Here is a comprehensive study guide covering the material from the provided slides and textbook chapters. It is structured for rapid review, focusing on implementation details, code logic, and architectural mechanisms.

Operating Systems: Condition Variables & Synchronization

Study Guide & Fast-Track Notes

1. The Core Concept: Condition Variables (CV)

Definition

- **What is it?** An explicit queue that threads can put themselves on to wait for a specific condition to become true (e.g., "buffer is not empty", "child has finished").
- **Role:** Allows a thread to sleep (yielding CPU) instead of busy-waiting (spinning) when execution cannot proceed.
- **Requirement:** Always used in conjunction with a **Mutex (Lock)**. The mutex protects the shared state associated with the condition.

Operations

1. **`wait(cond_t *cv, mutex_t *lock)`**
    - Assumes `lock` is **held** when called.
    - **Atomically:** Releases the `lock` and puts the calling thread to sleep.
    - **On Return:** Re-acquires the `lock` before returning to the caller.
2. **`signal(cond_t *cv)`**
    - Wakes up **one** waiting thread (if any).
    - If no threads are waiting, the signal is **lost** (no history/memory, unlike semaphores).
3. **`broadcast(cond_t *cv)`**
    - Wakes up **all** waiting threads.

2. Design Patterns & The "Join" Problem

Scenario

A parent thread waits for a child thread to complete execution.

The Incorrect Approaches

- **Spin-based:** `while(done == 0);` -> Wastes CPU cycles.
- **No State Variable:** Waiting only on the CV without a boolean flag (`done`).
  - *Race Condition:* If the child runs and exits *before* the parent calls `wait` , the `signal` is lost. The parent will wait forever for a signal that already happened.
- **No Lock:** Calling `wait` without holding a lock.
  - *Race Condition:* Parent checks `done` , sees 0. Context switch happens. Child runs, sets `done=1` , signals. Parent resumes and calls `wait` . Parent sleeps forever.

The Correct Pattern

```
// State
int done = 0;
pthread_mutex_t m;
pthread_cond_t c;

void thread_exit() {
    pthread_mutex_lock(&m);
    done = 1;               // 1. Change State
    pthread_cond_signal(&c); // 2. Signal
    pthread_mutex_unlock(&m);
}

void thread_join() {
    pthread_mutex_lock(&m);
    while (done == 0) {     // 3. Check State
        pthread_cond_wait(&c, &m); // 4. Wait (releases lock)
    }
    pthread_mutex_unlock(&m);
}
```

Key Takeaway: The CV is used to *wait* for a change, but the **state variable** (`done`) records *what* happened.

3. The Producer/Consumer (Bounded Buffer) Problem

Setup

- **Producer:** Puts items into a fixed-size buffer.
- **Consumer:** Removes items from the buffer.
- **Constraints:** Producer waits if buffer is full; Consumer waits if buffer is empty.

Evolution of the Solution (Code Analysis)

Attempt 1: `if` Statement (Broken)

```
if (count == 0)
    pthread_cond_wait(&c, &m);
// consume item
```

- **The Bug (Mesa Semantics):**
  1. Consumer T_{c1} wakes up because Producer signaled.
  2. T_{c1} moves to Ready queue.
  3. Another Consumer T_{c2} enters, acquires lock, and consumes the item *before* T_{c1} runs.
  4. T_{c1} runs, assumes buffer has data (because it woke up), and tries to dequeue from an empty buffer. **Error.**
- **Rule: Always use `while`, never `if`.** Signaling is a *hint* that state changed, not a guarantee.

Attempt 2: Single CV with `while` (Broken)

```
// Consumer
while (count == 0)
    pthread_cond_wait(&cond, &m);
consume();
pthread_cond_signal(&cond); // Could wake a consumer!

// Producer
while (count == MAX)
    pthread_cond_wait(&cond, &m);
produce();
pthread_cond_signal(&cond); // Could wake a producer!
```

- **The Bug:**
    i. Consumers T_{c1}, T_{c2} sleep (buffer empty). Producer T_p sleeps (buffer full).
    ii. T_p wakes T_{c1}. T_p sleeps.
    iii. T_{c1} runs, consumes, and calls `signal()` .
    iv. **Critical Error:** The signal wakes T_{c2} (another consumer) instead of T_p.
    v. T_{c2} sees buffer empty, goes back to sleep.
    vi. Everyone is sleeping. **Deadlock.**

Attempt 3: Two CVs (Correct Solution)

Use separate channels for separate conditions so threads don't wake the "wrong" type of peer.

