제공해주신 강의 자료, 교재 챕터, 그리고 기말고사 스타일을 바탕으로 작성된 **병행성(Concurrency), 락(Locks), 동기화(Synchronization)**에 대한 포괄적인 자체 학습 가이드의 한국어 번역본입니다.

이 가이드는 시험 형식에서 비중 있게 다루어지는 **컴퓨터 구조(하드웨어 프리미티브)**와 **저수준 코드 구현**을 강조하고 있습니다.

운영체제: 병행성(Concurrency) 및 동기화(Synchronization) 학습 가이드

1. 병행성의 기초 (Fundamentals of Concurrency)

스레드 상태 및 메모리 레이아웃

- **비공개 (스레드 로컬):** 각 스레드는 고유의 **스택(Stack)**을 가집니다.
  - 포함 내용: 지역 변수, 반환 주소(return addresses), 함수 매개변수.
  - *시험 팁:* 한 스레드의 스택에 있는 지역 변수를 가리키는 포인터를 *절대로* 다른 스레드에 전달해서는 안 됩니다.
- **공유 (프로세스 전역):**
  - **힙(Heap):** 동적 메모리 할당 (`malloc` / `new`).
  - **전역 변수(Global Variables):** 정적 데이터 영역(Static data segment).
  - **코드(Code):** 명령어(Instructions)는 공유됩니다.

경쟁 상태 (The Race Condition)

- **정의:** 프로그램의 결과가 스레드들의 비결정적인 타이밍이나 공유 데이터 접근 순서(interleaving)에 의존하는 경우.
- **임계 영역 (Critical Section, CS):** 공유 자원에 접근하는 코드 세그먼트.
- **해결을 위한 요구 사항:**
  i. **상호 배제 (Mutual Exclusion):** 한 번에 하나의 스레드만 CS에 들어갈 수 있어야 함.
  ii. **진행 (Progress, 데드락 방지):** CS에 아무도 없고 진입을 원하는 스레드가 있다면 진입이 허용되어야 함.
  iii. **한정 대기 (Bounded Waiting, 기아 방지):** 대기 중인 스레드는 언젠가 반드시 진입해야 함 (공정성).

2. 하드웨어 프리미티브 (원자적 명령어)

락(Lock)을 구현하기 위해, 하드웨어는 메모리를 원자적(atomically, 분할 불가능하게)으로 읽고 업데이트하는 명령어를 제공해야 합니다.

A. 인터럽트 비활성화 (단일 프로세서 전용)

- **메커니즘:** CS 진입 전 `cli()` (비활성화), CS 후 `sti()` (활성화).
- **장점:** 단순함.
- **단점:**
  - 위험함 (사용자 코드가 시스템을 멈추게 할 수 있음).
  - **멀티프로세서에서 작동하지 않음** (다른 코어가 여전히 메모리에 접근 가능).
  - 중요한 이벤트(타이머, 디스크 I/O)를 놓칠 수 있음.

B. Test-And-Set (TAS) / 원자적 교환

- **동작:** 메모리 위치의 *이전(old)* 값을 반환함과 동시에 그 위치에 `1` (또는 새로운 값)을 씁니다.
- **C 스타일 정의:**

```
int TestAndSet(int *ptr, int new_val) {
    int old = *ptr;
    *ptr = new_val;
    return old; // 설정하기 전(BEFORE)의 값을 반환
}
```

- **락 구현:**

```
void lock(lock_t *L) {
    while (TestAndSet(&L->flag, 1) == 1)
        ; // 스핀 대기 (Spin-wait)
}
void unlock(lock_t *L) {
    L->flag = 0;
}
```

- **분석:** 상호 배제는 제공하지만 **공정성이 없음** (기아 상태 발생 가능).

C. Compare-And-Swap (CAS)

- **동작:** `ptr` 이 현재 `expected` 값과 같을 때만 `ptr` 을 `new_val` 로 업데이트합니다. 원래 값을 반환합니다.

- **아키텍처:** `cmpxchg` (x86) 또는 `CAS` (SPARC)로 알려짐.
- **C 스타일 정의:**

```
int CompareAndSwap(int *ptr, int expected, int new_val) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new_val;
    return actual;
}
```

- **사용:** TAS보다 강력하며, 대기 없는(wait-free) 동기화의 기초가 됩니다.

