합니다
.
◦
이는 프로세스 외부에 생기므로 '외부 단편화'라고 부르며, 메모리 누수 현상을 초래하여 효율적인 메모리 관리를 어렵게 만듭니다
. 이러한 단편화 영역이 많아지면 메모리 관리가 매우 힘들어집니다
.
2. 페이징 (Paging) 시스템

페이징은 연속적인 메모리 할당이 아닌, 고정된 크기의 블록으로 메모리를 나누어 관리함으로써 외부 단편화 문제를 해결하는 기법입니다
.
•
개념 및 특징
◦
프로세스의 물리적 주소 공간은 비연속적일 수 있으며, 사용 가능한 물리적 메모리가 있을 때마다 할당됩니다
.
◦
이는 외부 단편화를 피하고
, 가변적인 크기의 메모리 청크(덩어리) 문제를 방지합니다
.
◦
프레임 (Frame): 메인 메모리를 고정된 크기의 블록으로 나눈 단위입니다
. 각 프레임에는 번호가 부여됩니다. 프레임의 크기는 2의 거듭제곱(512 바이트에서 16 메가바이트 사이)으로 정해집니다
.
◦
페이지 (Page): 프로세스(논리적 메모리)도 메인 메모리의 프레임과 동일한 고정된 크기의 블록으로 나누어지며, 이를 페이지라고 합니다
. 각 페이지에는 번호가 부여됩니다. 프로세스의 모든 코드가 즉시 필요하지 않을 수 있으므로 페이지 단위로 나누어 관리합니다
.
◦
CPU의 착각: CPU는 데이터나 명령들이 메모리에 연속적으로 정렬되어 있는 것처럼 인식하지만
, 실제로는 작은 블록 조각들을 순서에 맞춰 조립하여 접근하는 방식입니다
.
◦
백킹 스토어 (Backing Store): 프로그램의 원본 파일이 저장되는 하드 디스크도 페이지 단위로 나뉘어 관리됩니다
.
◦
자유 프레임 (Free Frames): 시스템은 모든 자유 프레임을 추적하며
, 프로그램 실행을 위해 N개의 페이지가 필요하면 N개의 자유 프레임을 찾아 프로그램을 로드합니다
.
•
주소 변환 방식 (Address Translation Scheme)
◦
CPU가 생성하는 논리 주소(Logical Address)는 **페이지 번호 (p)**와 **페이지 오프셋 (d)**으로 나뉩니다
.
◦
페이지 번호 (p): 페이지 테이블의 인덱스로 사용되며
, 물리적 메모리에서 각 페이지의 시작 주소(프레임 번호)를 포함합니다
.
◦
페이지 오프셋 (d) (변위: Displacement): 페이지의 시작 주소(프레임 번호)와 결합되어 메모리 유닛으로 전송되는 물리적 메모리 주소를 정의합니다
. 페이지 오프셋은 가상 주소와 물리 주소에서 동일하게 사용됩니다
.
◦

계산 예시: 주어진 논리 주소 공간이 2^m이고 페이지 크기가 2ⁿ이라면
:
▪
페이지 오프셋(d)은 n비트가 됩니다
.
▪
페이지 번호(p)는 (m-n)비트가 됩니다
.
▪
예시: 32비트 주소 체계(2³²)에서 4KB(2¹² 바이트) 페이지를 사용하면
:
•
페이지 오프셋은 12비트가 되고
.
•
페이지 번호는 20비트(32 - 12)가 되어 약 100만(2²⁰) 개의 페이지를 나타낼 수 있습니다
.
•
페이지 테이블 (Page Table)
◦
논리 주소를 물리 주소로 변환하기 위해 사용됩니다
.
◦
프로세스의 페이지가 어떤 프레임에 위치해 있는지를 나타내는 테이블입니다
.
◦
페이지 번호로 인덱스되어 있으며, 해당 페이지가 매핑된 프레임 번호를 저장합니다
.
◦
페이지가 현재 메모리에 없을 경우, 하드 디스크(백킹 스토리지)의 원본 파일 위치를 가리킬 수 있습니다
. 필요 시 하드 디스크에서 데이터를 읽어와 비어있는 프레임에 할당한 후 페이지 테이블을 업데이트합니다
.
◦
프레임 테이블 (Frame Table): 물리적 메모리의 프레임을 관리하며, 각 프레임 번호가 어떤 프로세스에 할당되었는지, 또는 비어있는지 등의 속성을 나타냅니다
. 새로운 프로세스가 메모리에 매핑될 때 이 테이블을 참조하여 비어 있는 프레임을 순차적으로 할당합니다
.
•
페이징 예시:
◦
논리 주소 공간 2⁴ (m=4)에서 페이지 크기 2² (n=2, 즉 4바이트)를 사용하고, 물리 메모리가 32바이트(8 페이지)인 경우
:
▪
페이지 번호는 (4-2) = 2비트, 페이지 오프셋은 2비트가 됩니다
.
▪
페이지 0부터 3까지의 논리 주소는 페이지 테이블에 따라 비연속적인 물리 프레임(예: 5번, 6번, 1번, 2번 프레임)에 매핑됩니다
. CPU는 페이지 테이블을 보고 순서대로 실행하므로 연속적인 것처럼 느낍니다

.
▪
리눅스 시스템에서 getconf PAGESIZE 명령어를 통해 프레임/페이지 크기를 확인할 수 있습니다 (예: 4096 바이트, 즉 4KB)
.
3. 페이징의 장단점 및 메모리 보호
페이징은 외부 단편화를 해결하지만, 새로운 종류의 단편화와 관리 오버헤드를 발생시킵니다.
•
내부 단편화 (Internal Fragmentation)
◦
페이징 시스템에서는 외부 단편화가 존재하지 않지만, 내부 단편화가 발생할 수 있습니다
.
◦
페이지 및 프레임 크기는 일반적으로 2의 거듭제곱으로 정의됩니다 (예: 512B ~ 16MB)
.
◦
프로세스 크기가 페이지 크기의 배수가 아닐 경우, 마지막 페이지의 일부 공간이 사용되지 않고 남게 됩니다. 이 부분이 내부 단편화가 됩니다
.
◦
계산 예시: 2,048 바이트(2KB) 페이지 크기에서 72,766 바이트 프로세스가 있다면
:
▪
35개의 페이지를 완전히 채우고 1,086 바이트가 남습니다
.
▪
이 1,086 바이트를 위해 하나의 2,048 바이트 프레임이 할당되고, 나머지 962 바이트(2048 - 1086)가 내부 단편화로 남습니다
.
◦
최악의 경우: 1 프레임 - 1 바이트만큼의 단편화가 발생할 수 있습니다 (예: 4KB 프레임에 1바이트만 저장 시 4095바이트 단편화)
.
◦
평균적으로: 1/2 프레임 크기 정도의 단편화가 발생하며
, 이는 일반적으로 감수할 만한 수준으로 간주됩니다
.
◦
페이지 크기의 영향: 프레임 크기를 작게 하면 내부 단편화는 줄어들지만, 각 페이지 테이블 엔트리를 추적하는 데 더 많은 메모리가 필요하여 페이지 테이블의 크기가 커지는 문제가 발생합니다
. 페이지 크기는 시간이 지남에 따라 증가하는 추세입니다 (예: Solaris는 8KB와 4MB 페이지 크기를 지원)
.
•
페이지 테이블 구현 및 성능 문제
◦
페이지 테이블은 일반적으로 메인 메모리에 저장됩니다
.
◦
PTBR (Page-Table Base Register): 페이지 테이블을 가리킵니다
.

◦
PTLR (Page-Table Length Register): 페이지 테이블의 크기를 나타냅니다
.
◦
두 번의 메모리 접근 문제: 이 방식에서는 데이터나 명령에 접근할 때마다 두 번의 메모리 접근이 필요합니다
.
1.
페이지 테이블에 접근하여 프레임 위치를 확인
.
2.
해당 프레임에 접근하여 실제 데이터를 읽음
.
◦
이는 시스템 성능 저하로 이어집니다
.
•
변환 색인 버퍼 (TLB: Translation Look-Aside Buffer)
◦
두 번의 메모리 접근 문제를 해결하기 위한 고속 하드웨어 캐시입니다
. '연관 메모리(Associative Memory)' 또는 '연관 레지스터(Associative Register)'라고도 합니다
.
◦
TLB는 CPU와 거의 같은 속도로 동작하여 페이지 테이블에 대한 캐시 역할을 수행합니다
.
◦
ASID (Address-Space Identifiers): 일부 TLB는 각 TLB 엔트리에 ASID를 저장하여 각 프로세스를 고유하게 식별하고 주소 공간 보호를 제공합니다. 이를 통해 컨텍스트 스위치 시 TLB를 전체적으로 플러시(비우는)할 필요가 없어집니다
.
◦
크기: TLB는 일반적으로 작습니다 (64에서 1,024 엔트리)
.
◦
TLB Miss: 원하는 페이지 정보가 TLB에 없을 경우, 메인 메모리의 페이지 테이블에서 정보를 가져온 후 TLB에 저장합니다
. 이 경우 두 번의 메모리 접근이 필요합니다
.
◦
교체 정책: TLB도 캐시이므로 교체 정책이 고려되어야 하며, 일부 엔트리는 영구적인 고속 접근을 위해 '고정(wired down)'될 수 있습니다
.
◦
유효 접근 시간 (EAT: Effective Access Time)
▪
적중률 (Hit Ratio): 원하는 페이지 번호를 TLB에서 찾는 비율입니다
. 적중률이 높을수록 성능이 향상됩니다
.
▪
계산 예시: 메모리 접근 시간이 10 나노초(ns)일 때
:

•
80% 적중률: EAT = 0.80 * 10ns (TLB 히트) + 0.20 * 20ns (TLB 미스, 2번 접근) = 12 나노초
. 이는 접근 시간의 20% 지연을 의미합니다
.
•
99% 적중률: EAT = 0.99 * 10ns + 0.01 * 20ns = 10.1 나노초
. 이는 접근 시간의 1% 지연만을 의미합니다
.
•
메모리 보호 (Memory Protection)
◦
메모리 보호는 각 프레임에 보호 비트(protection bit)를 연결하여 구현됩니다. 이 비트는 읽기 전용(read-only) 또는 읽기/쓰기(read-write) 접근 허용 여부를 나타냅니다
. 페이지 실행 전용(execute-only) 등 더 많은 비트를 추가할 수도 있습니다
.
◦
유효-무효 비트 (Valid-Invalid bit): 페이지 테이블의 각 엔트리에 유효-무효 비트가 붙어 있습니다
.
▪
'유효(valid)' (1)는 해당 페이지가 프로세스의 논리적 주소 공간에 유효하게 존재하며 합법적인 페이지임을 나타냅니다
.
▪
'무효(invalid)' (0)는 해당 페이지가 프로세스의 논리적 주소 공간에 없음을 나타냅니다
.
◦
PTLR(Page-Table Length Register)을 사용하여 보호할 수도 있습니다
.
◦
어떤 위반(무효 페이지 접근 등)이 발생하면 커널로 트랩(trap)이 발생하여 프로그램 수행이 정지될 수 있습니다
.
◦
예시: 14비트 주소 공간(0번지~16,383번지)에서 2KB 페이지를 사용하는 프로세스가 0번지부터 1468 번지까지만 사용하는 경우, 이 프로세스의 6번 및 7번 페이지 엔트리(실제 사용 범위를 벗어남)는 '무효'로 표시됩니다
. 이 무효 공간에 접근 시 트랩이 발생합니다
.
•
공유 페이지 (Shared Pages)
◦
공유 코드 (Shared Code): 읽기 전용(read-only)이거나 재진입 가능(reentrant)한 코드(예: 텍스트 편집기, 컴파일러, 윈도우 시스템)는 여러 프로세스 간에 단일 사본으로 공유될 수 있습니다
. 이는 여러 스레드가 동일한 프로세스 공간을 공유하는 것과 유사합니다
.
◦
읽기-쓰기 페이지의 공유가 허용될 경우 프로세스 간 통신(IPC)에도 유용합니다
.
◦
개인 코드 및 데이터 (Private Code and Data): 각 프로세스는 코드와 데이터의 별도 사본을 유지합니다
. 개인 코드 및 데이터 페이지는 논리적 주소 공간의 어느 곳에나 나타날 수 있습니다

.
4. 페이지 테이블의 구조
단일 페이지 테이블의 크기 문제는 현대 컴퓨터 시스템에서 중요한 제약 사항이 됩니다.
•
페이지 테이블 크기 문제
◦
단순한 페이지 테이블 방식으로는 메모리 구조가 매우 커질 수 있습니다
.
◦
예시: 32비트 논리 주소 공간(현대 컴퓨터)과 4KB(2¹²) 페이지 크기를 가정하면
:
▪
페이지 테이블은 100만 개(2³² / 2¹² = 2²⁰)의 엔트리를 가집니다
.
▪
각 엔트리가 4바이트라고 가정하면, 하나의 프로세스에 대한 페이지 테이블만으로도 4MB(2²⁰ * 4바이트)의 물리적 메모리가 필요합니다
.
▪
이 4MB를 메인 메모리에 연속적으로 할당하는 것은 비효율적입니다
.
▪
만약 프로세스가 100개 이상이라면 페이지 테이블만으로 수백 MB에서 기가바이트 단위의 메모리가 소모될 수 있습니다
.
◦
이러한 문제를 해결하기 위해 페이지 테이블을 더 작은 단위로 분할하는 여러 기법이 제안되었습니다
:
▪
계층적 페이징 (Hierarchical Paging)
▪
해시 페이지 테이블 (Hashed Page Tables)
▪
역 페이지 테이블 (Inverted Page Tables)
•
계층적 페이징 (Hierarchical Page Tables)
◦
논리 주소 공간을 여러 페이지 테이블로 분할하는 기법입니다
.
◦
간단한 방법은 2단계 페이지 테이블이며, 이 경우 페이지 테이블 자체를 페이징하는 방식입니다
.
◦
2단계 페이징 예시: 32비트 머신에서 1KB(2¹⁰) 페이지 크기를 사용할 경우
:
▪
논리 주소는 22비트 페이지 번호와 10비트 페이지 오프셋으로 나뉩니다
.
▪
페이지 테이블을 페이징하므로, 22비트 페이지 번호는 다시 두 부분으로 나뉩니다

:
•
10비트 페이지 번호 (p1): 아우터 페이지 테이블(Outer Page Table)의 인덱스로 사용
.
•
12비트 페이지 오프셋 (p2): 이너 페이지 테이블(Inner Page Table) 페이지 내의 변위(displacement)로 사용
.
▪
이 방식은 **전방 매핑 페이지 테이블(forward-mapped page table)**이라고도 불립니다
.
▪
이 경우 페이지 테이블의 총 용량은 4MB에서 80KB(1024 * 4바이트 + 1024 * 4바이트) 수준으로 크게 줄어들 수 있습니다
.
◦
64비트 논리 주소 공간: 2단계 페이징으로도 충분하지 않습니다
.
▪
4KB 페이지 크기를 사용하면 페이지 테이블은 2⁵²개의 엔트리를 가집니다
.
▪
2단계 방식이라도 아우터 페이지 테이블은 2⁴²개의 엔트리(또는 2⁴⁴ 바이트)를 가질 수 있습니다
.
▪
해결책으로 2차 아우터 페이지 테이블을 추가할 수 있지만, 예시에서는 여전히 2³⁴ 바이트 크기이며, 하나의 물리 메모리 위치에 접근하기 위해 4번의 메모리 접근이 필요할 수 있습니다
. 따라서 3단계 페이징 등이 필요할 수 있습니다
.
•
해시 페이지 테이블 (Hashed Page Tables)
◦
32비트보다 큰 주소 공간에서 흔히 사용됩니다
.
◦
가상 페이지 번호가 페이지 테이블로 해싱됩니다
.
◦
이 페이지 테이블은 동일한 위치로 해싱된 요소들의 체인(chain)을 포함합니다
.
◦
각 요소는 다음을 포함합니다: (1) 가상 페이지 번호, (2) 매핑된 페이지 프레임 값, (3) 다음 요소에 대한 포인터
.
◦
가상 페이지 번호는 이 체인에서 일치하는 것을 찾기 위해 비교됩니다. 일치하는 것을 찾으면 해당 물리적 프레임이 추출됩니다
.
◦
64비트 주소의 변형: 클러스터드 페이지 테이블(clustered page tables)은 해시 방식과 유사하지만, 각 엔트리가 1개 대신 여러 페이지(예: 16개)를 참조합니다

