4단원. 프로세스 동기화 (최빈출 단원)

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

Q1. 경쟁 상태(Race Condition)란 무엇인가요? 예를 들어 설명하세요. ★★★

답안 여러 프로세스나 스레드가 공유 데이터에 동시에 접근할 때, 실행 순서(타이밍)에 따라 결과가 달라지는 상태입니다. 예를 들어 두 스레드가 counter++를 동시에 수행하면, 이 연산은 실제로 읽기→증가→쓰기 세 단계로 이뤄지므로 서로의 중간 결과를 덮어써 2 증가해야 할 값이 1만 증가할 수 있습니다. 은행 계좌 이중 인출, 재고 차감 오류 등이 실무적 사례이며, 공유 자원 접근 구간(임계 구역)을 상호 배제로 보호해야 합니다.

관련 개념 공유 자원, 원자성 부재, counter++의 기계어 3단계, 임계 구역

Q2. 임계 구역(Critical Section) 문제의 해결 조건 3가지를 설명하세요. ★★★

답안 ① **상호 배제(Mutual Exclusion)**: 한 프로세스가 임계 구역에 있으면 다른 프로세스는 진입할 수 없습니다. ② **진행(Progress)**: 임계 구역이 비어 있고 진입을 원하는 프로세스가 있으면, 진입 결정이 무한정 미뤄져서는 안 됩니다. ③ **한정 대기(Bounded Waiting)**: 진입 요청 후 다른 프로세스의 진입 횟수에 상한이 있어 무한정 기다리지 않아야 합니다(기아 방지). 세 조건 중 하나라도 빠지면 정확성 또는 공정성이 깨집니다.

관련 개념 상호 배제, 진행, 한정 대기, 피터슨 알고리즘(3조건 만족의 SW적 예)

Q3. 뮤텍스와 세마포어의 차이를 설명하세요. ★★★

답안 뮤텍스는 소유권 개념이 있는 잠금(lock)으로, 락을 잡은 스레드만 해제할 수 있으며 한 번에 하나의 스레드만 임계 구역에 진입시킵니다. **세마포어**는 정수 카운터로, wait(P)에서 감소·0이면 대기, signal(V)에서 증가하며, 소유권이 없어 다른 스레드가 signal할 수 있습니다. 카운팅 세마포어는 N개의 동일 자원(커넥션 풀 등) 접근 제어에 쓰이고, 이진 세마포어는 뮤텍스처럼 쓸 수 있지만 소유권·우선순위 상속이 없다는 차이가 있습니다. 요약하면 뮤텍스는 '잠금', 세마포어는 '개수 있는 신호 전달' 메커니즘입니다.

관련 개념 소유권, wait(P)/signal(V), 이진/카운팅 세마포어, 커넥션 풀

Q4. 스핀락과 슬립 기반 락(뮤텍스)의 차이와 각각 유리한 상황은? ★★★

답안 스핀락은 락을 얻을 때까지 반복 검사(busy waiting)하며 CPU를 소모하고, **뮤텍스**는 락을 못 얻으면 스레드를 재워(블록) 다른 작업에 CPU를 양보합니다. 스핀락은 컨텍스트 스위칭 비용이 없으므로 임계 구역이 매우 짧고 멀티코어에서 락이 곧 풀릴 것으로 기대될 때 유리하며, 커널 내부나 인터럽트 컨텍스트(잠들 수 없는 곳)에서 필수입니다. 반대로 임계 구역이 길거나 단일 코어라면 스핀은 순수 낭비이므로 슬립 기반이 낫습니다. 리눅스의 futex는 경쟁이 없으면 사용자 공간에서 처리하고 경쟁 시에만 커널로 들어가는 하이브리드입니다.

관련 개념 busy waiting, 컨텍스트 스위칭 비용 비교, futex, 적응형 뮤텍스

Q5. 세마포어 wait/signal 연산으로 생산자-소비자 문제를 어떻게 해결하나요? ★★★

답안 유한 버퍼에 생산자는 넣고 소비자는 꺼내는 문제로, 세마포어 3개를 씁니다. empty(초기값 N: 빈 칸 수), full(초기값 0: 찬 칸 수), mutex(초기값 1: 버퍼 접근 보호). 생산자는 wait(empty) → wait(mutex) → 버퍼에 추가 → signal(mutex) → signal(full), 소비자는 wait(full) → wait(mutex) → 꺼내기 → signal(mutex) → signal(empty) 순서입니다. wait(mutex)를 wait(empty)보다 먼저 하면 버퍼가 가득 찼을 때 mutex를 쥔 채 잠들어 **데드락**이 되므로 순서가 중요합니다.

