다음은 제공된 PDF 자료를 바탕으로 작성된 학습 노트를 한국어로 번역한 내용입니다. 코드 로직과 아키텍처 세부 사항에 중점을 두어 시험 대비용으로 정리했습니다.

1. 세마포어 기초 (Semaphore Fundamentals)

정의

- **세마포어(Semaphore)**는 동기화에 사용되는 정수 값을 가진 객체입니다.
- 두 가지 연산인 `sem_wait()` (또는 P)와 `sem_post()` (또는 V)를 통해 **원자적(atomically)**으로 조작됩니다.
- **상태 불변 조건 (State Invariant):** 정수 값에 따라 동작이 결정됩니다.
  - **양의 정수:** 사용 가능한 자원의 수를 나타냅니다.
  - **음의 정수:** 그 절대값(magnitude)은 대기 중인 스레드의 수를 나타냅니다 (엄격한 다익스트라 정의에 따름).

연산

- **`sem_wait(s)` / $P()$:**
  - 세마포어 `s` 의 값을 1 감소시킵니다.
  - 결과 값이 음수(< 0)이면, 호출한 스레드는 **블록(blocked)**됩니다 (재워짐).
- **`sem_post(s)` / $V()$:**
  - 세마포어 `s` 의 값을 1 증가시킵니다.
  - 대기 중인 스레드가 있다면 (값이 음수였거나 0이 되는 로직), 대기 중인 스레드 하나를 **깨웁니다 (wake up)**.

초기화 전략

- **상호 배제 (Lock)용:** 초기값 = **1** (자원을 즉시 사용 가능).
- **순서 정하기 (이벤트 대기)용:** 초기값 = **0** (무언가 일어날 때까지 대기).
- **자원 풀(Resource Pool)용:** 초기값 = **N** (사용 가능한 유닛의 수).

2. 기본 패턴: 락 & 순서 (Basic Patterns: Locks & Ordering)

A. 이진 세마포어 (The Lock)

임계 구역(Critical Section)에서 **상호 배제(Mutual Exclusion)**를 보장하기 위해 사용됩니다.

- **초기화:** `sem_init(&m, 0, 1);`
- **코드 패턴:**

```
sem_wait(&m);       // 락 획득 (0으로 감소)
// 임계 구역 (CRITICAL SECTION)
sem_post(&m);       // 락 해제 (1로 증가)
```

- **실행 추적(Trace Analysis):**
  - 스레드 A가 `wait()` 를 호출하면 (값이 0이 됨) 진입합니다.
  - A가 안에 있는 동안 스레드 B가 `wait()` 를 호출하면 (값이 -1이 됨) B는 잠듭니다.
  - A가 `post()` 를 호출하면 값이 0이 되고, B가 깨어납니다.

B. 순서 정하기 (Rendezvous)

스레드 A가 스레드 B의 작업 완료를 기다려야 할 때 사용합니다.

- **초기화:** `sem_init(&s, 0, 0);`
- **코드 패턴:**
  - **부모:** `sem_wait(&s);` (자식을 기다림).
  - **자식:** `...작업 수행...; sem_post(&s);` (완료 신호 보냄).
- **로직:**
  - 부모가 먼저 실행될 경우: -1로 감소 → 잠듦. 자식이 실행되어 `post` (0으로 증가) → 부모를 깨움.
  - 자식이 먼저 실행될 경우: `post` (1로 증가). 부모가 실행되어 0으로 감소 → 대기 없이 즉시 리턴.

3. 생산자-소비자 문제 (The Producer-Consumer Problem)

시나리오:

- 생산자(Producer)는 데이터를 생성하고, 소비자(Consumer)는 데이터를 가져갑니다.
- 크기가 `MAX` 인 공유 버퍼를 사용합니다.
- 요구사항: 상호 배제(한 번에 하나만 버퍼 접근) + 동기화(꽉 차면 생산 중단, 비면 소비 중단).

변수 및 초기화:

1. `sem_t empty;` 초기값 = **MAX** (빈 슬롯의 수).
2. `sem_t full;` 초기값 = **0** (채워진 슬롯의 수).
3. `sem_t mutex;` 초기값 = **1** (버퍼 인덱스 보호용 이진 락).

올바른 구현

중첩 순서에 주목하세요. **락(`mutex`)은 반드시 조건 확인(`empty` / `full` 대기) *내부*에서 획득해야 합니다.**

```
void producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);   // P1: 빈 슬롯 대기
        sem_wait(&mutex);   // P1.5: 락 획득

        put(i);             // 임계 구역 (buffer[fill] = i)

        sem_post(&mutex);   // P2.5: 락 해제
        sem_post(&full);    // P3: 슬롯이 찼음을 알림
    }
}

void consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);    // C1: 채워진 슬롯 대기
        sem_wait(&mutex);   // C1.5: 락 획득

        tmp = get();        // 임계 구역 (tmp = buffer[use])

        sem_post(&mutex);   // C2.5: 락 해제
        sem_post(&empty);   // C3: 슬롯이 비었음을 알림
    }
}
```

치명적 오류: 데드락 (DEADLOCK)

- **버그:** 생산자/소비자에서 `wait` 순서를 바꾸는 경우.
  - *잘못된 예:* `sem_wait(&mutex)` **후에** `sem_wait(&empty)` 호출.
- **시나리오:**
  i. 소비자가 실행되어 `mutex` 를 잡습니다.
  ii. 소비자가 `sem_wait(&full)` 을 호출합니다. 버퍼가 비어 있으므로 소비자는 **락을 쥔 채로** 잠듭니다.
  iii. 생산자가 실행되어 `sem_wait(&mutex)` 를 호출합니다. 락은 자고 있는 소비자가 쥐고 있습니다.
  iv. **결과:** 생산자는 `mutex` 를 기다리며 멈추고, 소비자는 생산자가 `full` 신호를 보내길 기다리며 멈춰 있습니다.