. 이는 메모리 참조가 비연속적이고 분산된 희소 주소 공간(sparse address spaces)에 특히 유용합니다
.
•
역 페이지 테이블 (Inverted Page Table)
◦
각 프로세스가 모든 가능한 논리적 페이지를 추적하는 페이지 테이블을 가지는 대신, 모든 물리적 페이지를 추적합니다
.
◦
메모리의 각 실제 페이지마다 하나의 엔트리가 있습니다
.
◦
엔트리는 해당 실제 메모리 위치에 저장된 페이지의 가상 주소와 해당 페이지를 소유한 프로세스에 대한 정보를 포함합니다
.
◦
페이지 테이블을 저장하는 데 필요한 메모리는 줄어들지만, 페이지 참조 발생 시 테이블을 검색하는 데 필요한 시간은 증가합니다
.
◦
해시 테이블을 사용하여 검색을 한두 개의 페이지 테이블 엔트리로 제한할 수 있습니다
.
◦
TLB는 접근 속도를 높일 수 있습니다
.
◦
과제: 공유 메모리를 구현하는 방법이 복잡해집니다 (가상 주소에서 공유 물리 주소로의 단일 매핑 필요)
.
5. 스와핑 (Swapping)
스와핑은 물리적 메모리 공간이 부족할 때 프로세스를 임시로 보조 저장 장치로 옮기는 기법입니다.
•
개념
◦
프로세스는 메모리에서 **백킹 스토어(backing store)**로 일시적으로 교체(swap out)될 수 있으며, 나중에 실행을 위해 다시 메모리로 가져올(swap in) 수 있습니다
.
◦
이를 통해 프로세스의 총 물리적 메모리 공간이 실제 물리적 메모리 용량을 초과할 수 있습니다
.
•
백킹 스토어
◦
모든 사용자에게 할당된 모든 메모리 이미지의 복사본을 수용할 수 있을 만큼 충분히 크고 빠른 디스크여야 합니다
.
◦
이러한 메모리 이미지에 대한 직접 접근을 제공해야 합니다
.
•
롤 아웃, 롤 인 (Roll out, Roll in)

◦
우선순위 기반 스케줄링 알고리즘에 사용되는 스와핑 변형입니다
.
◦
우선순위가 낮은 프로세스는 우선순위가 높은 프로세스를 로드하고 실행하기 위해 스왑 아웃됩니다
.
•
스와핑 시간
◦
스와핑 시간의 주요 부분은 전송 시간이며, 총 전송 시간은 스왑되는 메모리 양에 정비례합니다
.
•
레디 큐 (Ready Queue)
◦
시스템은 디스크에 메모리 이미지를 가진, 실행 준비가 된 프로세스들의 레디 큐를 유지합니다
.
•
스와핑 고려사항
◦
스왑 아웃된 프로세스가 동일한 물리적 주소로 다시 스왑 인될 필요가 있는지는 주소 바인딩 방식에 따라 달라집니다
.
◦
프로세스 메모리 공간으로/로부터의 보류 중인 I/O 또한 고려해야 합니다
.
◦
현대 시스템에서의 스와핑: UNIX, Linux, Windows 등 많은 시스템에서 수정된 버전의 스와핑이 사용됩니다
.
▪
스와핑은 일반적으로 비활성화되어 있습니다
.
▪
할당된 메모리 양이 임계값을 초과하면 시작됩니다
.
▪
메모리 요구량이 임계값 미만으로 줄어들면 다시 비활성화됩니다
.

운영체제 핵심 개념 및 동기화 도구
1. 동시성 문제 및 동기화의 필요성 (Concurrency Problems & Need for Synchronization)
1.1. 배경 (Background)
•
프로세스는 동시적으로 실행될 수 있으며
, 실행 중 언제든지 중단될 수 있습니다
.
•
공유 데이터에 대한 동시 접근은 데이터 불일치를 초래할 수 있습니다
.
•
데이터 일관성 유지는 협력하는 프로세스들의 질서 있는 실행을 보장하는 메커니즘을 필요로 합니다
.
•
생산자-소비자 문제에서 카운터가 동시적으로 업데이트될 때 발생하는 레이스 컨디션(경쟁 상태)이 이러한 문제의 예시입니다
.
1.2. 레이스 컨디션 (Race Condition)
•
정의: 여러 프로세스가 공유 데이터에 동시적으로 접근하여 데이터를 조작할 때, 접근 순서에 따라 실행 결과가 달라지는 상황을 레이스 컨디션이라고 합니다
.
•
예시:
◦
카운터 문제: count++와 count-- 연산은 여러 기계어 명령으로 분해될 수 있습니다
. 예를 들어, count = 5인 상태에서 생산자와 소비자가 동시에 count++와 count--를 실행할 경우, 최종 count 값이 4 또는 6이 되는 등 예상치 못한 결과가 발생할 수 있습니다. 이는 count 변수를 레지스터로 읽어와 연산하고 다시 메모리에 쓰는 과정에서 컨텍스트 스위칭이 발생하여 데이터 불일치를 초래하기 때문입니다
.
◦
fork() 시스템 호출: 커널 변수 next_available_pid에 대한 레이스 컨디션이 발생하면, 두 개의 다른 프로세스에 동일한 PID가 할당될 수 있습니다
.
•
레이스 컨디션은 동시 실행 환경에서 공유 자원에 대한 접근 제어가 제대로 이루어지지 않을 때 발생합니다
.
1.3. 임계 구역 문제 (Critical-Section Problem)
•
정의: n개의 프로세스들이 공유 변수 변경, 테이블 업데이트, 파일 쓰기 등 공유 자원에 접근하는 코드 세그먼트인 임계 구역(Critical Section)을 가질 때 발생하는 문제입니다
. 하나의 프로세스가 임계 구역에서 실행 중일 때, 다른 어떤 프로세스도 임계 구역에 진입할 수 없도록 프로토콜을 설계하는 것이 임계 구역 문제입니다
.
•
프로세스 구조: 각 프로세스는 임계 구역 진입을 요청하는 진입 구역(Entry Section), 실제 공유 자원을 사용하는 임계 구역(Critical Section), 임계 구역 사용 후 나가는 퇴출 구역(Exit Section), 그리고 나머지 코드인 나머지 구역(Remainder Section)으로 구성됩니다
.

[시험 출제 예상] 임계 구역 문제 해결을 위한 3가지 요구 조건 (Requirements for a solution to the critical-section problem)
문제: 임계 구역 문제의 해결책이 반드시 만족해야 하는 세 가지 요구 조건을 설명하시오. 각 조건이 왜 중요한지 간략히 언급하시오.
해설: 임계 구역 문제의 해결책은 다음 세 가지 요구 조건을 만족해야 합니다
.
1.
상호 배제 (Mutual Exclusion)
◦
설명: 만약 프로세스 Pi가 자신의 임계 구역에서 실행 중이라면, 다른 어떤 프로세스도 그들 자신의 임계 구역에서 실행될 수 없어야 합니다
. 즉, 공유 자원은 한 번에 하나의 프로세스만 접근할 수 있도록 보장되어야 합니다
.
◦
중요성: 이는 공유 데이터의 일관성을 유지하는 가장 기본적인 조건입니다. 이 조건이 충족되지 않으면 레이스 컨디션이 발생하여 데이터가 손상될 수 있습니다.
2.
진행 (Progress)
◦
설명: 임계 구역에서 실행 중인 프로세스가 없고, 일부 프로세스들이 임계 구역에 진입하고자 할 때, 다음 임계 구역에 진입할 프로세스의 선택은 무기한 연기될 수 없어야 합니다
. 즉, 임계 구역 진입을 기다리는 프로세스들이 무한정 기다리지 않고 언젠가는 진입할 기회를 얻어야 합니다
.
◦
중요성: 이 조건은 시스템의 효율성과 응답성을 보장합니다. 어떤 프로세스도 불필요하게 대기하지 않도록 하여 시스템 자원의 활용을 최적화합니다.
3.
한정된 대기 (Bounded Waiting)
◦
설명: 한 프로세스가 임계 구역 진입을 요청한 후부터 그 요청이 허용될 때까지, 다른 프로세스들이 그들 자신의 임계 구역에 진입할 수 있는 횟수에 대한 상한(bound)이 존재해야 합니다
.
▪
각 프로세스는 0이 아닌 속도로 실행된다고 가정합니다
.
▪
n개 프로세스의 상대적 속도에 대한 가정은 없습니다
.
◦
중요성: 이 조건은 특정 프로세스의 기아 상태(Starvation)를 방지합니다
. 어떤 프로세스도 무한정 기다리지 않도록 보장하여 공평성을 유지합니다.
2. 임계 구역 문제 해결 기법 (Solutions to Critical-Section Problem)
2.1. 하드웨어 기반 해결책 (Hardware Support for Synchronization)
2.1.1. 인터럽트 비활성화 (Interrupt-based Solution)
•
방법: 진입 구역에서 인터럽트를 비활성화하고, 퇴출 구역에서 인터럽트를 활성화합니다
.
•
문제점:
◦

임계 구역 코드가 매우 길 경우, 인터럽트가 장시간 비활성화되어 시스템 응답성이 저하될 수 있습니다
.
◦
일부 프로세스가 임계 구역에 영원히 진입하지 못하는 기아 상태가 발생할 수 있습니다
.
◦
멀티 CPU 시스템에서는 하나의 CPU에서 인터럽트를 비활성화해도 다른 CPU에서는 여전히 공유 자원에 접근할 수 있어 상호 배제를 보장할 수 없습니다
. 이는 확장성이 좋지 않은 방법입니다
.
2.1.2. 하드웨어 명령어 (Hardware Instructions)
•
많은 시스템이 임계 구역 코드 구현을 위한 하드웨어 지원을 제공합니다
.
•
특별한 하드웨어 명령어는 워드 내용을 원자적으로(atomically) 테스트하고 수정하거나, 두 워드의 내용을 원자적으로 교환할 수 있도록 합니다
.
•
종류:
◦
Test-and-Set 명령어 (Test-and-Set Instruction)
▪
정의: boolean test_and_set (boolean *target) 함수는 *target의 원래 값을 반환하고, *target을 true로 설정합니다
. 이 과정은 원자적으로 실행됩니다
.
▪
임계 구역 해결: 공유 boolean 변수 lock (초기값 false)을 사용합니다
.
•
진입 구역: while (test_and_set(&lock)); (busy wait)
•
퇴출 구역: lock = false;
◦
Compare-and-Swap 명령어 (Compare-and-Swap Instruction)
▪
정의: int CompareAndSwap(int *address, int expected, int new) 함수는 *address가 expected와 같으면 *address를 new로 설정하고 1을 반환합니다. 그렇지 않으면 0을 반환합니다
. 이 과정은 원자적으로 실행됩니다
.
▪
활용: 락 없이 데이터 구조를 구축하는 락-프리(lock-free) 방식에 사용될 수 있습니다
. 예를 들어, AtomicIncrement 함수는 CompareAndSwap을 사용하여 락 없이 값을 원자적으로 증가시킬 수 있습니다
.
2.2. 소프트웨어 기반 해결책 (Software Solutions)
2.2.1. Peterson’s Solution
•
개념: 두 개의 프로세스(P0, P1)를 위한 소프트웨어 기반 해결책입니다

. 로드(load) 및 저장(store) 기계어 명령어가 원자적으로 실행된다고 가정합니다
.
•
공유 변수:
◦
int turn;: 누가 임계 구역에 진입할 차례인지를 나타냅니다 (초기값 i 또는 j)
.
◦
boolean flag
;: 각 프로세스가 임계 구역 진입 준비가 되었는지 나타냅니다 (초기값 false). flag[i] = true는 프로세스 Pi가 준비되었음을 의미합니다
.
•
알고리즘 (Process Pi)
:
•
정확성: Peterson의 해결책은 다음과 같은 임계 구역 요구 조건을 모두 만족함을 증명할 수 있습니다
:
1.
상호 배제 (Mutual Exclusion)
2.
진행 (Progress)
3.
한정된 대기 (Bounded Waiting)
[시험 출제 예상] Peterson의 해결책과 현대 아키텍처 (Peterson's Solution and Modern Architecture)
문제: Peterson의 해결책이 이론적으로는 임계 구역의 세 가지 요구 조건을 모두 만족함에도 불구하고, 현대 컴퓨터 아키텍처에서는 올바르게 작동하지 않을 수 있는 이유를 설명하고 예시를 제시하시오.
해설: Peterson의 해결책은 알고리즘을 시연하는 데 유용하지만
, 현대 컴퓨터 아키텍처에서는 올바르게 작동한다는 보장이 없습니다
.
•
문제의 원인: 성능 향상을 위해 프로세서(CPU)나 컴파일러는 의존성이 없는 명령어들의 순서를 재배열(reorder)할 수 있습니다
. 단일 스레드 환경에서는 이러한 재배열이 최종 결과에 영향을 미치지 않지만 (결과는 항상 동일), 다중 스레드 환경에서는 재배열이 일관되지 않거나 예상치 못한 결과를 초래할 수 있습니다
.
•
Peterson 해결책의 실패 예시:
◦
Peterson의 알고리즘에서 flag[i] = true;와 turn = j; 명령어가 재배열될 수 있습니다
. 예를 들어, 프로세스 P0가 turn = 1;을 먼저 실행하고 flag = true;를 나중에 실행하며, 이와 동시에 P1도 유사하게 재배열된 명령어를 실행할 경우, 두 프로세스 모두 임계 구역에 진입할 수 있게 되어 상호 배제 조건이 깨질 수 있습니다
.
◦
현대 아키텍처 예시: 두 스레드가 boolean flag = false; int x = 0;을 공유할 때, 스레드 2가 x = 100; flag = true;를 실행한다고 가정해 봅시다. 만약 이 두 명령어가 재배열되어 flag = true;가 먼저 실행되고 x = 100;이 나중에 실행된다면, 스레드 1(while (!flag); print x;)은 x가 100으로 업데이트되기 전에 flag가 true가 된 것을 보고 x를 출력하여 0이 출력될 수 있습니다
. 이는 원자성(atomicity)이 깨지는 예시입니다.