D. Load-Linked / Store-Conditional (LL/SC)

- **아키텍처:** MIPS, ARM (`ldrex` / `strex`), RISC-V.
- **로직:** 낙관적 락킹 (Optimistic locking).
  - `LL` : 값을 읽음.
  - `SC` : `LL` 이후 해당 주소에 **다른 저장이 발생하지 않았을 경우에만** 값을 씀.
  - `SC` 반환값: 성공 시 `1` , 실패 시 `0` .
- **락 구현:**

```
void lock(lock_t *L) {
    while (1) {
        while (LoadLinked(&L->flag) == 1) ; // 락이 걸려있는 동안 스핀
        if (StoreConditional(&L->flag, 1) == 1)
            return; // 성공
    }
}
```

E. Fetch-And-Add (FAA)

- **동작:** 원자적으로 값을 증가시키고 *이전(old)* 값을 반환합니다.
- **핵심 사용 사례: 티켓 락 (Ticket Locks)** (공정성/한정 대기 보장).
- **티켓 락 구현:**

```
struct lock_t { int ticket; int turn; };

void lock(lock_t *L) {
    int myturn = FetchAndAdd(&L->ticket, 1);
    while (L->turn != myturn)
        ; // 내 차례가 될 때까지 스핀
}
void unlock(lock_t *L) {
    L->turn++; // 다음 사람 차례
}
```

- **분석: FIFO (선입선출)** 순서를 보장합니다. 기아 상태를 해결합니다.

3. 소프트웨어 전용 알고리즘

특수 원자적 명령어 없이(원자적 Load/Store에만 의존하여) 락킹을 해결하려는 시도.

피터슨 알고리즘 (Peterson's Algorithm, 2개 스레드 전용)

- **변수:** `flag[]` (진입 의사), `turn` (우선권).
- **로직:**
  i. 나의 깃발(`flag`)을 든다.
  ii. `turn` 을 *상대방*으로 설정한다 (양보).
  iii. 대기 조건: 상대방이 진입을 원하고(`flag` 가 1) **그리고** 상대방의 차례(`turn`)인 경우.
- **코드:**

```
flag[me] = 1;
turn = other;
while (flag[other] == 1 && turn == other) ; // 스핀
```

램포트의 빵집 알고리즘 (Lamport's Bakery Algorithm, N개 스레드)

- **개념:** 빵집 번호표와 같음. 번호표를 뽑고, 가장 낮은 번호가 먼저 진입.
- **동점 처리:** 스레드들이 같은 번호를 받으면, 더 낮은 프로세스 ID(PID)를 가진 스레드가 우선함.
- **`choosing` 변수:**
  - *중요:* 스레드가 번호를 고르는 동안 `choosing[me] = true` 로 설정.
  - *이유:* 이 변수가 없다면, 첫 번째 스레드가 매우 큰 번호를 계산하는 도중 다른 스레드가 첫 번째 스레드의 `number[me]` 를 0(비활성)으로 읽고 먼저 진입하여 상호 배제가 깨질 수 있음.

- **사전적 순서 (Lexicographical Order):** `a < c` 이거나 `(a == c && b < d)` 인 경우 `(a, b) < (c, d)` 로 판단.

4. 고급 락킹 전략 (OS 지원)

스핀 문제 (The Spinning Problem)

- **시나리오:** 스레드 A가 락을 잡고 있는 상태에서 선점(타이머 인터럽트) 당함. 스레드 B가 실행되어 락을 얻으려 하지만 실패하고 계속 스핀(회전)함.
- **결과:** 스레드 B는 아무 유용한 작업도 하지 않고 전체 타임 슬라이스를 낭비함.
- **해결책 1: 양보 (Yield)**
  - 명령어: `yield()` (CPU를 포기하고 실행 상태 -> 준비 상태로 이동).
  - 코드: `if (locked) yield();`
  - *한계:* 문맥 교환(Context switching) 비용이 여전히 높음. 기아 상태 가능성 여전함.

해결책 2: 큐 (Sleep Locks)

- **메커니즘:** 스핀하는 대신 대기 스레드를 재우고(차단 상태, Blocked), 락이 해제되면 깨움.
- **Solaris API:** `park()` (재우기), `unpark(tid)` (깨우기).
- **경쟁 상태 (깨우기 신호 유실, The Lost Wakeup):**
  - 스레드 B가 `park()` 를 하기로 결정했지만, 실제로 호출하기 직전에 스레드 A가 락을 해제하고 `unpark(B)` 를 호출함.
  - 그 후 B가 `park()` 를 호출하면 영원히 잠들게 됨 (신호를 놓침).
  - *해결:* `setpark()` 를 사용하여 잠들 의사를 표시. `setpark` 와 `park` 사이에 `unpark` 가 호출되면 `park` 는 즉시 리턴함.

해결책 3: Futex (Linux)

- **Fast Userspace Mutex.**
- **개념:**
  - **Fast Path:** 사용자 공간에서 원자적 명령어(CAS)를 사용. 락이 비어있으면 바로 획득 (시스템 콜 없음).
  - **Slow Path:** 경쟁이 발생하면 시스템 콜(`futex_wait`)을 통해 커널 내에서 잠듦.
- **최적화:** 하이브리드 접근 방식. 경쟁이 적을 때 매우 효율적.

해결책 4: 2단계 락 (Two-Phase Locks)

1. **1단계:** 짧은 시간 동안 스핀 (락 보유자가 다른 CPU에 있고 곧 끝날 것을 기대).
2. **2단계:** 락을 얻지 못하면 잠듦(Sleep).