버퍼 연산 (원형 버퍼):

- `in = (in + 1) % N`
- `out = (out + 1) % N`

4. 읽기-쓰기 문제 (Readers-Writers Problem)

시나리오:

- 여러 스레드가 공유하는 자료 구조.
- **리더(Readers):** 여러 명이 동시에 읽을 수 있음.
- **라이터(Writers):** 배타적 접근 필요 (다른 라이터나 리더가 없어야 함).

변수:

1. `sem_t lock;` (mutex): `readcount` 변수를 보호. 초기값 = 1.
2. `sem_t writelock;` (rw): 임계 구역(쓰기)을 보호. 초기값 = 1.
3. `int readcount = 0;` : 현재 읽고 있는 리더 수 추적.

구현 (리더 선호 - Reader Preference):

- **라이터 코드:**

```
sem_wait(&writelock);
// 데이터 쓰기
sem_post(&writelock);
```

- **리더 코드:**

```
sem_wait(&lock);        // readcount 업데이트를 위한 락
readcount++;
if (readcount == 1)     // 첫 번째 리더라면...
    sem_wait(&writelock); // ...라이터를 차단함
sem_post(&lock);

// 데이터 읽기 (병렬 읽기가 여기서 일어남)

sem_wait(&lock);        // readcount 업데이트를 위한 락
readcount--;
if (readcount == 0)     // 마지막 리더라면...
    sem_post(&writelock); // ...라이터 진입 허용
sem_post(&lock);
```

기아 현상 (Starvation Issue):

- 이 해결책은 리더를 선호합니다. 리더가 계속 들어오면 `readcount` 가 0이 되지 않습니다.
- 라이터는 `writelock` 을 기다리며 기아 상태(starve)에 빠질 수 있습니다.

5. 식사하는 철학자 문제 (Dining Philosophers Problem)

시나리오: 5명의 철학자, 5개의 포크. 식사하려면 2개의 포크(왼쪽과 오른쪽)가 필요함.

잘못된 해결책 (데드락):

- 알고리즘: 왼쪽 집기, 오른쪽 집기, 먹기.
- **데드락:** 5명의 철학자가 동시에 각자의 **왼쪽** 포크를 집습니다. 모두가 이웃이 쥐고 있는 오른쪽 포크를 기다리게 됩니다.

해결책: 의존성 끊기 (Dependency Breaking):

- **한 명**의 철학자(예: 마지막 번호, 4번)의 포크 집는 순서를 바꿉니다.
- **철학자 0-3:** Wait(왼쪽) → Wait(오른쪽).
- **철학자 4:** Wait(**오른쪽**) → Wait(**왼쪽**).
- **작동 원리:** 순환 대기(circular wait)를 끊습니다. P4는 0번 포크(오른쪽)를 먼저 집으려 합니다. 만약 P0이 이미 그것을 가지고 있다면 P4는 (아무것도 쥐지 않은 채) 대기합니다. 이때 P3가 P4의 왼쪽 포크를 집을 수 있게 됩니다.

6. 고급 개념 & 구현 (Advanced Concepts & Implementation)

스레드 조절 (Thread Throttling)

- **목표:** 메모리 집약적인 구역을 너무 많은 스레드가 동시에 실행하지 못하게 함.
- **해결책:** 세마포어를 **K** (최대 스레드 수)로 초기화.
  - `sem_wait(s)` (구역 진입).
  - `sem_post(s)` (구역 퇴장).
- **입장 제어(Admission Control)** 역할을 합니다.

세마포어 구현하기 (Zemaphores)

락(Lock)과 **조건 변수(Condition Variables, CV)**를 사용하여 세마포어 만들기.

- **구조체:**

```
typedef struct {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;
```

- **Zem_wait:**

```
Mutex_lock(&s->lock);
while (s->value <= 0)         // IF가 아니라 반드시 WHILE 사용
    Cond_wait(&s->cond, &s->lock);
s->value--;
Mutex_unlock(&s->lock);
```

- **Zem_post:**

```
Mutex_lock(&s->lock);
s->value++;
Cond_signal(&s->cond);        // 대기자 하나를 깨움
Mutex_unlock(&s->lock);
```

- **참고:** 이 구현에서는 다익스트라의 정의(음수 값이 대기자 수를 나타냄)와 달리 `value` 가 0 밑으로 내려가지 않습니다.

장단점 요약 (Summary of Pros/Cons)

- **장점:** 단순하고 강력함. 하나의 원시(primitive) 도구로 상호 배제와 조정(스케줄링)을 모두 할 수 있음.
- **단점:** "공유 전역 변수"와 본질적으로 같음(어디서든 접근 가능). 동시성 프로그래밍의 "goto"문과 같음. 버그 발생이 쉬움(데드락, 시그널 누락 등). 디버깅이 어려움.