•
결론: Peterson의 해결책이 현대 아키텍처에서 올바르게 작동하도록 보장하려면, 메모리 배리어(memory barrier)와 같은 추가적인 동기화 도구가 필요합니다
. 이러한 문제를 이해하는 것은 레이스 컨디션을 더 잘 이해하는 데 도움이 됩니다
.
2.3. 고수준 동기화 도구 (High-level Synchronization Tools)
2.3.1. 뮤텍스 락 (Mutex Locks)
•
개념: 운영체제 설계자들이 임계 구역 문제를 해결하기 위해 구축한 소프트웨어 도구 중 가장 간단한 형태입니다
.
•
동작 방식:
◦
available이라는 불리언(Boolean) 변수를 사용하여 락의 가용 여부를 나타냅니다
.
◦
임계 구역을 보호하기 위해, 먼저 acquire() 함수로 락을 획득하고, 임계 구역 작업을 마친 후 release() 함수로 락을 해제합니다
.
◦
acquire()와 release() 호출은 원자적(atomic)으로 구현되어야 합니다. 이는 일반적으로 compare-and-swap과 같은 하드웨어 원자적 명령어를 통해 구현됩니다
.
•
Busy Waiting (Spinlock): 뮤텍스 락은 acquire() 함수에서 락이 사용 가능해질 때까지 while (!available);과 같이 반복적으로 검사하며 기다리는 **바쁜 대기(busy waiting)**를 필요로 합니다
. 이러한 락을 **스핀락(spinlock)**이라고 부릅니다
.
•
예시 코드: pthread_mutex_lock()으로 락을 획득하고 pthread_mutex_unlock()으로 락을 해제하는 mutex.c 코드는 카운터 문제를 올바르게 해결함을 보여줍니다
.
2.3.2. 세마포어 (Semaphores)
•
개념: 뮤텍스 락보다 더 정교하게 프로세스의 활동을 동기화하는 도구입니다
.
•
구조: 정수 변수 S (세마포어 값)로 구성되며, 이 변수는 오직 두 가지 원자적인 연산인 wait()와 signal() (또는 P()와 V())을 통해서만 접근할 수 있습니다
.
•
연산 정의:
◦
wait(S) (또는 P(S)):
◦
S가 0보다 크거나 같아질 때까지 대기하고, S 값을 1 감소시킵니다
.
◦
signal(S) (또는 V(S)):
◦

S 값을 1 증가시킵니다
.
•
종류:
◦
카운팅 세마포어 (Counting semaphore): 정수 값이 제한 없는 도메인에서 범위 할 수 있습니다
.
◦
이진 세마포어 (Binary semaphore): 정수 값이 0 또는 1 사이에서만 범위 할 수 있습니다
. 뮤텍스 락과 동일하게 사용될 수 있습니다. 카운팅 세마포어를 이진 세마포어로 구현할 수도 있습니다
.
•
활용 예시:
◦
임계 구역 문제 해결: semaphore mutex = 1;을 생성한 후 wait(mutex); Critical Section; signal(mutex);
를 사용합니다
.
◦
실행 순서 동기화: P1의 S1이 P2의 S2보다 먼저 발생해야 할 때, semaphore synch = 0;을 생성합니다
.
▪
P1: S1; signal(synch);
▪
P2: wait(synch); S2;
•
세마포어 구현: wait()와 signal() 연산 자체도 임계 구역 문제를 해결해야 하므로 원자적으로 실행되어야
합니다
. 초기 세마포어 구현은 busy waiting을 포함할 수 있지만, 구현 코드가 짧고 임계 구역이 거의 점유되지
않는다면 큰 문제가 되지 않습니다
.
•
바쁜 대기 없는 세마포어 구현 (Semaphore Implementation with no Busy waiting):
◦
각 세마포어는 대기 큐를 가집니다
. 대기 큐의 각 엔트리는 세마포어 값과 다음 레코드에 대한 포인터를 가집니다
.
◦
두 가지 연산:
▪
block: 현재 프로세스를 적절한 대기 큐에 놓습니다
.
▪
wakeup: 대기 큐에 있는 프로세스 중 하나를 제거하여 준비 큐에 놓습니다
.
◦
wait(S) 연산은 S 값을 감소시킨 후 음수가 되면 프로세스를 block합니다. signal(S) 연산은 S 값을 증가시킨 후
0보다 작거나 같으면 대기 큐의 프로세스를 wakeup합니다
.
•
세마포어 사용의 문제점: 세마포어 연산의 잘못된 사용은 여러 동기화 문제를 야기할 수 있습니다

.
◦
signal(mutex) 다음에 wait(mutex)를 호출하는 경우: 여러 프로세스가 동시에 임계 구역에서 실행될 수 있습니다
.
◦
wait(mutex)를 두 번 호출하는 경우: 프로세스가 두 번째 wait() 호출에서 영구적으로 블록될 수 있습니다
.
◦
wait(mutex) 또는 signal(mutex) 중 하나를 생략하는 경우
.
2.3.3. 모니터 (Monitors)
•
개념: 프로세스 동기화를 위한 편리하고 효과적인 메커니즘을 제공하는 고수준 추상화입니다
.
•
특징:
◦
추상 데이터 타입으로, 내부 변수들은 모니터 내의 프로시저를 통해서만 접근 가능합니다
.
◦
한 번에 오직 하나의 프로세스만 모니터 내에서 활성화될 수 있습니다
. 이는 모니터 자체가 상호 배제를 보장함을 의미합니다
.
•
구조:
•
이러한 구조는 공유 변수 선언, 프로시저 선언, 초기화 코드로 이루어집니다
.
•
세마포어를 이용한 모니터 구현:
◦
semaphore mutex = 1;을 사용하여 모니터의 상호 배제를 보장합니다
.
◦
모니터 내의 각 프로시저 P는 wait(mutex); ... body of P; ... signal(mutex); 형태로 대체됩니다
.
•
조건 변수 (Condition Variables):
◦
모니터 내에서 프로세스가 특정 조건을 만족할 때까지 대기하거나, 조건을 만족하는 프로세스를 깨우기 위해 사용됩니다
.
◦
condition x, y;와 같이 선언됩니다
.
◦
두 가지 연산:
▪
x.wait(): 이 연산을 호출한 프로세스는 x.signal()이 호출될 때까지 중단됩니다

.
▪
x.signal(): x.wait()을 호출했던 프로세스(있다면 하나)를 재개합니다
. 만약 x.wait()으로 대기 중인 프로세스가 없다면, 변수에 아무런 효과를 미치지 않습니다
.
◦
조건 변수를 이용한 모니터 구현 (세마포어 기반):
▪
각 조건 변수 x에 대해 semaphore x_sem = 0;와 int x_count = 0; (대기 중인 프로세스 수)를 가집니다
.
▪
x.wait()는 x_count 증가, mutex 해제 후 x_sem에서 대기하고, 깨어나면 x_count 감소합니다
.
▪
x.signal()은 x_count가 0보다 크면 next_count 증가, x_sem에 signal 후 next에서 대기하고, 깨어나면 next_count 감소합니다
.
◦
프로세스 재개 문제: 여러 프로세스가 조건 변수 x에서 대기 중일 때 x.signal()이 실행되면 어떤 프로세스를 재개할 것인가가 문제입니다
. FCFS는 적절하지 않은 경우가 많습니다
.
◦
조건부 대기 (Conditional-wait): x.wait(c) 형태로 사용하며, c는 우선순위 번호입니다
. 가장 낮은 숫자(c)를 가진 프로세스(가장 높은 우선순위)가 다음에 스케줄됩니다
.
3. 고전적인 동기화 문제 예시 (Classical Problems of Synchronization)
3.1. 생산자-소비자 문제 (Bounded-Buffer Problem)
•
n개의 버퍼가 각각 하나의 항목을 저장할 수 있습니다
.
•
역할:
◦
생산자 (Producer): 소비자를 위해 가득 찬 버퍼를 생산합니다
.
◦
소비자 (Consumer): 생산자를 위해 빈 버퍼를 생산합니다
.
•
공유 데이터:
◦
semaphore mutex = 1;: 버퍼 풀에 대한 접근에 대한 상호 배제를 제공합니다
.
◦
semaphore full = 0;: 가득 찬 버퍼의 수를 세는 세마포어입니다
.
◦
semaphore empty = n;: 빈 버퍼의 수를 세는 세마포어입니다
.

•
생산자 프로세스 구조:
•
소비자 프로세스 구조:
•
C 언어 코드 예시가 제공됩니다
.
3.2. 독자-저술가 문제 (Readers-Writers Problem)
•
개념: 다수의 동시 프로세스들 사이에서 공유되는 데이터 집합이 있을 때, 데이터를 읽기만 하는 독자(readers)와 읽고 쓸 수 있는 저술가(writers)가 존재합니다
.
•
규칙:
◦
다수의 독자가 동시에 데이터를 읽을 수 있습니다
.
◦
오직 한 명의 저술가만이 공유 데이터에 접근할 수 있습니다
.
◦
저술가가 접근 중일 때는 독자를 포함한 다른 어떤 프로세스도 접근할 수 없습니다
.
•
공유 데이터:
◦
semaphore rw_mutex = 1;: 독자와 저술가 프로세스 모두에게 공통인 상호 배제 세마포어입니다
.
◦
semaphore mutex = 1;: read_count 변수가 업데이트될 때 상호 배제를 보장하는 세마포어입니다
.
◦
int read_count = 0;: 현재 객체를 읽고 있는 프로세스의 수를 추적합니다
.
•
저술가 프로세스 구조:
•
독자 프로세스 구조:
•
변형 (Variations):
◦
위의 해결책은 저술가 기아 상태(writer starvation)를 초래할 수 있습니다. 이를 "첫 번째 독자-저술가 문제(First reader-writer problem)"라고 합니다
.
◦
"두 번째 독자-저술가 문제(Second reader-writer problem)"는 저술가가 쓸 준비가 되면 "새로 도착한 독자"는 읽기가 허용되지 않는 변형입니다
.
◦

두 문제 모두 기아 상태를 초래할 수 있으며, 커널이 제공하는 독자-저술가 락(reader-writer locks)으로 해결될 수 있습니다

.

3.3. 식사하는 철학자들 문제 (Dining-Philosophers Problem)

•

개념: N명의 철학자들이 둥근 테이블에 앉아 있으며, 각 철학자는 생각하거나 식사합니다

. 식사하기 위해서는 두 개의 젓가락(chopsticks)이 필요하며, 젓가락은 한 번에 하나씩 집을 수 있습니다. 각 철학자 사이에 젓가락이 하나씩 놓여 있습니다

.

•

공유 데이터: semaphore chopstick

= 1;

. (5명의 철학자 경우)

[시험 출제 예상] 식사하는 철학자들 문제의 세마포어 해결책과 교착 상태

문제: 식사하는 철학자들 문제에 대한 세마포어 기반의 고전적인 알고리즘을 제시하고, 이 알고리즘이 교착 상태(Deadlock)에 빠질 수 있는 시나리오를 설명하시오. 또한, 교착 상태를 해결하기 위한 몇 가지 방법을 제시하시오.

해설:

1. 세마포어 해결책 (Semaphore Solution): 각 철학자 i의 구조는 다음과 같습니다

:

```
while (true) {
    wait(chopstick[i]);              // 왼쪽 젓가락 획득 시도
    wait(chopstick[(i + 1) % 5]);    // 오른쪽 젓가락 획득 시도
    /* eat for awhile */             // 식사
    signal(chopstick[i]);            // 왼쪽 젓가락 반납
    signal(chopstick[(i + 1) % 5]);  // 오른쪽 젓가락 반납
    /* think for awhile */           // 생각
}
```

•

chopstick[i]는 철학자 i의 왼쪽 젓가락을, chopstick[(i+1)%5]는 오른쪽 젓가락을 나타냅니다

.

•

wait() 연산은 젓가락을 획득하는 것을, signal() 연산은 젓가락을 반납하는 것을 의미합니다

.

2. 문제점: 교착 상태 (Deadlock): 이 알고리즘은 교착 상태에 빠질 수 있습니다

.

•

시나리오: 모든 다섯 명의 철학자가 동시에 배가 고파져서 (즉, 동시에 pickup()을 호출하여) 자신의 왼쪽 젓가락을 동시에 집어들 경우

.

◦

모든 chopstick[i] 세마포어의 값이 0이 됩니다.

◦

이제 각 철학자는 자신의 오른쪽 젓가락(chopstick[(i+1)%5])을 집으려 시도합니다. 그러나 이 젓가락은 이미 옆에 있는 철학자가 왼쪽 젓가락으로 집고 있는 상태입니다.

◦

결과적으로, 모든 철학자는 자신이 기다리는 젓가락을 다른 철학자가 놓아주기를 기다리게 되지만, 아무도 젓가락을 놓아주지 않으므로 무한 대기(infinite waiting) 상태에 빠지게 됩니다. 이는 교착 상태의 전형적인 예시입니다
.
3. 교착 상태 해결을 위한 방법: 교착 상태를 해결하기 위한 몇 가지 가능한 방법은 다음과 같습니다
:
•
동시에 테이블에 앉을 수 있는 철학자의 수 제한: 최대 N-1명의 철학자만 동시에 테이블에 앉을 수 있도록 허용합니다. (예: 5명의 철학자 중 4명만 허용)
. 이렇게 하면 항상 적어도 한 명의 철학자가 두 젓가락을 모두 집을 수 있는 기회를 얻게 됩니다.
•
두 젓가락을 동시에 획득: 철학자가 두 젓가락이 모두 사용 가능한 경우에만 젓가락을 집도록 허용합니다. 이를 위해 두 젓가락을 집는 과정을 하나의 임계 구역(critical section)으로 묶을 수 있습니다
.
•
비대칭적인 해결책 (Asymmetric Solution):
◦
홀수 번호의 철학자는 먼저 왼쪽 젓가락을 집고 그 다음 오른쪽 젓가락을 집습니다
.
◦
짝수 번호의 철학자는 먼저 오른쪽 젓가락을 집고 그 다음 왼쪽 젓가락을 집습니다
.
◦
이러한 비대칭적인 순서 부여를 통해 순환 대기를 방지할 수 있습니다.
4. 모니터 해결책 (Monitor Solution): 모니터는 고수준 추상화를 제공하여 교착 상태를 방지하는 데 도움을 줍니다.
•
상태 변수: enum {THINKING, HUNGRY, EATING} state
;를 사용하여 각 철학자의 상태를 구분합니다
.
•
조건 변수: condition self
;를 사용하여 배고프지만 젓가락을 얻지 못하는 철학자 i를 지연시킬 수 있습니다
.
•
test(int i) 프로시저: 철학자 i가 식사 가능한지 확인하는 함수입니다. (state[(i+4)%5] != EATING) && (state[i] == HUNGRY) && (state[(i+1)%5] != EATING) 조건이 만족될 때만 state[i]를 EATING으로 설정하고 self[i].signal()을 호출합니다
.
•
pickup(int i) 프로시저:
•
철학자 i가 배고파지면 test()를 호출하고, 식사할 수 없으면 self[i].wait을 통해 대기합니다
.

•
putdown(int i) 프로시저:
•
젓가락을 놓은 후에는 이웃 철학자들이 식사할 수 있는지 test() 함수를 호출하여 확인합니다
.
•
결과: 모니터 해결책은 교착 상태를 발생시키지 않지만, 기아 상태(starvation)는 여전히 발생할 수 있습니다
. (C 코드 예시 제공
)
4. 교착 상태 (Deadlock)
4.1. 시스템 모델 (System Model)
•
시스템은 다양한 자원들(R1, R2, ..., Rm)로 구성됩니다
.
•
각 자원 유형 Ri는 Wi개의 인스턴스(instances)를 가집니다
.
•
각 프로세스는 자원을 다음 세 단계로 활용합니다
:
◦
요청 (request): 프로세스가 자원을 요청합니다.
◦
사용 (use): 프로세스가 자원을 사용합니다.
◦
해제 (release): 프로세스가 사용을 마친 자원을 해제합니다.
4.2. 교착 상태의 정의 (Definition of Deadlock)
•
교착 상태는 두 개 이상의 프로세스가 서로 상대방이 가지고 있는 자원을 기다리며 무기한으로 블록되는 상태를 말합니다
.
•
예시: 세마포어 S1과 S2가 각각 1로 초기화되어 있을 때, 스레드 T1은 wait(S1); wait(S2);를, 스레드 T2는 wait(S2); wait(S1);을 실행한다고 가정해 봅시다
.
◦
T1이 S1을 획득하고 컨텍스트 스위칭이 발생하여 T2가 S2를 획득합니다
.
◦
이제 T1은 S2를 기다리고, T2는 S1을 기다립니다. 서로 상대방이 자원을 해제하기를 기다리므로 두 스레드 모두 영구적으로 블록됩니다
.
4.3. 라이블락 (Livelock)
•

정의: 스레드가 어떤 동작을 지속적으로 시도하지만 계속 실패하는 상황을 말합니다
. 교착 상태와 달리 스레드들은 계속 실행 중이지만, 실질적인 진행은 이루어지지 않습니다
.
•
예시: pthread_mutex_trylock()을 사용하는 코드에서, 두 스레드가 서로 락을 획득하려고 시도하지만, 다른 스레드가 먼저 락을 획득하면 자신이 가지고 있던 락을 포기하고 다시 시도하는 과정을 무한히 반복할 수 있습니다
. 이로 인해 실제 작업은 수행되지 않고 락을 획득하려는 시도만 반복됩니다
.
•
해결책: 무작위 지연(random delay)을 추가하여 반복적인 간섭을 줄이는 방법이 있습니다
.
[시험 출제 예상] 교착 상태 발생의 네 가지 필요 조건 (Four Necessary Conditions for Deadlock)
문제: 교착 상태가 발생하기 위한 네 가지 필요 조건을 각각 설명하시오. 각 조건이 의미하는 바를 구체적인 예시와 함께 제시하시오.
해설: 교착 상태는 다음 네 가지 조건이 모두 동시에 충족될 때만 발생할 수 있습니다
. 이 중 하나라도 충족되지 않으면 교착 상태는 발생하지 않습니다
.
1.
상호 배제 (Mutual Exclusion)
◦
설명: 최소한 하나의 자원이 비공유 모드(non-sharable mode)로 점유되어야 합니다
. 즉, 한 번에 오직 하나의 스레드/프로세스만이 그 자원을 사용할 수 있어야 합니다. 읽기 전용 파일과 같은 공유 가능한 자원에는 필요하지 않지만, 뮤텍스 락이나 프린터와 같은 비공유 자원에는 필수적입니다
.
◦
예시: 프린터와 같은 장치는 한 번에 하나의 프로세스만 사용할 수 있습니다. 만약 여러 프로세스가 동시에 프린터를 사용하려 한다면 충돌이 발생합니다. 뮤텍스 락(pthread_mutex_t)도 대표적인 상호 배제 자원입니다
.
2.
점유 및 대기 (Hold and Wait)
◦
설명: 스레드/프로세스가 최소한 하나의 자원을 점유(holding)하고 있으면서, 다른 스레드/프로세스에 의해 점유된 추가적인 자원을 획득하기 위해 대기(waiting)해야 합니다
. 즉, 자원을 들고 있으면서 다른 자원을 기다리는 상황입니다.
◦
예시: 프로세스 P0가 자원 R1을 점유한 상태에서 자원 R2를 기다리고, R2는 프로세스 P1이 점유하고 있는 상황.
3.
비선점 (No Preemption)
◦
설명: 자원(예: 락)은 강제로 스레드/프로세스로부터 회수될 수 없으며, 오직 자원을 점유하고 있는 스레드/프로세스가 자신의 작업을 완료한 후 자발적으로(voluntarily)만 해제할 수 있어야 합니다
. 즉, 운영체제가 자원을 강제로 뺏을 수 없는 경우입니다