관련 개념 유한 버퍼, empty/full/mutex, wait 순서와 데드락, 조건 변수 대안

Q6. 모니터(Monitor)란 무엇이고 세마포어보다 나은 점은? ★★

답안 모니터는 공유 데이터와 그 데이터를 다루는 프로시저를 하나로 묶고, 한 시점에 하나의 스레드만 모니터 내부에서 실행되도록 언어 차원에서 보장하는 고수준 동기화 구조입니다. 조건 변수(wait/signal)로 조건 대기를 표현합니다. 세마포어는 wait/signal 호출 순서를 하나만 틀려도 데드락이나 상호 배제 실패가 나는데, 모니터는 상호 배제가 자동이라 실수 여지가 적습니다. 자바의 synchronized/wait/notify가 모니터 개념의 구현입니다.

관련 개념 조건 변수, Hoare vs Mesa 시맨틱, 자바 synchronized, spurious wakeup(while로 조건 재검사)

Q7. 데드락과 라이브락, 기아의 차이를 설명하세요. ★★

답안 **데드락**은 프로세스들이 서로가 가진 자원을 기다리며 아무도 진행하지 못하는 정지 상태입니다. **라이브락**은 프로세스들이 멈추지는 않고 계속 상태를 바꾸지만(예: 서로 양보를 반복) 실질적 진전이 없는 상태입니다. **기아**는 시스템 전체는 진행되지만 특정 프로세스만 자원을 계속 할당받지 못하는 상태입니다. 데드락은 자원 그래프의 사이클, 라이브락은 잘못된 회피 로직, 기아는 불공정한 스케줄링/락 정책이 원인입니다.

관련 개념 자원 할당 그래프, 백오프(라이브락 해결), 공정 락, 에이징

Q8. 원자적 연산(CAS)과 락프리 프로그래밍을 설명하세요. ★★

답안 CAS(Compare-And-Swap)는 '메모리 값이 기댓값과 같으면 새 값으로 교체'를 하드웨어가 원자적으로 수행하는 명령입니다. 락 없이 읽기→계산→CAS로 갱신하고, 실패하면(다른 스레드가 먼저 바꿈) 재시도하는 방식으로 락프리 자료구조를 만듭니다. 락 획득/해제와 블로킹이 없어 경쟁이 적을 때 매우 빠르고 데드락이 없지만, 경쟁이 심하면 재시도 낭비가 커지고 ABA 문제(값이 A→B→A로 돌아와 변경을 감지 못함)에 주의해야 합니다. 자바의 AtomicInteger, 무락 큐 등이 활용 예입니다.

관련 개념 CAS, ABA 문제(버전 카운터로 해결), lock-free/wait-free, 원자적 명령(test-and-set)

Q9. test_and_set 같은 하드웨어 명령이 동기화에 왜 필요한가요? ★★

답안 소프트웨어만으로 상호 배제를 구현(피터슨 알고리즘)할 수 있지만, 현대 CPU의 명령어 재배치와 멀티코어 캐시 일관성 문제 때문에 순수 SW 방식은 신뢰하기 어렵고 비효율적입니다. test_and_set, compare_and_swap 같은 명령은 '검사와 수정'을 하나의 원자적 단위로 하드웨어가 보장하므로, 그 사이에 다른 코어가 끼어들 수 없습니다. 모든 현대 락 구현(스핀락, 뮤텍스)의 기반이 이 원자적 명령입니다.

관련 개념 원자성, 명령어 재배치, 메모리 배리어, 피터슨 알고리즘의 한계

Q10. 데이터베이스나 파일에 여러 프로세스가 접근할 때 동기화는 어떻게 하나요? (실무 연계) ★★

답안 프로세스 간에는 스레드용 뮤텍스를 그대로 쓸 수 없으므로 다른 수단을 씁니다. ① **파일 락**: flock/fcntl로 파일 단위 advisory lock을 겁니다(크론 작업 중복 실행 방지 등). ② **공유 메모리 + 프로세스 공유 뮤텍스/네임드 세마포어**. ③ **DB 수준 락**: 트랜잭션과 행 단위 락, SELECT ... FOR UPDATE. ④ **분산 환경**: 단일 호스트를 넘어서면 Redis(SETNX 기반), ZooKeeper, etcd를 이용한 분산 락을 사용합니다. 분산 락은 락 소유자의 장애를 대비해 TTL(임대 기간)과 펜싱 토큰을 함께 설계해야 안전합니다.

