

제공해주신 자료를 바탕으로 **조건 변수(Condition Variables, CVs)**, **생산자/소비자 문제**, 그리고 **구현 세부 사항**에 대해 정리한 핵심 요약 노트입니다.

빠른 회독과 기술적인 이해를 돕기 위해 코드 로직과 아키텍처의 함정에 집중하여 구성했습니다.

1. 핵심 개념: 조건 변수 (Condition Variables)

왜 필요한가?

- **Spinning은 낭비다:** 조건이 충족될 때까지 반복문(`while(done == 0);`)을 돌며 기다리는 것은 CPU 사이클을 비효율적으로 소모합니다.
- **목표:** 특정 조건이 참이 될 때까지 스레드를 **재워(Sleep)** 자원 낭비를 막는 것입니다.

정의

- **조건 변수**는 실행 상태가 원하지 않는 상태일 때 스레드가 스스로를 대기시킬 수 있는 ****명시적인 큐(Queue)****입니다.
- 상태가 변경되었을 때 다른 스레드가 대기 중인 스레드를 **깨울(Wake)** 수 있게 해줍니다.

연산 (API)

1. `wait(cond_t *c, mutex_t *m)` :
 - **반드시** 뮤텍스(Mutex)를 잠근 상태에서 호출해야 합니다.
 - 원자적으로(Atomically) 락을 ****해제(Release)****하고 스레드를 **재웁니다(Sleep)**.
 - 깨어날 때, 리턴하기 전 다시 락을 ****획득(Re-acquire)****합니다.
2. `signal(cond_t *c)` :
 - 대기 중인 스레드 중 **하나**를 깨웁니다. 대기 중인 스레드가 없으면 신호는 소실됩니다(세마포어와 다른 점).
3. `broadcast(cond_t *c)` :
 - 대기 중인 **모든** 스레드를 깨웁니다.

2. "스레드 조인(Thread Join)" 문제 (코드의 진화)

목표: 부모 스레드가 자식 스레드가 끝날 때까지 기다려야 함.

시도 1: 상태 변수 없음 (고장남)

```
void child() {
    lock(&m); signal(&c); unlock(&m);
}
void parent() {
    lock(&m); wait(&c, &m); unlock(&m);
}
```

- **버그:** 부모가 wait 를 호출하기 전에 자식이 실행되면, 신호가 빈 큐로 보내져 소실됩니다. 부모는 영원히 대기하게 됩니다.
- **교훈:** 실제 조건을 추적할 상태 변수(예: int done)가 필요합니다.

시도 2: 락(Lock) 없음 (고장남)

```
void parent() {
    if (done == 0)
        // 여기서 인터럽트 발생
        wait(&c);
}
```

- **경쟁 상태(Race Condition):** 부모가 done 을 확인(0임)합니다. wait 를 호출하기 직전에 인터럽트가 걸립니다. 자식이 실행되어 done=1 로 설정하고 신호를 보냅니다(소실됨). 부모가 다시 실행되면 영원히 잠들게 됩니다.
- **교훈:** 조건을 확인하고 잠드는 과정은 원자적이어야 합니다(뮤텍스로 보호).

시도 3: 올바른 해결책

```
mutex_t m; cond_t c; int done = 0;

void child() {
    lock(&m);
    done = 1;
    signal(&c);
    unlock(&m);
}

void parent() {
    lock(&m);
    while (done == 0) // 항상 while을 사용할 것!
        wait(&c, &m);
    unlock(&m);
}
```

- **프로토콜:** 락 획득 -> 상태 확인 -> 대기(필요 시) -> 락 해제

3. 생산자/소비자 (유한 버퍼) 문제

시나리오: 생산자는 데이터를 버퍼에 넣고, 소비자는 꺼냅니다.

제약 사항:

- 버퍼가 **가득 차면** 생산자 대기.
- 버퍼가 **비면** 소비자 대기.
- 공유 버퍼 count 에 대한 동기화된 접근 필요.

메사 시맨틱 (Mesa Semantics - 중요 개념)

- **정의:** 스레드에게 신호를 보내는 것은 상태가 변경되었을 수 있다는 **힌트**일 뿐입니다. 스레드가 실행될 때 그 상태가 여전히 유효하다는 **보장은 없습니다**.
- **규칙:** 조건을 확인할 때는 **if** 문 대신 항상 **while 루프**를 사용하세요.
 - *이유:* 스레드가 깨어났을 때, 다른 스레드가 중간에 끼어들어 상태를 바꿔버렸을 수 있으므로 조건을 **다시 확인**해야 합니다.
- **가짜 기상(Spurious Wakeups):** 신호 없이도 스레드가 깨어날 수 있는 OS의 특성 때문에 **while 루프**를 통한 재확인이 필수적입니다.

해결책의 진화

버전 1: 고장남 (단일 CV + if)

- 코드: `if (count == 0) wait(...)` .
- 시나리오:
 - i. 소비자 1(C1)이 잠듦 (버퍼 빈).
 - ii. 생산자(P)가 버퍼를 채우고 C1에 신호. C1은 **준비(Ready)** 상태 (아직 실행 안 됨).
 - iii. 소비자 2(C2)가 끼어들어 실행하고 데이터를 가로챈. 버퍼는 다시 빈.
 - iv. C1 실행. `if` 를 썼기 때문에 버퍼가 찼다고 가정하고 `get()` 실행 -> 실패(Assertion Error).
- 수정: `if` 를 `while` 로 변경.