.
◦
예시: 프로세스가 파일을 쓰기 위해 디스크를 점유하고 있는 동안, 다른 프로세스가 디스크를 필요로 해도 운영체제가 디스크를 강제로 뺏을 수 없습니다. 오직 쓰기 작업이 완료된 후에야 자원이 해제됩니다.
4.
순환 대기 (Circular Wait)
◦
설명: 대기 중인 스레드/프로세스들의 집합 {T0, T1, ..., Tn}이 존재하여, T0는 T1이 점유한 자원을 기다리고, T1은 T2가 점유한 자원을 기다리고, ..., Tn-1은 Tn이 점유한 자원을 기다리고, 최종적으로 Tn은 T0가 점유한 자원을 기다리는 순환 고리(circular chain)가 형성되어야 합니다
.
◦
예시:
▪
T0가 R1을 점유하고 R2를 기다림
▪
T1이 R2를 점유하고 R3를 기다림
▪
T2가 R3를 점유하고 R1을 기다림 이런 상황에서는 T0 -> T1 -> T2 -> T0와 같은 순환 대기 고리가 형성됩니다
.
4.4. 자원 할당 그래프 (Resource-Allocation Graph)
•
개념: 시스템의 스레드와 자원 간의 관계를 시각적으로 표현하여 교착 상태를 직관적으로 파악할 수 있게 해주는 도구입니다
.
•
구성:
◦
정점 (Vertices) V: 시스템의 모든 스레드 집합 T = {T1, T2, ..., Tn}과 모든 자원 유형 집합 R = {R1, R2, ..., Rm}으로 분할됩니다
.
◦
간선 (Edges) E:
▪
요청 간선 (Request edge): 스레드 Ti가 자원 Rj를 요청하고 있음을 나타냅니다 (Ti → Rj)
. 스레드 노드에서 자원 유형 노드로 향하는 화살표로 표시됩니다
.
▪
할당 간선 (Assignment edge): 자원 Rj가 스레드 Ti에 할당되었음을 나타냅니다 (Ti ← Rj)
. 자원 유형 노드에서 스레드 노드로 향하는 화살표로 표시됩니다
.
•
자원 인스턴스 표현: 자원 유형 노드 내에 점으로 표시된 각 인스턴스(객체)는 해당 유형의 사용 가능한 자원 단위를 나타냅니다

.
•
교착 상태 판단:
◦
그래프에 사이클이 없으면 교착 상태는 없습니다
.
◦
그래프에 사이클이 있다면:
▪
자원 유형당 인스턴스가 하나만 있는 경우: 사이클이 존재하면 반드시 교착 상태입니다
.
▪
자원 유형당 인스턴스가 여러 개 있는 경우: 사이클이 존재하더라도 교착 상태가 발생할 가능성이 있습니다
. (예: 자원 R1이 2개 있는데 T1이 R1 하나를 기다리고 T2가 R1 하나를 가지고 있는 경우, T2가 끝나면 T1이 R1을 획득할 수 있으므로 교착 상태가 아닐 수 있습니다
).
4.5. 교착 상태 처리 방법 (Methods for Handling Deadlocks)
•
시스템은 교착 상태를 처리하기 위해 다음 세 가지 방법 중 하나를 선택할 수 있습니다
:
1.
교착 상태 방지 (Deadlock Prevention): 시스템이 교착 상태에 진입하지 않도록 보장합니다
. (네 가지 필요 조건 중 하나를 깨뜨림)
2.
교착 상태 회피 (Deadlock Avoidance): 시스템이 교착 상태에 진입하지 않도록 보장하면서, 더 많은 동시성을 허용합니다
. (자원의 최대 요구량을 미리 알고 동적으로 안전한 상태를 유지)
3.
교착 상태 탐지 및 회복 (Deadlock Detection and Recovery): 시스템이 교착 상태에 진입하는 것을 허용하고, 나중에 교착 상태를 탐지하여 회복합니다
.
4.
문제 무시 (Ignore the problem): 시스템에 교착 상태가 발생하지 않는다고 가정하고 무시합니다. (일반적으로 교착 상태가 드물게 발생하는 시스템에서 사용)
.
5. 교착 상태 방지 (Deadlock Prevention)
•
교착 상태를 유발하는 네 가지 필요 조건 중 하나 이상을 위반함으로써 교착 상태를 방지합니다
.
[시험 출제 예상] 각 교착 상태 발생 조건 위반을 통한 방지 기법
문제: 교착 상태를 발생시키는 네 가지 필요 조건(상호 배제, 점유 및 대기, 비선점, 순환 대기) 각각에 대해, 해당 조건을 위반하여 교착 상태를 방지하는 방법을 설명하고, 각 방법의 장단점을 제시하시오.
해설:
1.

상호 배제 (Mutual Exclusion)
◦
방지 방법: 상호 배제의 필요성 자체를 없애는 것입니다
. 즉, 자원을 공유 가능한(sharable) 자원으로 만들거나, 락 없이(lock-free) 동작하는 데이터 구조를 설계합니다
.
◦
설명: 모든 자원이 동시에 접근 가능하도록 허용하면 상호 배제 조건이 깨지므로 교착 상태는 발생하지 않습니다
. 예를 들어, 읽기 전용 파일은 여러 프로세스가 동시에 읽을 수 있어 상호 배제가 필요 없습니다. 락-프리 자료구조는 Compare-and-Swap (CAS)과 같은 원자적 하드웨어 명령어를 사용하여 명시적인 락 없이 동시성을 확보합니다
.
◦
장점: 교착 상태가 근본적으로 발생하지 않습니다.
◦
단점:
▪
적용의 어려움: 프린터나 공유 메모리 영역과 같은 비공유(non-sharable) 자원에는 상호 배제가 필수적이므로, 모든 자원에 대해 이 방법을 적용하기는 어렵습니다
.
▪
복잡성: 락-프리 자료구조는 구현이 매우 복잡하고, 디버깅이 어렵습니다
. 여전히 라이블락 가능성이 있습니다
.
2.
점유 및 대기 (Hold and Wait)
◦
방지 방법: 프로세스가 자원을 요청할 때 다른 어떤 자원도 점유하고 있지 않음을 보장해야 합니다
. 두 가지 주요 전략이 있습니다:
▪
모든 자원 한 번에 획득: 프로세스가 실행을 시작하기 전에 필요한 모든 자원을 요청하고 할당받거나
.
▪
자원 요청 시 모든 자원 해제: 프로세스가 자원을 요청할 때, 현재 점유하고 있는 자원이 있다면 모두 해제하고, 필요한 모든 자원을 새로 요청하여 할당받은 후에 다시 시작하도록 합니다
.
◦
설명: 이 방법은 프로세스가 자원을 일부만 점유하고 다른 자원을 기다리는 상황 자체를 원천 봉쇄합니다
. pthread_mutex_lock(prevention);과 같은 전역 락을 사용하여 모든 락을 원자적으로 한 번에 획득하는 방식으로 구현될 수 있습니다
.
◦
장점: 교착 상태를 확실히 방지합니다.
◦

단점:

- 자원 활용률 저하: 프로세스가 실제로 필요할 때까지 자원을 점유해야 하므로 자원 활용률이 낮아집니다.
- 기아 상태 가능성: 특정 프로세스가 필요한 모든 자원을 한 번에 획득하기 어렵다면 무한히 기다릴 수 있습니다.
- 캡슐화 위반: 함수 호출 시 어떤 락이 필요한지 미리 알아야 하므로 모듈성(modularity)을 해칩니다.
- 동시성 감소: 모든 락을 미리 획득해야 하므로 동시성(concurrency)이 줄어듭니다.

3. 비선점 (No Preemption)

- 방지 방법: 자원을 선점 가능하게 만듭니다. 즉, 어떤 프로세스가 현재 점유하고 있는 자원 외에 추가 자원을 요청했는데 즉시 할당받을 수 없다면, 현재 점유하고 있던 모든 자원을 해제(선점)하게 합니다. 이 프로세스는 나중에 모든 필요한 자원(기존에 점유했던 자원과 새로 요청한 자원)을 다시 얻을 수 있을 때만 다시 시작됩니다.
- 설명: pthread_mutex_trylock()과 같은 비블로킹(non-blocking) 락 획득 시도 함수를 사용하여 구현될 수 있습니다. 락 획득에 실패하면, 이미 획득한 락을 해제하고 나중에 다시 시도하는 방식입니다.
- 장점: 융통성 있는 락 획득 방식을 제공하여 교착 상태를 방지할 수 있습니다.
- 단점:
  - 라이블락 가능성: 두 스레드가 서로 락을 풀고 다시 시도하는 과정을 무한히 반복하여 라이블락에 빠질 수 있습니다. (무작위 지연을 통해 완화 가능).
  - 자원 상태 관리 복잡성: 중간에 획득한 자원(메모리 등)을 실패 시 다시 해제해야 하므로 코드 복잡성이 증가합니다.
  - 진정한 선점이 아님: 엄밀히 말해 강제적인 선점(forcible preemption)이라기보다는 스레드 스스로가 락 소유권을 포기하는 방식입니다.

4.

순환 대기 (Circular Wait)
◦
방지 방법: 모든 자원 유형에 대해 총체적인 순서(total ordering)를 부여하고, 프로세스는 이 순서에 따라 자원을 오름차순으로만 요청하도록 요구합니다
.
◦
설명: 이 방법은 교착 상태 방지 기법 중 가장 실용적이고 자주 사용되는 방법입니다
. 예를 들어, 시스템에 L1과 L2 두 개의 락만 있다면, 항상 L1을 L2보다 먼저 획득하도록 규칙을 정하는 식입니다. 복잡한 시스템에서는 부분 순서(partial ordering)를 사용할 수도 있습니다. 락의 메모리 주소를 사용하여 낮은 주소의 락을 먼저 획득하는 방법으로도 구현할 수 있습니다
.
◦
장점: 구현이 비교적 간단하며 효과적입니다.
◦
단점:
▪
설계의 어려움: 복잡한 시스템에서는 모든 락에 대한 총체적/부분적 순서를 설계하는 것이 어렵고, 코드 베이스에 대한 깊은 이해가 필요합니다
.
▪
유연성 저하: 자원 요청 순서가 고정되므로 프로그램의 유연성이 떨어질 수 있습니다
.
▪
위반 가능성: 프로그래머가 락 프로토콜을 따르지 않으면 여전히 교착 상태가 발생할 수 있습니다
.
6. 교착 상태 회피 (Deadlock Avoidance)
•
개념: 시스템이 교착 상태에 절대 진입하지 않도록 보장하기 위해, 자원 할당 상태를 동적으로 검사하는 접근 방식입니다
. 교착 상태 방지보다 덜 제한적이며 더 높은 동시성을 허용합니다
.
•
요구 사항:
◦
각 프로세스는 자신이 필요로 할 각 자원 유형의 최대 개수를 **미리 선언(a priori declare)**해야 합니다
.
◦
프로세스가 자원을 요청할 때, 시스템은 즉시 할당하는 것이 시스템을 **안전한 상태(safe state)**로 남겨두는지 결정해야 합니다
.
•
안전한 상태 (Safe State):
◦
시스템이 안전한 상태라면 교착 상태는 발생하지 않습니다

.
◦
시스템이 안전한 상태인 경우: 시스템의 모든 스레드 T1, T2, ..., Tn에 대해, 각 Ti가 현재 사용할 수 있는 자원과 Tj (여기서 j < i)가 점유하고 있는 모든 자원을 해제한 후 Ti가 필요한 모든 자원을 충족할 수 있는 실행 순서가 존재하는 경우입니다
. 즉, Tj가 끝나면 Ti가 필요한 자원을 얻고 실행, 반환, 종료할 수 있다는 의미입니다
.
◦
시스템이 **안전하지 않은 상태 (unsafe state)**인 경우: 교착 상태가 발생할 **가능성(possibility)**이 있습니다
. (교착 상태는 아니지만 교착 상태로 이어질 수 있는 상태)
.
◦
회피 알고리즘의 목표: 시스템이 절대 안전하지 않은 상태에 진입하지 않도록 보장하는 것입니다
.
•
회피 알고리즘 종류:
◦
자원 유형당 인스턴스가 하나인 경우: 자원 할당 그래프 알고리즘을 사용합니다
.
◦
자원 유형당 인스턴스가 여러 개인 경우: 뱅커스 알고리즘을 사용합니다
.
6.1. 자원 할당 그래프 방식 (Resource-Allocation Graph Scheme) - 회피 목적
•
개념: 자원 할당 그래프를 사용하여 교착 상태를 회피합니다
.
•
클레임 간선 (Claim edge): 프로세스 Ti가 미래에 자원 Rj를 요청할 수 있음을 나타내는 점선(Ti → Rj)입니다
. 모든 자원은 스레드 시작 전에 미리 클레임되어야 합니다
.
•
알고리즘: 스레드 Ti가 자원 Rj를 요청할 때, 이 요청이 할당 간선으로 변환될 경우 자원 할당 그래프에 사이클이 형성되지 않는 경우에만 요청을 승인합니다
.
[시험 출제 예상] 뱅커스 알고리즘 (Banker's Algorithm)
문제: 뱅커스 알고리즘의 작동 원리, 사용되는 데이터 구조, 그리고 안전성 알고리즘의 단계를 자세히 설명하시오. 뱅커스 알고리즘이 교착 상태를 어떻게 회피하는지 명확히 밝히시오.
해설: 뱅커스 알고리즘은 자원 유형당 여러 개의 인스턴스가 있는 시스템에서 교착 상태를 회피하는 데 사용되는 알고리즘입니다
. 이 알고리즘은 은행이 고객에게 돈을 빌려줄 때 고객의 최대 요구량을 미리 파악하고, 대출 후 은행이 안전한 상태(모든 고객의 대출 상환을 보장할 수 있는 상태)를 유지하는지 확인하는 것과 유사합니다
.
1. 작동 원리:

•
모든 스레드/프로세스는 자신이 평생 동안 필요로 할 각 자원 유형의 **최대 사용량(maximum use)**을 시스템에 미리 알려야 합니다 (a priori claim)
.
•
스레드가 자원을 요청할 때, 시스템은 이 요청을 즉시 할당하는 것이 시스템을 안전한 상태로 유지하는지 여부를 동적으로 검사합니다
.
•
만약 요청을 승인한 후에도 시스템이 안전한 상태를 유지한다면 자원을 할당합니다. 그렇지 않다면, 해당 스레드는 자원이 사용 가능해질 때까지 기다려야 합니다
.
2. 데이터 구조: n을 프로세스의 수, m을 자원 유형의 수라고 할 때, 다음 데이터 구조가 사용됩니다
:
•
Available (가용): 길이가 m인 벡터. Available[j] = k는 자원 유형 Rj의 k개 인스턴스가 현재 사용 가능함을 나타냅니다
.
◦
예시: Available = (3, 3, 2) (A 유형 3개, B 유형 3개, C 유형 2개 사용 가능)
.
•
Max (최대): n x m 행렬. Max[i, j] = k는 프로세스 Pi가 자원 유형 Rj의 인스턴스를 최대 k개까지 요청할 수 있음을 나타냅니다
. (사전 선언된 최대 요구량)
◦
예시: Max[T0, A] = 7 (T0는 A 유형 자원을 최대 7개까지 요청 가능)
.
•
Allocation (할당): n x m 행렬. Allocation[i, j] = k는 프로세스 Pi가 현재 자원 유형 Rj의 k개 인스턴스를 할당받았음을 나타냅s니다
.
◦
예시: Allocation[T0, A] = 0 (T0는 A 유형 자원을 현재 0개 할당받음)
.
•
Need (요구): n x m 행렬. Need[i, j] = k는 프로세스 Pi가 작업을 완료하기 위해 자원 유형 Rj의 인스턴스를 추가로 k개 더 필요로 함을 나타냅니다
.
◦
계산: Need[i, j] = Max[i, j] - Allocation[i, j]
.
◦
예시: Need[T0, A] = 7 - 0 = 7 (T0는 A 유형 자원이 7개 더 필요함)