관련 개념 flock/fcntl, 네임드 세마포어, SELECT FOR UPDATE, 분산 락(Redis/etcd), 펜싱 토큰

Q11. 리더-라이터(Readers-Writers) 문제를 설명하세요. ★★

답안 공유 데이터에 대해 읽기는 여러 스레드가 동시에 해도 안전하지만, 쓰기는 배타적이어야 하는 문제입니다. 읽기 우선 방식은 읽기 처리량이 좋지만 쓰기가 기아에 빠질 수 있고, 쓰기 우선 방식은 그 반대입니다. 해결 도구가 **읽기-쓰기 락(rwlock)**으로, 읽기 락은 공유 획득, 쓰기 락은 배타 획득합니다. 읽기가 압도적으로 많은 워크로드(설정 캐시, 라우팅 테이블)에서 뮤텍스보다 처리량이 크게 좋아집니다. DB의 공유 락(S)/배타 락(X)도 같은 개념입니다.

관련 개념 rwlock, 읽기/쓰기 우선과 기아, S-lock/X-lock, RCU(리눅스 커널의 읽기 최적화)

Q12. 식사하는 철학자 문제와 해결 방법을 설명하세요. ★★

답안 원탁의 철학자 5명이 양옆 포크 2개를 들어야 식사할 수 있는 상황에서, 모두가 동시에 왼쪽 포크를 들면 오른쪽 포크를 영원히 기다리는 데드락이 되는 고전 문제입니다. 해결책은 ① 동시에 식사 가능한 인원을 4명으로 제한(카운팅 세마포어), ② 포크 2개를 모두 집을 수 있을 때만 집기(원자적 획득), ③ **자원에 번호를 매겨 홀수 철학자는 왼쪽부터, 짝수는 오른쪽부터 집기(비대칭 획득 순서)** 등이 있습니다. 핵심 교훈은 '락 획득 순서를 전역적으로 일관되게 하면 순환 대기가 깨진다'는 것으로, 실무의 다중 락 코드에도 그대로 적용됩니다.

관련 개념 순환 대기 제거, 락 순서 규칙(lock ordering), 데드락 4조건과의 연결

Q13. 컨텍스트가 다른 두 스레드가 같은 데이터를 볼 때 발생하는 가시성 문제란? ★★

답안 각 코어는 자체 캐시와 스토어 버퍼를 갖고 컴파일러·CPU는 명령을 재배치하므로, 한 스레드가 쓴 값을 다른 스레드가 즉시 또는 순서대로 보지 못할 수 있습니다. 이것이 **가시성/순서(ordering) 문제**입니다. 락은 상호 배제뿐 아니라 메모리 배리어 역할도 하여 락 경계에서 가시성을 보장합니다. 락 없이 플래그 변수로 통신하면(예: while(!done)) 컴파일러 최적화로 무한 루프가 될 수 있어, volatile(자바)이나 atomic 타입(C++/C11)으로 선언해야 합니다.

관련 개념 캐시 일관성(MESI), 메모리 배리어, volatile, happens-before 관계

Q14. 뮤텍스 없이 싱글 스레드 이벤트 루프로 동시성을 처리하는 방식(Node.js/Redis)은 어떻게 안전한가요? ★★

답안 이벤트 루프 모델은 하나의 스레드가 이벤트 큐에서 작업을 꺼내 순차 실행하므로, 애플리케이션 코드 수준에서 두 작업이 동시에 같은 데이터를 만질 수 없어 락이 필요 없습니다. I/O는 비동기(epoll 등)로 처리해 스레드가 놀지 않게 합니다. 대신 ① CPU를 오래 쓰는 작업 하나가 전체 루프를 블록하고, ② 멀티코어 활용은 프로세스를 여러 개 띄워야(클러스터링) 합니다. Redis가 단일 스레드로도 빠른 이유는 메모리 연산 위주 + 이벤트 루프 + 락 오버헤드 부재 덕분입니다.

관련 개념 이벤트 루프, epoll, 논블로킹 I/O, Redis 단일 스레드 모델, Node 클러스터

Q15. 동기화 관점에서 데드락을 예방하는 코딩 습관을 말해보세요. (실무 연계) ★★

답안 ① 여러 락을 잡을 때는 항상 **전역적으로 동일한 순서**로 획득합니다(순환 대기 제거). ② 락 보유 시간을 최소화하고, 락을 쥔 채 I/O나 외부 호출을 하지 않습니다. ③ 가능하면 락 하나로 해결하도록 설계를 단순화하거나 락 범위를 통합합니다. ④ trylock+타임아웃으로 무한 대기를 피하고 실패 시 보유 락을 놓고 재시도합니다(단, 라이브락 주의). ⑤ 락 대신 불변 데이터, 메시지 전달, 원자적 연산으로 공유 자체를 줄입니다. 실제 장애의 상당수는 락 순서 불일치에서 나오므로 ①이 가장 중요합니다.

관련 개념 lock ordering, trylock, 락 범위 최소화, 불변성, 메시지 전달 아키텍처