- **`empty` CV:** Signaled when a slot becomes empty (wakes Producer).
- **`fill` CV:** Signaled when a slot becomes full (wakes Consumer).

```
void *producer(void *arg) {
    pthread_mutex_lock(&m);
    while (count == MAX)            // Check if full
        pthread_cond_wait(&empty, &m); // Wait for space
    put_data(i);
    pthread_cond_signal(&fill);     // Wake consumer
    pthread_mutex_unlock(&m);
}

void *consumer(void *arg) {
    pthread_mutex_lock(&m);
    while (count == 0)              // Check if empty
        pthread_cond_wait(&fill, &m);  // Wait for data
    get_data();
    pthread_cond_signal(&empty);    // Wake producer
    pthread_mutex_unlock(&m);
}
```

4. Covering Conditions (Memory Allocation)

The Problem

- Thread A needs 100 bytes (`wait` loop: `while (bytes < 100)`).

- Thread B needs 10 bytes (`wait` loop: `while (bytes < 10)`).
- Thread C frees 50 bytes.
- If C calls `signal()` , it might wake A.
- A wakes up, sees 50 < 100, goes back to sleep.
- B is still sleeping, even though 50 > 10. The resource is wasted.

The Solution: Broadcast

- Replace `signal()` with `broadcast()` .
- Wake **all** threads.
- A wakes, checks, sleeps.
- B wakes, checks, proceeds.
- **Trade-off:** Correctness vs. Performance (context switch overhead). This is called a **Covering Condition**.

5. Kernel Implementation Details (Xv6)

How are these primitives implemented at the OS level?

Structure: `sleeplock`

A "sleep lock" allows a process to yield the CPU while holding a "lock" (unlike a spinlock where you must not yield).

```
struct sleeplock {
    uint locked;        // Is the lock held?
    struct spinlock lk; // Protects this structure
    // ... pid, name
};
```

The `sleep` mechanism

Used inside the kernel to wait for IO or locks.

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

1. **Safety dance:** The process must hold `p->lock` (process lock) to change its state to `SLEEPING` atomically so no `wakeup` is missed.

2. `acquire(&p->lock);`
3. `release(lk);` (The lock passed in, e.g., the directory lock, must be released so others can use it while we sleep).
4. `p->chan = chan;` (Set the "channel" ID we are waiting on).
5. `p->state = SLEEPING;`
6. `sched();` (Switch to scheduler - context switch).
7. `p->chan = 0;` (Upon return).
8. `release(&p->lock);`
9. `acquire(lk);` (Re-acquire the original lock before returning to caller).

The `wakeup` mechanism

`void wakeup(void *chan)`

1. Loop through the process table (`proc[NPROC]`).
2. `acquire(&p->lock);`
3. If `p->state == SLEEPING && p->chan == chan` :
   - `p->state = RUNNABLE;`
4. `release(&p->lock);`

6. Summary & Rules of Thumb

Mesa Semantics (Standard)

- **Behavior:** When a thread is woken, it is moved to the ready queue. By the time it runs, the condition may no longer be true.
- **Implication: Must use `while` loops.**
- **Spurious Wakeups:** Threads may wake up without any signal (OS implementation artifacts). `while` loops handle this naturally.

Hoare Semantics (Theoretical)

- **Behavior:** When a thread is signaled, the signaling thread pauses, and the ownership of the lock is passed *directly* to the waiting thread.
- **Implication:** The condition is guaranteed to be true upon return from `wait` . `if` statements would suffice. (Rarely implemented in real systems).

Comparisons

- **Spinlocks:** Good for short waits, multicore. Wastes CPU.
- **Mutex + CV:** Good for long waits. Yields CPU.
- **Semaphores:**
  - `semaphore = mutex + CV + counter`.
  - Semaphore `signal` increments state (has history).
  - CV `signal` is lost if no one is waiting (stateless).

Code Checklist for Exams

1. Is the state variable checked in a `while` loop?
2. Is the lock held before calling `wait`?
3. Is the lock held when modifying the state variable?
4. Is the lock re-acquired immediately after `wait` returns (automatic in API)?
5. Are distinct conditions utilizing distinct CVs (e.g., `empty` vs `fill`)?