.
3. 안전성 알고리즘 (Safety Algorithm): 시스템이 현재 안전한 상태인지 여부를 결정합니다
.
1.
두 개의 작업용 벡터 Work (길이 m)와 Finish (길이 n)를 초기화합니다
:
◦
Work = Available
.
◦
Finish[i] = false (모든 i = 0, 1, ..., n-1에 대해)
.
2.
다음 두 조건을 모두 만족하는 i를 찾습니다
:
◦
(a) Finish[i] == false (아직 완료되지 않은 프로세스)
◦
(b) Need[i] <= Work (현재 사용 가능한 자원으로 해당 프로세스의 남은 요구량을 충족시킬 수 있는 경우)
◦
만약 이러한 i가 존재하지 않으면, 4단계로 이동합니다
.
3.
Work와 Finish[i]를 업데이트합니다 (프로세스 Pi가 완료된 것으로 가정)
:
◦
Work = Work + Allocation[i] (Pi가 자원을 해제하여 Work에 추가)
◦
Finish[i] = true
◦
2단계로 돌아갑니다
.
4.
만약 모든 i에 대해 Finish[i] == true라면, 시스템은 안전한 상태입니다
. 그렇지 않다면 시스템은 안전하지 않은 상태입니다.
◦
안전한 상태 시퀀스: 알고리즘이 성공적으로 완료될 때, i를 찾은 순서가 안전한 시퀀스가 됩니다
.
4. 자원 요청 알고리즘 (Resource-Request Algorithm for Process Pi): 프로세스 Pi가 Request_i 벡터만큼 자원을 요청했을 때, 이 요청을 승인할 수 있는지 여부를 결정합니다
.
1.
Request_i <= Need_i인지 확인합니다
.

◦
Request_i가 Need_i보다 크면 오류를 발생시킵니다 (최대 요청량을 초과했기 때문)
.
◦
작거나 같으면 2단계로 이동합니다.
2.
Request_i <= Available인지 확인합니다
.
◦
Request_i가 Available보다 크면, 프로세스 Pi는 자원이 사용 가능해질 때까지 기다려야 합니다
.
◦
작거나 같으면 3단계로 이동합니다.
3.
요청된 자원을 Pi에 할당했다고 가정하고 상태를 수정합니다
:
◦
Available = Available - Request_i;
◦
Allocation[i] = Allocation[i] + Request_i;
◦
Need[i] = Need[i] - Request_i;
4.
수정된 상태에서 안전성 알고리즘을 실행합니다
.
◦
안전한 상태로 판명되면: 자원을 Pi에 할당합니다
.
◦
안전하지 않은 상태로 판명되면: Pi는 기다려야 하며, 이전 자원 할당 상태를 복원합니다
.
5. 교착 상태 회피 메커니즘: 뱅커스 알고리즘은 **사전 정보(최대 요구량)**를 기반으로
, 동적(dynamic)으로 자원 할당의 안전성을 검사하여, 시스템이 안전하지 않은 상태에 진입하는 것을
방지함으로써 결과적으로 교착 상태를 회피합니다. 즉, 자원 요청이 있을 때마다 미래에 교착 상태가 발생할
가능성이 있는지 시뮬레이션하여 안전이 보장될 때만 할당을 허용합니다
.
6. 단점:
•
정보의 한계: 모든 프로세스의 최대 자원 요구량을 미리 알아야 하는데, 이는 실제 시스템에서는 어렵습니다
.
•
프로세스 수 고정: 프로세스의 수가 고정되어야 하며, 동적으로 프로세스가 생성/소멸될 때 유연하지 않습니다.
•
자원 유형 고정: 자원 유형의 수가 고정되어야 합니다.

•
동시성 제한: 보수적인(conservative) 접근 방식으로 인해 불필요하게 자원 할당을 지연시켜 시스템의 동시성(concurrency)을 제한할 수 있습니다
.
•
낮은 활용도: 최대 요구량을 기준으로 자원을 예약하므로 자원 활용률이 낮아질 수 있습니다.
•
복잡성: 알고리즘 자체가 복잡하여 오버헤드가 발생할 수 있습니다.
•
이러한 단점들로 인해 뱅커스 알고리즘은 모든 정보를 완벽하게 알 수 있는 **제한된 환경(예: 임베디드 시스템)**에서만 주로 유용하며, 일반적인 범용 운영체제에서는 널리 사용되지 않습니다
.
7. 교착 상태 탐지 및 회복 (Deadlock Detection and Recovery)
•
개념: 시스템이 교착 상태에 진입하는 것을 허용하고, 주기적으로 교착 상태를 탐지한 후, 탐지된 교착 상태로부터 시스템을 회복시키는 방법입니다
.
•
적용: 교착 상태가 드물게 발생하고, 그 비용이 크지 않은 시스템(예: 일 년에 한 번 OS가 멈추면 재부팅하는 정도)에 실용적입니다
. 데이터베이스 시스템에서 많이 활용됩니다
.
7.1. 교착 상태 탐지 (Deadlock Detection)
[시험 출제 예상] 자원 유형별 교착 상태 탐지 알고리즘
문제:
1.
자원 유형당 인스턴스가 하나인 경우의 교착 상태 탐지 방법을 설명하시오.
2.
자원 유형당 인스턴스가 여러 개인 경우의 교착 상태 탐지 알고리즘의 데이터 구조와 단계를 자세히 설명하시오.
해설:
1. 자원 유형당 인스턴스가 하나인 경우 (Single Instance of Each Resource Type)
•
방법: **대기-포-그래프 (wait-for graph)**를 유지합니다
.
◦
노드: 시스템의 스레드들로 구성됩니다
.
◦
간선: 스레드 Ti가 스레드 Tj가 점유한 자원을 기다리고 있음을 나타내는 간선 Ti → Tj를 추가합니다
. 이는 자원 할당 그래프에서 자원 노드를 제거하여 스레드 간의 직접적인 대기 관계를 표시함으로써 얻을 수 있습니다
.
•
탐지: 주기적으로 알고리즘을 호출하여 대기-포-그래프에서 사이클(cycle)이 존재하는지를 탐색합니다

. 사이클이 발견되면 교착 상태가 존재합니다
.
•
복잡성: 그래프에서 사이클을 탐지하는 알고리즘은 n을 그래프의 정점(스레드) 수라고 할 때, 약 O(n^2)의 연산을 필요로 합니다
.
2. 자원 유형당 인스턴스가 여러 개인 경우 (Several Instances of a Resource Type)
•
사용되는 데이터 구조: (뱅커스 알고리즘과 유사)
◦
Available: 길이가 m인 벡터. 각 자원 유형의 현재 사용 가능한 인스턴스 수를 나타냅니다
.
◦
Allocation: n x m 행렬. 각 프로세스가 현재 할당받은 각 자원 유형의 인스턴스 수를 나타냅니다
.
◦
Request: n x m 행렬. 각 프로세스가 현재 요청하고 있는 각 자원 유형의 추가 인스턴스 수를 나타냅니다
. Request[i][j] = k는 스레드 Ti가 자원 유형 Rj의 k개 인스턴스를 요청하고 있음을 의미합니다
.
•
탐지 알고리즘 (Detection Algorithm):
1.
Work (길이 m)와 Finish (길이 n) 벡터를 초기화합니다:
▪
(a) Work = Available
.
▪
(b) i = 0, 1, ..., n-1에 대해, Allocation[i]가 0이 아니면 (0이 아닌 자원을 할당받은 경우) Finish[i] = false로 설정하고, 그렇지 않으면 Finish[i] = true로 설정합니다
. (즉, 현재 자원을 할당받지 않은 프로세스는 완료된 것으로 간주)
2.
다음 두 조건을 모두 만족하는 인덱스 i를 찾습니다:
▪
(a) Finish[i] == false (아직 완료되지 않은 프로세스)
▪
(b) Request[i] <= Work (해당 프로세스의 현재 요청을 Work로 충족시킬 수 있는 경우)
▪
만약 이러한 i가 존재하지 않으면, 4단계로 이동합니다
.
3.
Work와 Finish[i]를 업데이트합니다:
▪
Work = Work + Allocation[i] (Pi가 자원을 해제하여 Work에 추가)
▪

Finish[i] = true
▪
2단계로 돌아갑니다
.
4.
만약 일부 i에 대해 Finish[i] == false라면 (1 <= i <= n), 시스템은 교착 상태입니다. 또한, Finish[i] == false인 프로세스 Ti는 교착 상태에 빠져 있습니다
.
•
복잡성: 이 알고리즘은 시스템이 교착 상태에 있는지 여부를 탐지하는 데 약 O(m * n^2)의 연산을 필요로 합니다
.
•
탐지 알고리즘 사용 시점:
◦
빈도: 교착 상태가 얼마나 자주 발생할 것으로 예상되는지, 그리고 롤백해야 할 프로세스가 몇 개나 될 것인지(하나의 분리된 사이클당 하나)에 따라 결정됩니다
.
◦
문제점: 알고리즘이 임의적으로 자주 호출되면, 자원 그래프에 많은 사이클이 존재할 수 있어 어떤 교착 상태 프로세스가 "원인"인지 식별하기 어려울 수 있습니다
.
7.2. 교착 상태 회복 (Recovery from Deadlock)
교착 상태가 탐지되면, 시스템은 교착 상태를 해제하고 정상 작동을 재개해야 합니다.
•
프로세스 종료 (Process Termination)
:
◦
모든 교착 상태 프로세스 종료: 교착 상태에 연루된 모든 프로세스를 중단시킵니다. 가장 간단하지만 비효율적인 방법입니다
.
◦
한 번에 하나씩 프로세스 종료: 교착 상태 사이클이 제거될 때까지 교착 상태에 연루된 프로세스를 한 번에 하나씩 중단시킵니다
.
▪
종료할 프로세스 선택 기준: (비용 최소화)
1.
프로세스의 우선순위
.
2.
프로세스가 계산을 수행한 시간 및 완료까지 남은 시간
.
3.

프로세스가 사용한 자원
.
4.
프로세스가 완료하기 위해 필요한 자원
.
5.
종료해야 할 프로세스의 수
.
6.
프로세스가 대화형(interactive)인지 배치(batch)형인지
.
•
자원 선점 (Resource Preemption)
:
◦
교착 상태를 해제하기 위해 프로세스로부터 자원을 강제로 회수하는 방법입니다
.
◦
희생자 선택 (Selecting a victim): 선점할 자원을 가진 프로세스(희생자)를 선택합니다. 이 과정에서도 비용을 최소화하는 기준이 적용됩니다
.
◦
롤백 (Rollback): 선택된 희생자 프로세스를 교착 상태가 발생하기 이전의 안전한 상태(safe state)로 되돌립니다
. 프로세스는 해당 상태부터 다시 시작됩니다
.
◦
기아 상태 (Starvation): 동일한 프로세스가 항상 희생자로 선택되어 무한히 롤백될 수 있습니다
. 이를 방지하기 위해 롤백 횟수를 비용 요소에 포함시킬 수 있습니다
.
7.3. 일반적인 동시성 버그 (Common Concurrency Problems from)
Lu et al.의 연구에 따르면, 대부분의 동시성 버그는 교착 상태가 아닌 버그입니다 (74/105)
.
•
원자성 위반 버그 (Atomicity-Violation Bugs)
◦
설명: 여러 메모리 접근 간에 의도된 직렬성(serializability)이 위반될 때 발생합니다
. 즉, 특정 코드 영역이 원자적으로 실행되어야 하지만, 실제 실행 중 원자성이 보장되지 않는 경우입니다
.
◦
예시: thd->proc_info 필드에 접근하는 두 스레드 (Thread 1: if (thd->proc_info) { fputs(thd->proc_info, ...); }, Thread 2: thd->proc_info = NULL;)
. Thread 1이 proc_info가 non-NULL인지 확인한 후 fputs 호출 전에 인터럽트되어 Thread 2가 proc_info를 NULL로 설정하면, Thread 1 재개 시 NULL 포인터 역참조로 인해 크래시가 발생합니다

.
◦
해결: 공유 변수 접근 주변에 락(mutex)을 추가하여 원자성을 보장합니다 (pthread_mutex_lock/unlock)
.
•
순서 위반 버그 (Order-Violation Bugs)
◦
설명: 두 (그룹의) 메모리 접근 간에 원하는 순서가 뒤바뀌는 경우입니다
. 즉, A는 항상 B보다 먼저 실행되어야 하지만, 실행 중 이 순서가 강제되지 않는 경우입니다
.
◦
예시: Thread 2가 mThread->State에 접근하기 전에 Thread 1이 mThread를 초기화할 것이라고
가정하지만, Thread 2가 먼저 실행되면 mThread가 초기화되지 않아 NULL 포인터 역참조로 크래시가 발생할 수 있습니다
.
◦
해결: 조건 변수(condition variables)를 사용하여 실행 순서를 강제합니다
. 락과 상태 변수(예: mtInit)를 함께 사용하여 조건이 충족될 때까지 대기하거나, 충족되면 계속 진행하도록 합니다 (pthread_cond_wait/signal)
.
8. 메모리 관리 (Main Memory)
8.1. 배경 (Background)
•
프로그램이 실행되려면 디스크에서 메모리로 가져와 프로세스 내에 배치되어야 합니다
.
•
주 메모리(Main Memory)와 레지스터(Registers)는 CPU가 직접 접근할 수 있는 유일한 저장소입니다
.
•
레지스터 접근은 1 CPU 클록(또는 그 이하)에 이루어지지만, 주 메모리는 많은 클록 사이클을 소모하여 CPU를 대기(stall)시킬 수 있습니다
.
•
캐시(Cache)는 주 메모리와 CPU 레지스터 사이에 위치하여 속도 차이를 완화합니다
.
•
정확한 작업을 위해 메모리 보호가 필수적입니다
.
8.2. 메모리 보호 (Protection)
•
프로세스가 자신의 주소 공간 내의 주소만 접근할 수 있도록 보장해야 합니다
.
•

베이스 레지스터(Base Register)와 리미트 레지스터(Limit Register) 한 쌍을 사용하여 프로세스의 논리적 주소 공간을 정의함으로써 메모리 보호를 제공할 수 있습니다
.
◦
베이스 레지스터: 프로세스의 가장 작은 물리적 주소 값을 포함합니다
.
◦
리미트 레지스터: 프로세스의 논리적 주소 공간의 범위(크기)를 포함합니다. 생성된 각 논리적 주소는 리미트 레지스터 값보다 작아야 합니다
.
•
CPU는 사용자 모드에서 생성된 모든 메모리 접근이 해당 사용자의 베이스와 리미트 레지스터 범위 내에 있는지 확인해야 합니다
.
•
베이스 및 리미트 레지스터를 로드하는 명령어는 특권 명령어(privileged instructions)입니다 (사용자 프로세스가 함부로 변경할 수 없음)
.
8.3. 주소 바인딩 (Address Binding)
•
개념: 프로그램의 주소가 메모리 주소에 연결되는 과정입니다
.
•
프로그램 수명 주기별 주소 표현:
◦
소스 코드 주소: 일반적으로 기호적(symbolic)입니다 (예: count)
.
◦
컴파일된 코드 주소: 재배치 가능(relocatable) 주소로 바인딩됩니다 (예: "이 모듈 시작부터 14바이트")
.
◦
링커(Linker) 또는 로더(Loader): 재배치 가능 주소를 절대(absolute) 주소로 바인딩합니다 (예: 74014)
.
•
바인딩 시점: 명령어와 데이터의 메모리 주소 바인딩은 세 가지 다른 단계에서 발생할 수 있습니다
:
◦
컴파일 시간 (Compile time): 메모리 위치가 미리 알려진 경우 절대 코드가 생성됩니다. 시작 위치가 변경되면 코드를 다시 컴파일해야 합니다
.
◦
로드 시간 (Load time): 컴파일 시간에 메모리 위치가 알려지지 않은 경우 재배치 가능 코드를 생성해야 합니다
.
◦