버전 2: 고장남 (단일 CV + while)

- 코드: P와 C가 **같은** 조건 변수 `cond` 에서 대기.
- 시나리오 ("모두가 잠드는" 버그):
 - i. C1과 C2 모두 잠듦 (버퍼 빈).
 - ii. P 실행, 버퍼 채움, C1 깨움. P가 다시 채우려다 버퍼 가득 참 -> P 잠듦.
 - iii. C1 실행, 데이터 소비. 버퍼 빈.
 - iv. C1이 누군가를 깨워야 함. `signal()` 호출.
 - v. **치명적 실패**: C1이 P 대신 실수로 **C2**를 깨움.
 - vi. C2 기상, 버퍼 확인(빈), 다시 잠듦.
 - vii. 결과: P, C1, C2 모두 영원히 잠듦.
- 교훈: 하나의 CV를 사용하면 잘못된 "유형"의 스레드를 깨울 수 있습니다.

버전 3: 정답 (두 개의 CV + while)

- 구조: 두 개의 분리된 CV 사용: `empty` (빈 공간)와 `fill` (채워진 데이터).
- 로직:
 - **생산자**: `empty` 에서 대기(공간 기다림), `fill` 에 신호(데이터 있음 알림).
 - **소비자**: `fill` 에서 대기(데이터 기다림), `empty` 에 신호(공간 있음 알림).
- 코드 스니펫:

```

void producer() {
    lock(&m);
    while (count == MAX)
        wait(&empty, &m); // 공간 대기
    put_data();
    signal(&fill);          // 소비자 깨움
    unlock(&m);
}

void consumer() {
    lock(&m);
    while (count == 0)
        wait(&fill, &m); // 데이터 대기
    get_data();
    signal(&empty);        // 생산자 깨움
    unlock(&m);
}

```

4. 커버링 컨디션 (Covering Conditions - 메모리 할당 예제)

문제점

- 스레드 A는 100바이트 필요. 스레드 B는 10바이트 필요. 현재 0바이트 남음. 둘 다 잠듦.
- 스레드 C가 50바이트 해제. signal() 호출.
- 만약 신호가 스레드 A를 깨우면: A 기상, $50 < 100$ 확인, 다시 잠듦. **B는 실행될 수 있었음에도($50 > 10$) 깨어나지 못함.**

해결책: broadcast()

- 스레드 C가 pthread_cond_broadcast() 호출.
- **결과:** 대기 중인 **모든** 스레드를 깨움. A 실행(실패, 잠듦). B 실행(성공).
- **트레이드오프:** 성능 비용(Thundering herd 문제)이 있지만, **어떤** 스레드의 조건이 충족되었는지 모를 때는 정확성을 위해 필수입니다.

5. 세마포어 vs 뮤텝스/CV

비교

- 세마포어는 뮤텝스 + CV와 동등한 표현력을 가집니다.
- **차이점:** CV의 `signal()` 은 대기자가 없으면 소실되지만, 세마포어의 `post()` 는 저장됩니다(값을 증가 시킴).

CV로 세마포어 구현하기 (시험 출제 패턴)

```
typedef struct {
    int value;
    mutex_t m;
    cond_t c;
} sem_t;

void sem_wait(sem_t *s) {
    lock(&s->m);
    while (s->value <= 0) // 0이면 대기
        wait(&s->c, &s->m);
    s->value--;
    unlock(&s->m);
}

void sem_post(sem_t *s) {
    lock(&s->m);
    s->value++;           // 상태 증가
    signal(&s->c);         // 대기자 깨움
    unlock(&s->m);
}
```

6. OS 구현 세부 사항 (xv6)

슬립락(Sleeplock) 구조

- `locked` 필드(볼리언 플래그)를 보호하기 위해 **스핀락(spinlock)**(`lk`)을 사용합니다.
- 디스크 I/O와 같이 스레드가 CPU를 양보(Yield)해야 하는 긴 잠금에 사용됩니다.

`sleep(void *chan, struct spinlock *lk)`

1. 프로세스 락 `p->lock` 획득.
2. 인자로 받은 `lk` (뮤텝스)를 안전하게 해제.
3. 프로세스 상태 설정: `p->state = SLEEPING` .
4. 채널 설정: `p->chan = chan` .

5. sched() 호출하여 문맥 전환(Context Switch).
6. 리턴 시(깨어난 후): p->chan 초기화, lk 다시 획득.

wakeup(void *chan)

1. 프로세스 테이블 순회.
2. 대상 프로세스 p 잠금.
3. 만약 p->state == SLEEPING **AND** p->chan == chan 이면:
 - p->state = RUNNABLE 로 설정.
4. p 잠금 해제.

7. 시험 문제 풀이를 위한 핵심 규칙

1. **대기 규칙 (Wait Rule):** wait 호출 시 반드시 락을 잡고 있어야 합니다. OS가 원자적으로 락을 해제합니다.
2. **신호 규칙 (Signal Rule):** signal 호출 시 락을 잡고 있는 것이 가장 간단하고 안전한 방법입니다.
3. **루프 규칙 (Loop Rule):** 항상 while (condition) 을 사용하고, 절대 if (condition) 을 사용하지 마세요.
4. **방향 규칙 (Direction Rule):** 생산자/소비자 문제에서는 신호를 반대 그룹으로 보내기 위해 **두 개의 CV**를 사용하세요 (생산자 -> 소비자, 소비자 -> 생산자).
5. **브로드캐스트 규칙 (Broadcast Rule):** 같은 변수(예: 메모리 크기)에 대해 여러 스레드가 서로 다른 조건을 기다릴 때는 broadcast 를 사용하세요.