5. 흔한 병행성 함정 (Common Concurrency Pitfalls)

A. 우선순위 역전 (Priority Inversion)

- **시나리오:** 고우선순위(H), 중우선순위(M), 저우선순위(L).
  i. L이 락을 획득.
  ii. H가 락 획득 시도 -> 차단됨 (L을 기다림).
  iii. M이 실행 가능해짐. M > L 이므로 M이 L을 선점.
  iv. **결과:** H > M 임에도 불구하고, H는 사실상 M(그리고 L)을 기다리게 됨.
- **해결책:**
  - **우선순위 상속 (Priority Inheritance):** L이 H가 필요한 락을 잡고 있는 동안, L은 일시적으로 H의 우선순위를 상속받음.
  - **우선순위 상한 (Priority Ceiling):** 자원에 접근하는 모든 스레드는 해당 자원과 연관된 최고 우선순위로 격상됨.

B. 교착 상태 (Deadlock)

- **정의:** 대기 중인 스레드들의 사이클; 아무도 진행할 수 없음.
- **4가지 필수 조건 (모두 충족되어야 함):**
  i. **상호 배제 (Mutual Exclusion):** 자원이 독점적으로 점유됨.
  ii. **점유 및 대기 (Hold and Wait):** 자원을 가진 채로 다른 자원을 기다림.
  iii. **비선점 (No Preemption):** 자원을 강제로 뺏을 수 없음.
  iv. **환형 대기 (Circular Wait):** T1은 T2를, T2는 ... Tn을, Tn은 T1을 기다림.
- **예방 (조건 중 하나 깨기):**
  - *점유 및 대기 타파:* 필요한 모든 락을 한 번에 원자적으로 획득.
  - *환형 대기 타파 (가장 실용적):* **락 순서 정하기 (Lock Ordering)**. 락에 전역 순서(주소 순서 등)를 부여. 항상 락 A를 락 B보다 먼저 획득하도록 함.

6. 코드 구현: 장벽(Barriers) 및 세마포어(Semaphores)

(시험 스타일 문제 중심)

장벽 동기화 (Barrier Synchronization)

- **목표:** N개의 스레드가 모두 특정 지점에 도착해야만 *누구든* 다음으로 진행할 수 있음.

뮤텍스 & 조건 변수(Condition Variable)를 이용한 구현 (Mesa 시맨틱):

```
typedef struct {
    int n;           // 필요한 총 스레드 수
    int count;       // 현재 도착한 수
    mutex_t m;
    cond_t cv;
} barrier_t;

void barrier_init(barrier_t *b, int n) {
    b->n = n;
    b->count = 0;
    mutex_init(&b->m);
    cond_init(&b->cv);
}

void barrier(barrier_t *b) {
    mutex_lock(&b->m);
    b->count++;
    if (b->count == b->n) {
        cond_broadcast(&b->cv); // 모두를 깨움
        // 재사용을 위해 count를 리셋하려면 더 복잡한 로직이 필요함
        // (예: 세대 카운터나 두 번째 장벽 등)
    } else {
        while (b->count < b->n) {
            cond_wait(&b->cv, &b->m);
        }
    }
    mutex_unlock(&b->m);
}
```

세마포어 (Semaphores)

- **정의:** 정수 값을 가진 객체.
- **연산:**
  - `sema_wait()` (P): 감소. 값이 0보다 작으면 대기.
  - `sema_post()` (V): 증가. 대기자 하나를 깨움.
- **세마포어를 이용한 순서 정하기:**
  - 스레드 A가 스레드 B를 기다리게 하려면:
    - 초기화 `sem = 0`.
    - 스레드 A: `sema_wait(sem)` (즉시 차단됨).
    - 스레드 B: `do_work(); sema_post(sem)`.

7. 하드웨어/어셈블리 코드 분석 팁

("x86.py" 시뮬레이션 참조 기반)

- **명령어 원자성 (Instruction Atomicity):** C 코드 한 줄이 하나의 어셈블리 명령어로 매핑되는지 여러 개로 매핑되는지 항상 확인하세요.
  - `g++` 는 원자적이지 **않습니다** (Load `g`, Add 1, Store `g`). 3개의 명령어가 필요합니다.
  - 이 명령어들 *사이*에 인터럽트가 발생할 수 있습니다.
- **스핀 루프 효율성:**
  - `while(TAS(&lock, 1));` 은 높은 버스 트래픽(캐시 일관성 폭풍)을 유발합니다.
  - **Test-and-Test-and-Set:** 일반적인 읽기로 먼저 값을 확인하고(캐시 효율 좋음), *그 후* 락이 비어 보일 때만 원자적 TAS를 시도합니다.

```
while(1) {
    while(lock->flag == 1); // 캐시된 복사본으로 스핀
    if(TAS(&lock->flag, 1) == 0) break; // 획득 시도
}
```