실행 시간 (Execution time): 프로세스가 실행 중에 메모리 세그먼트 간에 이동할 수 있는 경우 바인딩이 런타임까지 지연됩니다
. 주소 매핑을 위한 하드웨어 지원(예: 베이스 및 리미트 레지스터)이 필요합니다
.
•
논리적 vs. 물리적 주소 공간 (Logical vs. Physical Address Space):
◦
논리적 주소 (Logical address): CPU에 의해 생성되는 주소이며, 가상 주소(virtual address)라고도 불립니다
. 프로세스가 인식하는 주소입니다.
◦
물리적 주소 (Physical address): 메모리 장치에 의해 보이는 실제 메모리 주소입니다
.
◦
관계: 컴파일 시간 및 로드 시간 주소 바인딩 방식에서는 논리적 주소와 물리적 주소가 동일합니다
. 실행 시간 주소 바인딩 방식에서는 논리적 주소와 물리적 주소가 다릅니다
.
◦
논리적 주소 공간: 프로그램이 생성하는 모든 논리적 주소의 집합
.
◦
물리적 주소 공간: 프로그램이 생성하는 모든 물리적 주소의 집합
.
8.4. 메모리 관리 장치 (Memory-Management Unit, MMU)
•
개념: 런타임에 가상 주소를 물리적 주소로 매핑하는 하드웨어 장치입니다
.
•
동작: 베이스 레지스터(재배치 레지스터라고도 함)의 값이 사용자 프로세스에 의해 생성된 모든 주소에 메모리로 전송될 때 더해집니다
. 사용자 프로그램은 논리적 주소를 다루며 실제 물리적 주소는 보지 못합니다. 실행 시간 바인딩은 메모리 위치를 참조할 때 발생합니다
.
8.5. 동적 적재 및 동적 연결 (Dynamic Loading & Dynamic Linking)
8.5.1. 동적 적재 (Dynamic Loading)
•
개념: 프로그램 전체가 메모리에 있을 필요 없이, 루틴이 호출될 때까지 로드되지 않는 방식입니다
.
•
특징:
◦
메모리 공간 활용률을 향상시킵니다 (사용되지 않는 루틴은 로드되지 않음)
.
◦
모든 루틴은 디스크에 재배치 가능한 로드 형식으로 유지됩니다

.
◦
자주 발생하지 않는 경우를 처리하는 데 많은 코드가 필요할 때 유용합니다
.
◦
운영체제의 특별한 지원이 필요하지 않으며, 프로그램 설계에 의해 구현될 수 있습니다 (OS는 라이브러리 제공으로 도움)
.
8.5.2. 동적 연결 (Dynamic Linking)
•
개념: 링킹(linking) 과정이 실행 시간까지 연기되는 방식입니다
.
•
종류:
◦
정적 연결 (Static linking): 시스템 라이브러리와 프로그램 코드가 로더(loader)에 의해 바이너리 프로그램 이미지로 결합됩니다
.
◦
동적 연결 (Dynamic linking): 링킹이 실행 시간까지 지연됩니다
.
•
스텁 (Stub): 적절한 메모리 상주 라이브러리 루틴을 찾는 데 사용되는 작은 코드 조각입니다
. 스텁은 자신을 루틴의 주소로 교체하고 해당 루틴을 실행합니다
.
•
동작: 운영체제는 루틴이 프로세스의 메모리 주소 공간에 있는지 확인하고, 없으면 주소 공간에 추가합니다
.
•
활용: 특히 라이브러리에 유용하며, 공유 라이브러리(shared libraries)라고도 알려져 있습니다
. 시스템 라이브러리 패치에 적용 가능하며, 버전 관리(versioning)가 필요할 수 있습니다
.
8.6. 연속 메모리 할당 (Contiguous Allocation)
•
개념: 주 메모리가 운영체제와 사용자 프로세스 모두를 지원해야 할 때 사용되는 초기 메모리 할당 방법 중 하나입니다
. 각 프로세스가 메모리의 단일 연속 섹션에 포함됩니다
.
•
구성: 주 메모리는 일반적으로 두 개의 파티션으로 나뉩니다
:
◦
상주 운영체제 (Resident operating system): 일반적으로 인터럽트 벡터와 함께 낮은 메모리 영역에 유지됩니다

.
◦
사용자 프로세스 (User processes): 높은 메모리 영역에 유지됩니다
.
•
보호: 재배치 레지스터(relocation registers)가 사용자 프로세스를 서로, 그리고 운영체제 코드 및 데이터로부터 보호하는 데 사용됩니다
. MMU는 논리적 주소를 동적으로 매핑합니다
.
•
가변 파티션 (Variable Partition):
◦
다중 파티션 할당은 다중 프로그래밍의 정도가 파티션 수에 의해 제한됩니다
.
◦
효율성을 위해 가변 파티션 크기가 사용됩니다 (주어진 프로세스의 필요에 따라 크기 조절)
.
◦
홀 (Hole): 사용 가능한 메모리 블록으로, 다양한 크기의 홀이 메모리 전체에 흩어져 있습니다
.
◦
프로세스가 도착하면, 프로세스를 수용할 만큼 충분히 큰 홀에서 메모리를 할당받습니다
.
◦
프로세스가 종료되면 파티션을 해제하고, 인접한 빈 파티션은 병합됩니다
.
◦
운영체제는 할당된 파티션과 빈 파티션(홀)에 대한 정보를 유지합니다
.
8.6.1. 동적 저장 공간 할당 문제 (Dynamic Storage-Allocation Problem)
•
주어진 크기 n의 요청을 빈 홀 목록에서 어떻게 만족시킬 것인가에 대한 문제입니다
.
•
전략:
◦
최초 적합 (First-fit): 충분히 큰 첫 번째 홀을 할당합니다
.
◦
최적 적합 (Best-fit): 충분히 큰 가장 작은 홀을 할당합니다. 전체 목록을 검색해야 할 수 있습니다 (크기 순서로 정렬되지 않은 경우)
. 가장 작은 남은 홀을 생성합니다
.
◦

최악 적합 (Worst-fit): 가장 큰 홀을 할당합니다. 마찬가지로 전체 목록을 검색해야 합니다
. 가장 큰 남은 홀을 생성합니다
.
•
성능: First-fit과 Best-fit은 Worst-fit보다 속도와 저장 공간 활용 측면에서 더 좋습니다
.
8.6.2. 단편화 (Fragmentation)
•
외부 단편화 (External Fragmentation): 총 메모리 공간은 요청을 만족시킬 만큼 존재하지만, 그 공간이 연속적이지 않아 할당할 수 없는 상황입니다
.
•
내부 단편화 (Internal Fragmentation): 할당된 메모리가 요청된 메모리보다 약간 더 클 수 있으며, 이 크기 차이는 파티션 내부의 메모리이지만 사용되지 않는 부분입니다
.
•
First-fit 분석: 할당된 N개 블록 중 0.5N 블록이 단편화로 손실되는 경향이 있습니다 (50% 규칙)
.
•
외부 단편화 감소: **압축(compaction)**을 통해 모든 빈 메모리를 하나의 큰 블록으로 모아 외부 단편화를 줄일 수 있습니다
.
◦
압축은 재배치가 동적이고 실행 시간에 이루어지는 경우에만 가능합니다
.
◦
I/O 문제: I/O에 관여된 작업은 메모리에 래치(latch)되어야 하므로, I/O는 OS 버퍼로만 수행해야 합니다
.
◦
백업 저장소(backing store)도 동일한 단편화 문제를 가집니다
.
9. CPU 스케줄링 (CPU Scheduling)
9.1. 기본 개념 (Basic Concepts)
•
프로세스 실행: CPU 실행(CPU burst)과 I/O 대기(I/O wait)의 사이클로 구성됩니다
.
•
CPU-I/O 버스트 사이클: CPU 버스트 후 I/O 버스트가 오는 형태를 가집니다
.
•
CPU 버스트 분포: 짧은 버스트가 많고 긴 버스트가 적은 분포를 보입니다
.
•

CPU 스케줄러 (CPU Scheduler): 준비 큐(ready queue)에 있는 프로세스들 중에서 하나를 선택하여 CPU 코어를 할당합니다
. 큐는 다양한 방식으로 정렬될 수 있습니다
.
•
CPU 스케줄링 결정 시점: 프로세스가 다음 상황 중 하나일 때 스케줄링 결정이 이루어질 수 있습니다
:
1.
실행(running) 상태에서 대기(waiting) 상태로 전환될 때
2.
실행(running) 상태에서 준비(ready) 상태로 전환될 때
3.
대기(waiting) 상태에서 준비(ready) 상태로 전환될 때
4.
종료(terminates)될 때
•
선점 vs. 비선점 스케줄링:
◦
비선점(Nonpreemptive) 스케줄링: 스케줄링 결정 1번과 4번 (협력적)
. 일단 CPU를 할당받으면 자발적으로 해제하기 전까지 CPU를 놓지 않습니다.
◦
선점(Preemptive) 스케줄링: 그 외의 모든 스케줄링 (2번, 3번)
. 공유 데이터 접근(레이스 컨디션), 커널 모드에서의 선점, 중요한 OS 활동 중 인터럽트 발생 등을 고려해야 합니다
.
9.1.1. 스케줄러 종류 (Scheduler Types)
•
장기 스케줄러 (Long-term Scheduler / Job scheduler):
◦
디스크에서 메모리로 작업을 가져와 처리할 순서를 결정합니다 (준비 큐)
.
◦
프로세스 수를 제어합니다 (다중 프로그래밍의 정도를 제어)
.
•
단기 스케줄러 (Short-term Scheduler / CPU scheduler):
◦
메모리에 적재된 프로세스 중 프로세서를 할당하여 실행 상태가 되도록 결정하는 프로세스입니다
.
◦
프로세스에 CPU를 할당하는 역할을 합니다
.
•
중기 스케줄러 (Medium-term Scheduler / Swapper):

◦
스왑-인(Swap-in)과 스왑-아웃(Swap-out)을 결정합니다
.
◦
프로세스 수를 제어합니다
.
9.1.2. 디스패처 (Dispatcher)
•
개념: 단기 스케줄러가 선택한 프로세스에 CPU 제어권을 넘겨주는 모듈입니다
.
•
역할:
◦
컨텍스트 스위칭 (Context Switching): 실행 중인 프로세스의 상태를 저장하고, 새로 실행될 프로세스의 상태를 로드합니다
.
◦
사용자 모드로 전환
.
◦
사용자 프로그램의 적절한 위치로 점프하여 해당 프로그램을 다시 시작합니다
.
•
디스패치 지연 (Dispatch Latency): 디스패처가 하나의 프로세스를 중지시키고 다른 프로세스를 실행시키는 데 걸리는 시간입니다
.
9.2. 스케줄링 기준 (Scheduling Criteria)
스케줄링 알고리즘의 성능을 평가하는 데 사용되는 기준들입니다
.
•
CPU 활용률 (CPU utilization): CPU를 가능한 한 바쁘게 유지합니다 (최대화 목표)
.
•
처리량 (Throughput): 단위 시간당 완료되는 프로세스의 수입니다 (최대화 목표)
.
•
반환 시간 (Turnaround time): 특정 프로세스를 실행하는 데 걸리는 총 시간입니다 (최소화 목표)
. 준비 큐에서 대기한 시간, CPU에서 실행된 시간, I/O를 수행한 시간 등 모든 시간을 포함합니다.
•
대기 시간 (Waiting time): 프로세스가 준비 큐에서 대기한 총 시간입니다 (최소화 목표)
.
•
응답 시간 (Response time): 요청이 제출된 시점부터 첫 번째 응답이 생성될 때까지 걸리는 시간입니다 (출력 시간 아님, 시분할 환경에서 중요) (최소화 목표)

.
9.3. 스케줄링 알고리즘 (Scheduling Algorithms)
9.3.1. 선입선출 (First-Come, First-Served, FCFS) 스케줄링
•
개념: 준비 큐에 먼저 도착한 프로세스가 CPU를 먼저 할당받는 방식입니다
.
•
특징:
◦
구현이 가장 간단합니다.
◦
호위 효과 (Convoy effect): 하나의 CPU-bound 프로세스(긴 버스트 시간)가 CPU를 점유하고 있을 때, 뒤에 있는 여러 I/O-bound 프로세스(짧은 버스트 시간)들이 모두 대기해야 하는 현상입니다
. 이로 인해 평균 대기 시간이 크게 증가할 수 있습니다
.
•
예시:
◦
P1(24ms), P2(3ms), P3(3ms) 순으로 도착 시 평균 대기 시간 = (0 + 24 + 27) / 3 = 17ms
.
◦
P2(3ms), P3(3ms), P1(24ms) 순으로 도착 시 평균 대기 시간 = (0 + 3 + 6) / 3 = 3ms
. (도착 순서에 따라 성능이 크게 달라짐)
.
9.3.2. 최단 작업 우선 (Shortest-Job-First, SJF) 스케줄링
•
개념: 다음 CPU 버스트 길이가 가장 짧은 프로세스에게 CPU를 할당합니다
.
•
특징:
◦
최적 (Optimal): 주어진 프로세스 집합에 대해 최소 평균 대기 시간을 제공합니다
.
◦
문제점: 다음 CPU 버스트의 길이를 미리 아는 것이 어렵습니다
.
◦
추정 (Estimation): 과거 CPU 버스트 길이를 사용하여 지수 평균(exponential averaging) 방식으로 다음 버스트 길이를 추정할 수 있습니다
.
▪
$\tau(n+1) = \alpha * t(n) + (1 - \alpha) * \tau(n)$
•
t(n): n번째 실제 CPU 버스트 길이

•
τ(n): n번째 CPU 버스트에 대한 예측 값
•
α (알파): 0과 1 사이의 가중치 (일반적으로 1/2 사용)
.
▪
α=0일 경우: τ(n+1) = τ(n) (최근 이력 무시)
.
▪
α=1일 경우: τ(n+1) = t(n) (마지막 CPU 버스트만 반영)
.
•
선점 SJF (Preemptive SJF):
◦
최단 잔여 시간 우선 (Shortest-Remaining-Time-First, SRTF)이라고도 불립니다
.
◦
새로운 프로세스가 준비 큐에 도착했을 때, 현재 실행 중인 프로세스의 잔여 시간과 새로 도착한 프로세스의 버스트 시간을 비교하여 더 짧은 프로세스에게 CPU를 할당합니다
.
◦
예시: P1(0, 8), P2(1, 4), P3(2, 9), P4(3, 5) [도착시간, 버스트시간] 평균 대기 시간 = [(10-1) + (1-1) + (17-2) + (5-3)] / 4 = 26 / 4 = 6.5ms
.
9.3.3. 라운드 로빈 (Round Robin, RR) 스케줄링
•
개념: 각 프로세스는 작은 단위의 CPU 시간(시간 할당량 q, time quantum)을 부여받습니다 (일반적으로 10~100밀리초)
. 시간 할당량이 지나면 프로세스는 선점되어 준비 큐의 끝에 추가됩니다
.
•
특징:
◦
n개의 프로세스가 준비 큐에 있고 시간 할당량이 q일 때, 각 프로세스는 CPU 시간의 1/n을 최대 q시간 단위로 얻습니다
.
◦
어떤 프로세스도 (n-1)q 시간 이상 기다리지 않습니다
.
◦
시간 할당량 q와 성능:
▪
q가 매우 클 경우: FCFS와 유사하게 동작합니다
.

▪
q가 매우 작을 경우: 컨텍스트 스위치 오버헤드가 너무 높아져 CPU 활용률이 떨어집니다
. q는 컨텍스트 스위치 시간보다 커야 합니다 (예: q가 10~100ms일 때 컨텍스트 스위치 시간은 10ms 미만)
.
◦
평균 반환 시간: SJF보다 일반적으로 높지만, 응답 시간은 더 좋습니다
.
◦
CPU 버스트의 약 80%가 q보다 짧아야 효율적입니다
.
•
예시: P1(24), P2(3), P3(3)에 대해 시간 할당량 q=4ms 적용
. 평균 대기 시간 = [(10-4) + (4) + (7)] / 3 = 17 / 3 = 5.66ms
.
9.3.4. 우선순위 (Priority) 스케줄링
•
개념: 각 프로세스에 우선순위 번호(정수)가 할당됩니다
. CPU는 가장 높은 우선순위를 가진 프로세스에게 할당됩니다 (가장 작은 정수가 가장 높은 우선순위를 의미, 3비트 또는 14비트)
.
•
특징:
◦
선점형 또는 비선점형으로 구현될 수 있습니다
.
◦
SJF는 다음 CPU 버스트 시간의 역수를 우선순위로 사용하는 우선순위 스케줄링의 한 형태입니다
.
◦
동일한 우선순위를 가진 프로세스들은 FCFS 순서로 스케줄링됩니다
.
•
문제점: 기아 상태 (Starvation): 낮은 우선순위 프로세스는 영원히 실행되지 못할 수 있습니다
.
•
해결책: 에이징 (Aging): 시간이 지남에 따라 프로세스의 우선순위를 점진적으로 증가시키는 방법으로 기아 상태를 해결합니다
.
•
우선순위 역전 (Priority Inversion): 낮은 우선순위 프로세스가 높은 우선순위 프로세스가 필요로 하는 락을 점유하고 있는 스케줄링 문제입니다
. 우선순위 상속 프로토콜(priority-inheritance protocol)로 해결됩니다
.
•

우선순위 스케줄링 + 라운드 로빈: 가장 높은 우선순위 프로세스를 실행하고, 동일한 우선순위 프로세스들은 라운드 로빈 방식으로 실행합니다

.

9.3.5. 다단계 큐 (Multilevel Queue) 스케줄링

- 개념: 우선순위 스케줄링과 함께, 각 우선순위별로 별도의 준비 큐를 가집니다

.

- 특징:
  - 가장 높은 우선순위 큐에 있는 프로세스를 스케줄합니다

.

  - 각 작업은 메모리 크기나 프로세스 유형에 따라 다른 묶음으로 분류되어 특정 큐에 지정될 수 있습니다

.

  - 각 큐는 자체적인 스케줄링 알고리즘을 가질 수 있습니다

.

9.3.6. 다단계 피드백 큐 (Multilevel Feedback Queue) 스케줄링

- 개념: 프로세스가 여러 큐 사이를 이동할 수 있도록 허용하여, 에이징과 같은 동적인 우선순위 조정을 구현할 수 있는 스케줄러입니다

.

- 정의 매개변수:
  - 큐의 개수

.

  - 각 큐의 스케줄링 알고리즘

.

  - 프로세스를 상위 큐로 승격(upgrade)시키는 방법

.

  - 프로세스를 하위 큐로 강등(demote)시키는 방법

.

  - 프로세스가 서비스를 필요로 할 때 진입할 큐를 결정하는 방법

.

- 예시: 세 개의 큐 (Q0, Q1, Q2)

.

  -

Q0: 시간 할당량 8ms의 RR

.

◦

Q1: 시간 할당량 16ms의 RR

.

◦

Q2: FCFS

.

◦

스케줄링: 새로운 작업은 Q0(FCFS로 서비스됨)에 진입하여 8ms의 CPU 시간을 받습니다
. 8ms 안에 완료되지 않으면 Q1으로 이동합니다. Q1에서 FCFS로 서비스되며 추가 16ms를 받습니다. 그래도 완료되지 않으면 선점되어 Q2로 이동합니다

.

메모리 관리의 배경 및 문제점

•

메모리 관리 장치 (MMU): 운영 체제 16번째 슬라이드에서 다룬 내용으로, 메모리 관리를 하드웨어적으로 지원하는 장치입니다

.

◦

단순한 MMU는 베이스 레지스터(Base Register)와 리밋 레지스터(Limit Register)를 포함합니다

.

◦

베이스 레지스터: 한 프로세스의 물리적 메모리 시작 위치를 지정합니다

.

◦

리밋 레지스터: 프로세스가 할당받을 수 있는 메모리 공간의 크기를 제한합니다

.

◦

이 레지스터들은 다른 프로세스와의 메모리 영역 구분을 통해 서로 침범할 수 없도록 하는 안전장치 역할을 합니다

.

•

프로세스의 주소 공간: 일반적으로 0번지부터 실행 파일의 크기(파일 사이즈)만큼 상대적인 번지로 커집니다

.

•

연속적인 메모리 공간 할당: 프로세스는 물리적 메모리 할당 시, 진입점(시작 위치)으로부터 연속적인 메모리 공간을 할당받습니다

.

•

외부 단편화 (External Fragmentation):

◦

각 프로세스의 크기가 다르기 때문에 메모리 할당 시 가변적으로 이루어집니다

.

◦

시간이 지나면서 프로세스가 메모리를 반납하면 '홀(Hole)'이라는 빈 공간이 생깁니다
.
◦
이 홀에 새로운 프로세스가 들어갈 수 있지만, 홀의 크기가 너무 작아 다른 프로세스가 사용하기 어려울 경우 메모리 단편화가 발생합니다
.
◦
이는 프로세스 외부에 생기므로 '외부 단편화'라고 부르며, 메모리 누수 현상을 초래하여 효율적인 메모리 관리를 어렵게 만듭니다
.
페이징 (Paging) 시스템
•
외부 단편화 해결책: 연속적인 메모리 공간 할당이 아닌, 고정된 크기의 블록으로 메모리를 나누어 관리하는 페이징 기법이 외부 단편화를 줄이는 해결책입니다
.
•
프레임 (Frame): 메인 메모리를 고정된 크기의 작은 블록으로 나눈 단위입니다. 각 프레임에는 번호가 부여됩니다
.
•
페이지 (Page): 프로세스(프로그램)도 메인 메모리의 프레임과 동일한 고정된 크기의 블록으로 나누어지며, 이를 페이지라고 합니다. 각 페이지에는 번호가 부여됩니다
.
◦
프로세스의 모든 코드가 즉시 필요하지 않을 수 있으므로 페이지 단위로 나누어 관리합니다
.
•
CPU의 착각: CPU는 데이터나 명령들이 메모리에 연속적으로 정렬되어 있는 것처럼 인식하지만, 실제로는 작은 블록 조각들을 순서에 맞춰 조립하여 접근하는 방식입니다
.
•
페이지 테이블 (Page Table):
◦
프로세스의 페이지가 어떤 프레임에 위치해 있는지를 나타내는 테이블입니다
.
◦
페이지 번호로 인덱스되어 있으며, 해당 페이지가 매핑된 프레임 번호를 저장합니다
.
◦
페이지가 현재 메모리에 없을 경우, 하드 디스크(백킹 스토리지)의 원본 파일 위치를 가리킬 수 있습니다
. 필요 시 하드 디스크에서 데이터를 읽어와 비어있는 프레임에 할당한 후 페이지 테이블을 업데이트합니다
.
◦
페이지 테이블 엔트리(PTE)에는 프레임 번호 외에 유효/무효 비트(Valid/Invalid bit), 더티 비트(Dirty bit), 참조 비트(Reference bit) 등 다양한 정보가 포함됩니다
.
▪
유효/무효 비트 (Valid/Invalid bit): 해당 페이지가 현재 물리적 메모리(프레임)에 유효하게 로드되어 있는지 여부를 나타냅니다. '1'이면 유효, '0'이면 무효로, 무효인 공간에 접근 시 트랩을 발생시켜 프로그램 수행을 정지시킬 수 있습니다

.
- 더티 비트 (Dirty bit): 해당 페이지의 내용이 수행 도중 변경되었는지(쓰기 작업이 있었는지)를 나타냅니다. 변경된 페이지는 메모리에서 쫓겨날 때(Eviction) 원본 백킹 스토리지에 업데이트되어야 합니다.
- 참조 비트 (Reference bit): 페이지 교체 알고리즘에서 사용되며, 페이지가 최근에 참조되었는지 여부를 표시하여 덜 사용된 페이지를 퇴출시키는 데 활용됩니다.

- 프레임 테이블 (Frame Table): 물리적 메모리의 프레임을 관리하며, 각 프레임 번호가 어떤 프로세스에 할당되었는지, 또는 비어있는지 등의 속성을 나타냅니다. 새로운 프로세스가 메모리에 매핑될 때 이 테이블을 참조하여 비어 있는 프레임을 순차적으로 할당합니다.

페이징의 장단점 및 관련 개념

- 내부 단편화 (Internal Fragmentation):
  - 페이징 시스템에서는 외부 단편화는 존재하지 않지만, 내부 단편화가 발생할 수 있습니다.
  - 페이지 및 프레임 크기는 일반적으로 2의 거듭제곱(예: 512B ~ 16MB)으로 정의됩니다.
  - 프로세스 크기가 페이지 크기의 배수가 아닐 경우, 마지막 페이지의 일부 공간이 사용되지 않고 남게 됩니다. 이 부분이 내부 단편화가 됩니다.
  - 예시: 72,766 바이트 프로세스가 2048 바이트(2KB) 프레임을 사용할 때, 35개의 페이지를 채우고 1086 바이트가 남습니다. 이 1086 바이트를 위해 하나의 2048 바이트 프레임이 할당되고, 나머지 962 바이트(2048 - 1086)가 내부 단편화로 남게 됩니다.
  - 최악의 경우, 프레임 크기 - 1 바이트만큼의 단편화가 발생할 수 있습니다.
  - 평균적으로 프레임 크기의 절반 정도가 내부 단편화로 발생하지만, 이는 일반적으로 감수할 만한 수준으로 간주됩니다.
  - 프레임 크기를 작게 하면 내부 단편화는 줄어들지만, 페이지 테이블의 크기가 커지는 문제가 발생합니다.
- 가상 주소(Logical Address)와 물리 주소(Physical Address) 변환:
  - CPU가 사용하는 주소는 프로세스가 가지는 논리적인(가상) 주소입니다.
  - 이 가상 주소는 페이지 번호(Page Number, P)와 페이지 오프셋(Page Offset, D 또는 Displacement)으로 구성됩니다

.
◦
페이지 테이블은 이 페이지 번호를 통해 해당 페이지가 매핑된 물리적 프레임 번호(Frame Number, F)를 찾아냅니다
.
◦
물리 주소는 프레임 번호(F)와 페이지 오프셋(D)을 결합하여 생성됩니다
. 페이지 오프셋은 가상 주소와 물리 주소 모두에서 동일하게 사용됩니다
.
◦
예시: 32비트 주소 체계에서 4KB 페이지(2^12 바이트)를 사용하면, 페이지 오프셋은 12비트, 페이지 번호는 20비트(32-12)가 됩니다
.
페이지 테이블 크기 및 성능 최적화
•
페이지 테이블의 크기 문제:
◦
32비트 주소 체계에서 4KB 프레임을 사용하는 경우, 페이지 수는 2^20개(약 100만 개)가 됩니다
.
◦
각 페이지 테이블 엔트리가 4바이트라고 가정하면, 하나의 프로세스에 대한 페이지 테이블만으로 4MB (2^20 * 4바이트)의 메모리가 필요합니다
.
◦
프로세스가 100개 이상일 경우 페이지 테이블만으로도 수백 MB에서 기가바이트 단위의 메모리가 소모될 수 있습니다
.
◦
특히 64비트 시스템으로 확장될 경우, 단순한 페이지 테이블 구조로는 감당하기 어려워집니다
.
•
메모리 접근 오버헤드:
◦
페이지 테이블은 일반적으로 메인 메모리에 저장됩니다
.
◦
어떤 데이터에 접근하기 위해서는 (1) 페이지 테이블에 접근하여 프레임 위치를 확인하고, (2) 해당 프레임에 접근하여 실제 데이터를 읽어야 하므로, 하나의 데이터 접근에 두 번의 메모리 접근이 필요하게 됩니다
. 이는 시스템 성능 저하로 이어집니다
.
•
변환 색인 버퍼 (TLB: Translation Lookaside Buffer):
◦
메모리 접근 오버헤드를 줄이기 위한 캐시 메모리입니다
. '연관 메모리(Associative Memory)' 또는 '연관 레지스터(Associative Register)'라고도 합니다
.
◦
TLB는 CPU와 거의 같은 속도로 동작하므로, 페이지 테이블에 대한 캐시 역할을 하여 빠른 접근을 가능하게 합니다
.

◦
TLB Hit: 원하는 페이지 정보가 TLB에 있을 경우, 페이지 테이블에 직접 접근하지 않고 TLB에서 바로 프레임 번호를 얻어 물리적 메모리에 접근합니다. 이는 매우 빠른 데이터 접근을 가능하게 합니다
.
◦
TLB Miss: 원하는 페이지 정보가 TLB에 없을 경우, 메인 메모리의 페이지 테이블에 접근하여 정보를 가져온 후 TLB에 저장합니다. 이 경우 두 번의 메모리 접근이 필요합니다
.
◦
적중률(Hit Ratio): TLB의 효율성을 나타냅니다. 적중률이 높을수록 성능이 향상됩니다
. 예를 들어, 메모리 접근 시간이 10ns일 때, TLB 적중률이 80%이면 평균 접근 시간은 12ns가 됩니다 (0.8 * 10ns + 0.2 * 20ns)
.
•
다단계 페이지 테이블 (Hierarchical Page Table):
◦
단일 페이지 테이블의 크기 문제를 해결하기 위한 구조입니다
.
◦
페이지 번호 영역을 여러 부분으로 나누어 페이지 테이블을 계층적으로 구성합니다
.
◦
예시: 32비트 주소 체계에서 페이지 번호를 10비트, 10비트, 페이지 오프셋을 12비트로 나눌 경우, 아우터 페이지 테이블(Outer Page Table)과 이너 페이지 테이블(Inner Page Table)로 구성됩니다
.
◦
이를 통해 페이지 테이블의 총 용량을 크게 줄일 수 있습니다 (예: 4MB에서 80KB 수준으로 감소)
.
◦
그러나 64비트와 같은 더 큰 주소 공간에서는 여전히 문제가 될 수 있습니다
.
•
다른 페이지 테이블 구조: 다단계 페이지 테이블 외에도 해시 페이지 테이블(Hashed Page Table)이나 역 페이지 테이블(Inverted Page Table)과 같은 방법들이 있습니다
.

1. 교착상태(Deadlock) 개요 및 특성
영상에서는 운영체제(Operating System)에서 발생하는 교착상태(Deadlock)에 대해 설명하며,
멀티프로그래밍 환경에서 여러 프로세스나 스레드가 한정된 컴퓨터 시스템 자원을 공유할 때 발생하는 문제라고 언급합니다
. 특정 스레드가 어떤 자원을 점유하고 다른 스레드가 점유한 또 다른 자원을 기다릴 때, 서로가 영원히 작업을 완료할 수 없는 상황이 발생하는데, 이를 교착상태라고 정의합니다. 이 개념은 6장에서 다룬 '활성 실패 (Liveness failure)'의 일종으로 언급되며, 철학자들의 식사 문제(Dining Philosophers Problem)를 예시로 들어 설명됩니다
.
시스템 모델, 교착상태 특성, 교착상태 처리 방법 등이 논의의 주요 주제로 제시됩니다
.
1.1. 시스템 모델 (System Model)
시스템은 여러 자원들로 구성됩니다
. 자원들은 CPU 사이클, 메모리 공간, I/O 장치 등 다양한 유형(R1, R2, ..., Rm)으로 나뉘며, 각 자원 유형 Ri는 Wi개의 인스턴스(instances)를 가질 수 있습니다. 각 프로세스 또는 스레드는 자원을 다음과 같은 방식으로 활용합니다
:
•
요청(request): 필요한 자원을 요청합니다
.
•
사용(use): 요청한 자원을 할당받아 사용합니다
.
•
해제(release): 자원 사용을 완료하면 자원을 반납합니다
.
1.2. 교착상태 특성 (Deadlock Characterization)
교착상태는 다음 네 가지 필요 조건이 모두 동시에 충족될 때 발생할 수 있습니다
:
1.
상호 배제(Mutual Exclusion):
◦
오직 하나의 스레드(또는 프로세스)만이 한 번에 자원을 사용할 수 있습니다
.
◦
즉, 최소한 하나의 자원이 비공유 모드(non-sharable mode)로 점유되어야 합니다. 뮤텍스 락(mutex lock)이나 세마포(semaphore)와 같은 자원들이 이에 해당합니다
.
2.
점유 및 대기(Hold and Wait):
◦
자원을 최소한 하나 이상 점유하고 있는 스레드가 다른 스레드에 의해 점유된 추가적인 자원을 얻기 위해 대기하는 상태를 의미합니다
.
◦
영상에서 세마포 S1과 S2 예시를 통해 T1이 S1을 점유한 채 S2를 기다리고, T2가 S2를 점유한 채 S1을 기다리는 상황이 설명됩니다
.
3.
비선점(No Preemption):

◦
자원은 해당 스레드가 작업을 완료한 후 자발적으로만 해제될 수 있으며, 강제로 선점(preempted)될 수 없습니다
.
◦
스레드가 자발적으로 자원을 내려놓지 않으면 다른 스레드가 그 자원을 강제로 뺏을 수 없습니다
.
4.
순환 대기(Circular Wait):
◦
대기하는 스레드들의 집합 {T0, T1, ..., Tn}이 존재하여, T0는 T1이 점유한 자원을 기다리고, T1은 T2가 점유한 자원을 기다리며, ..., Tn-1은 Tn이 점유한 자원을 기다리고, Tn은 다시 T0가 점유한 자원을 기다리는 순환 고리가 형성되는 상태를 의미합니다
.
1.3. 라이브락 (Livelock)
라이브락은 데드락과 유사하지만, 스레드들이 완전히 멈춰있는 데드락과 달리 지속적으로 무언가를 시도하지만 실제로는 아무런 진전도 이루어지지 않는 상태를 말합니다
. 이는 복도에서 두 사람이 서로 비켜주려다가 계속 부딪히는 상황에 비유될 수 있습니다
.
•
예시: pthread_mutex_trylock 함수를 사용하여 스레드가 두 번째 뮤텍스를 얻지 못하면 첫 번째 뮤텍스마저 해제하는 방식으로 코드를 작성할 경우 발생할 수 있습니다
. 스레드들은 계속해서 락을 획득하고 해제하며 작업을 시도하지만, 임계 영역에 진입하지 못하고 무한히 반복하게 됩니다
.
---
2. 교착상태 처리 방법 (Methods for Handling Deadlocks)
교착상태를 처리하는 방법은 크게 세 가지 접근법으로 나눌 수 있습니다
:
1.
교착상태 방지(Deadlock Prevention): 시스템이 교착상태에 진입하는 것을 아예 막는 방법입니다
.
2.
교착상태 회피(Deadlock Avoidance): 시스템이 교착상태가 될 수 있는 불안전한 상태(unsafe state)로 진입하지 않도록 동적으로 자원 할당을 검사하는 방법입니다
.
3.
교착상태 탐지(Deadlock Detection) 및 회복(Recovery from Deadlock): 시스템이 교착상태에 진입하는 것을 허용하고, 나중에 교착상태가 발생했는지 탐지한 후 회복하는 방법입니다
. 또한, 문제를 무시하고 교착상태가 시스템에 절대 발생하지 않는다고 가정하는 접근법도 있습니다
.
2.1. 교착상태 방지 (Deadlock Prevention)
교착상태 방지는 교착상태의 네 가지 필요 조건 중 하나 이상을 무효화함으로써 교착상태를 방지하는 방법입니다
.
•
상호 배제(Mutual Exclusion): 공유 가능한 자원(예: 읽기 전용 파일)에는 상호 배제가 필요하지 않지만, 공유 불가능한 자원(non-sharable resources)에는 반드시 적용되어야 하므로 이 조건을 완전히 제거하는 것은 어렵습니다
.

•
점유 및 대기(Hold and Wait):
◦
프로세스가 자원을 요청할 때, 다른 어떤 자원도 점유하고 있지 않음을 보장하는 방법입니다
.
◦
예를 들어, 프로세스는 실행을 시작하기 전에 필요한 모든 자원을 한꺼번에 요청하고 할당받거나, 어떤 자원도 할당되지 않은 상태에서만 자원을 요청할 수 있도록 합니다
.
◦
단점으로는 자원 활용률이 낮아지고 기아 상태(starvation)가 발생할 수 있습니다
.
•
비선점(No Preemption):
◦
만약 프로세스가 이미 일부 자원을 점유하고 있는 상태에서 즉시 할당받을 수 없는 다른 자원을 요청한다면, 현재 점유하고 있는 모든 자원을 강제로 해제(release)하도록 합니다
.
◦
선점된 자원들은 해당 프로세스가 기다리는 자원 목록에 추가되며, 프로세스는 원래 점유했던 자원과 새로 요청한 자원 모두를 다시 얻을 수 있을 때만 다시 시작됩니다
.
•
순환 대기(Circular Wait):
◦
모든 자원 유형에 대해 총체적인 순서(total ordering)를 부과하고, 각 프로세스가 자원을 이 순서에 따라 오름차순으로만 요청하도록 요구하는 방법입니다
.
◦
예를 들어, 뮤텍스 락(mutex locks)에 고유한 번호를 할당하고, 항상 낮은 번호의 락부터 높은 번호의 락 순서로 획득하도록 합니다
. 영상에서 first_mutex = 1, second_mutex = 5와 같이 번호를 부여하여 thread_two의 락 획득 순서가 wait(s2) 다음 wait(s1)과 같이 되면 순환 대기가 발생할 수 있음을 보여줍니다
.
◦
이 방법은 교착상태 방지에서 가장 일반적으로 사용되는 방법입니다
.
2.2. 교착상태 회피 (Deadlock Avoidance)
교착상태 회피는 시스템이 교착상태에 절대 진입하지 않도록 동적으로 자원 할당 상태를 검사합니다
. 이를 위해 시스템은 사전에 각 프로세스가 필요로 할 최대 자원 수를 알아야 합니다
.
•
자원 할당 상태(Resource-allocation state): 사용 가능한(available) 자원, 할당된(allocated) 자원, 프로세스의 최대 요구(maximum demands) 수에 의해 정의됩니다
.
•
안전 상태(Safe State):
◦
시스템이 안전 상태에 있다는 것은 모든 스레드가 작업을 완료할 수 있는 일련의 순서(<T1, T2, ..., Tn>)가 존재함을 의미합니다

.
◦
이 순서에서 각 Ti는 현재 사용 가능한 자원과 Tj(j < i)가 사용을 마친 후 반납할 자원만으로도 필요한 자원을 충족하여 작업을 완료할 수 있어야 합니다
.
◦
안전 상태이면 교착상태가 발생하지 않습니다. 그러나 불안전 상태(unsafe state)라고 해서 반드시 교착상태인 것은 아니지만, 교착상태가 발생할 가능성이 있습니다
.
◦
교착상태 회피 알고리즘은 시스템이 불안전 상태로 진입하는 것을 방지하는 것을 목표로 합니다
.
2.2.1. 회피 알고리즘 (Avoidance Algorithms)
•
자원 유형당 인스턴스가 하나인 경우(Single instance of a resource type): 자원 할당 그래프(Resource-Allocation Graph) 알고리즘을 사용합니다
.
◦
클레임 간선(Claim edge): 프로세스 Ti가 자원 Rj를 요청할 수 있음을 나타내는 점선 화살표(Ti --> Rj)입니다. 프로세스 실행 전에 모든 클레임 간선이 그래프에 표시되어야 합니다
.
◦
요청 간선(Request edge): 스레드가 자원을 요청할 때 클레임 간선이 요청 간선(실선)으로 바뀝니다 (Ti → Rj)
.
◦
할당 간선(Assignment edge): 자원이 스레드에 할당될 때 요청 간선이 할당 간선(Ti ← Rj)으로 바뀝니다
.
◦
스레드가 자원을 해제하면 할당 간선은 다시 클레임 간선으로 바뀝니다
.
◦
자원 할당 그래프 알고리즘은 스레드 Ti가 자원 Rj를 요청할 때, 요청 간선을 할당 간선으로 변경하는 것이 자원 할당 그래프에 사이클(cycle)을 형성하지 않는 경우에만 요청을 승인합니다
.
◦
사이클이 없으면 교착상태가 없고, 사이클이 있으면 인스턴스가 하나일 때는 데드락, 여러 개일 때는 데드락 가능성(unsafe)이 있습니다
.
•
자원 유형당 인스턴스가 여러 개인 경우(Multiple instances of a resource type): **은행원 알고리즘(Banker's Algorithm)**을 사용합니다
.
◦
각 스레드는 사전에 자신이 필요로 할 **최대 자원 사용량(maximum use)**을 선언해야 합니다
.
◦
스레드가 자원을 요청할 때 대기해야 할 수도 있습니다
.
◦
모든 자원을 얻은 스레드는 유한한 시간 내에 자원을 반납해야 합니다

.
◦
은행원 알고리즘을 위한 자료 구조 (Data Structures)
:
▪
n: 프로세스(스레드)의 수.
▪
m: 자원 유형의 수.
▪
Available: 길이 m의 벡터. Available[j] = k는 자원 유형 Rj의 k개의 인스턴스가 사용 가능함을 나타냅니다
.
▪
Max: n x m 행렬. Max[i,j] = k는 프로세스 Pi가 자원 유형 Rj의 최대 k개의 인스턴스를 요청할 수 있음을 나타냅니다
.
▪
Allocation: n x m 행렬. Allocation[i,j] = k는 프로세스 Pi가 현재 자원 유형 Rj의 k개의 인스턴스를 할당받았음을 나타냅니다
.
▪
Need: n x m 행렬. Need[i,j] = k는 프로세스 Pi가 작업을 완료하기 위해 자원 유형 Rj의 k개의 인스턴스가 더 필요함을 나타냅니다. Need[i,j] = Max[i,j] – Allocation[i,j]
.
◦
안전성 알고리즘 (Safety Algorithm)
:
1.
Work (길이 m)와 Finish (길이 n) 벡터를 초기화합니다. Work = Available, Finish[i] = false (i = 0, 1, …, n-1)
.
2.
다음 두 조건을 모두 만족하는 i를 찾습니다
: (a) Finish[i] == false (b) Need_i <= Work 만약 그러한 i가 없으면 4단계로 이동합니다
.
3.
Work = Work + Allocation_i. Finish[i] = true. 2단계로 돌아갑니다
.
4.
만약 모든 i에 대해 Finish[i] == true이면, 시스템은 안전 상태입니다
.
◦
자원 요청 알고리즘 (Resource-Request Algorithm for Process Pi)
: 프로세스 Pi가 자원 유형 Rj의 k개 인스턴스를 요청할 때(Request_i 벡터)
:
1.
만약 Request_i <= Need_i이면 2단계로 이동합니다. 그렇지 않으면, 프로세스가 최대 요청량을 초과했으므로 오류 조건을 발생시킵니다
.
2.

만약 Request_i <= Available이면 3단계로 이동합니다. 그렇지 않으면, 자원이 사용 가능하지 않으므로 Ti는 대기해야 합니다
.
3.
요청된 자원을 Ti에 할당하는 것을 가정하여 시스템 상태를 임시로 수정합니다
: Available = Available – Request_i; Allocation_i = Allocation_i + Request_i; Need_i = Need_i – Request_i;
4.
이 새로운 상태에서 안전성 알고리즘을 실행합니다
.
•
만약 안전하면, 자원이 Ti에 할당됩니다
.
•
만약 불안전하면, Ti는 대기해야 하며 이전 자원 할당 상태로 복원됩니다
.
◦
예시: 5개 스레드(T0-T4), 3개 자원 유형(A:10, B:5, C:7)의 스냅샷을 통해 은행원 알고리즘을 적용한 결과, <T1, T3, T4, T2, T0> 순서가 안전성 기준을 만족함을 보입니다
. T1이 (1,0,2)를 요청했을 때, 시스템이 안전 상태로 유지될 수 있음을 보여주는 추가 예시도 제공됩니다
.
2.3. 교착상태 탐지 (Deadlock Detection)
교착상태 탐지는 시스템이 교착상태에 진입하는 것을 허용하고, 나중에 교착상태 여부를 주기적으로 검사하여 탐지하는 방법입니다
.
•
자원 유형당 인스턴스가 하나인 경우(Single Instance of Each Resource Type): **대기 그래프(Wait-for graph)**를 유지합니다
.
◦
노드는 스레드를 나타내고, Ti → Tj는 스레드 Ti가 스레드 Tj를 기다리고 있음을 나타냅니다
.
◦
주기적으로 그래프에서 사이클을 탐지합니다. 사이클이 존재하면 교착상태가 존재합니다
.
◦
사이클 탐지 알고리즘은 n이 그래프의 정점(스레드) 수일 때 O(n^2)의 연산이 필요합니다
.
•
자원 유형당 인스턴스가 여러 개인 경우(Several Instances of a Resource Type): 은행원 알고리즘과 유사한 탐지 알고리즘을 사용합니다
.
◦
자료 구조
:
▪
Available: 길이 m의 벡터. 사용 가능한 자원의 수
.
▪
Allocation: n x m 행렬. 각 프로세스에 현재 할당된 자원의 수

.
▪
Request: n x m 행렬. 각 프로세스의 현재 요청량. Request[i][j] = k는 스레드 Ti가 자원 유형 Rj의 k개 인스턴스를 더 요청하고 있음을 나타냅니다
.
◦
탐지 알고리즘 (Detection Algorithm)
:
1.
Work (길이 m)와 Finish (길이 n) 벡터를 초기화합니다
: (a) Work = Available (b) Allocation_i != 0이면 Finish[i] = false, 그렇지 않으면 Finish[i] = true로 초기화합니다
.
2.
다음 두 조건을 모두 만족하는 i를 찾습니다
: (a) Finish[i] == false (b) Request_i <= Work 그러한 i가 없으면 4단계로 이동합니다
.
3.
Work = Work + Allocation_i. Finish[i] = true. 2단계로 돌아갑니다
.
4.
만약 일부 i에 대해 Finish[i] == false이면 시스템은 교착상태입니다. 이때 Finish[i] == false인 Ti는 교착상태에 빠진 스레드입니다
.
▪
이 알고리즘은 시스템이 교착상태인지 탐지하는 데 O(m x n^2)의 연산이 필요합니다
.
◦
예시: 5개 스레드(T0-T4), 3개 자원 유형(A:7, B:2, C:6)의 스냅샷을 통해 탐지 알고리즘을 적용합니다
. 초기 상태에서 <T0, T2, T3, T1, T4> 순서가 안전 기준을 만족하여 모든 Finish[i]가 true가 됩니다. 그러나 T2가 추가적으로 C 유형 자원 1개를 요청하는 상황이 발생하면, 탐지 알고리즘 수행 시 T1, T2, T3, T4는 Finish[i]가 false로 남아 교착상태가 존재함을 확인합니다
.
•
탐지 알고리즘 사용 시점 (Detection-Algorithm Usage)
:
◦
교착상태가 얼마나 자주 발생하는지.
◦
얼마나 많은 프로세스를 롤백(rollback)해야 하는지.
◦
자원 그래프에 많은 사이클이 존재할 때, 어떤 프로세스가 교착상태를 "유발"했는지 판단하기 어렵습니다
.
2.4. 교착상태로부터의 회복 (Recovery from Deadlock)
교착상태가 탐지된 후에는 시스템을 회복시켜야 합니다
.
•
프로세스 종료(Process Termination)
:
◦

모든 교착상태 프로세스 강제 종료(Abort all deadlocked processes): 교착상태에 연루된 모든 프로세스를 한꺼번에 종료합니다
.
◦
하나씩 종료(Abort one process at a time): 교착상태 사이클이 제거될 때까지 프로세스를 하나씩 종료합니다
.
◦
종료할 프로세스 선택 기준(Order to abort)
:
1.
프로세스의 우선순위(Priority of the process).
2.
프로세스가 지금까지 계산한 시간과 완료까지 남은 시간 (How long process has computed, and how much longer to completion).
3.
프로세스가 사용한 자원(Resources the process has used).
4.
프로세스 완료에 필요한 자원(Resources process needs to complete).
5.
종료해야 할 프로세스의 수(How many processes will need to be terminated).
6.
프로세스가 대화식(interactive)인지 배치(batch)인지 여부
.
•
자원 선점(Resource Preemption)
:
◦
희생자 선택(Selecting a victim): 교착상태에서 회복하기 위해 자원을 선점할 프로세스(희생자)를 선택하며, 이로 인한 비용을 최소화하는 것이 목표입니다
.
◦
롤백(Rollback): 선택된 희생자 프로세스를 안전한 상태로 되돌리고, 그 상태부터 다시 시작합니다
.
◦
기아 상태(Starvation): 동일한 프로세스가 계속해서 희생자로 선택될 수 있으므로, 롤백 횟수를 비용 요소에 포함하여 방지해야 합니다
.