Here is a comprehensive, high-density study guide based on the provided lecture notes (`ads.pdf`) and the final exam content (`2024-1-final.pdf`).

This guide is structured to be internally complete, explaining concepts from the ground up so you do not need the textbook. It emphasizes system architecture and code logic.

Systems Software & Architecture: Comprehensive Study Notes

Part 1. Concurrency: Monitors

Source Material: `ads.pdf`

1.1. Monitor Fundamentals

- **Definition:** A high-level synchronization construct that encapsulates shared data, operations on that data, and synchronization logic.
- **Encapsulation:**
  - **Shared Data:** Private to the monitor; cannot be accessed directly from outside.
  - **Procedures:** Entry points. Only one process/thread can be active inside a monitor procedure at any time (Implicit Mutual Exclusion).
  - **Entry Queue:** Threads trying to call a monitor procedure block here if the monitor is busy.

1.2. Condition Variables (CV)

- **Purpose:** Allow threads to wait for specific states (e.g., "buffer not full") inside the monitor.
- **Mechanism:**
  - `wait(c)` : Releases the monitor lock and puts the thread to sleep on condition c.
  - `signal(c)` : Wakes up **one** thread waiting on c. If no one is waiting, the signal is lost (stateless, unlike semaphores).
  - `broadcast(c)` : Wakes up **all** threads waiting on c.

1.3. Monitor Semantics: Hoare vs. Mesa

The crucial difference lies in what happens immediately after a thread issues a `signal()` .

Feature	Hoare Semantics (Signal-and-Wait)	Mesa Semantics (Signal-and-Continue)
Behavior	Signaler suspends; Waiter runs *immediately*.	Signaler continues; Waiter is placed on ready queue.
Guarantee	The condition is guaranteed to be true when the waiter resumes.	The condition is a *hint*. It might have changed before the waiter runs.
Code Pattern	`if (condition) wait(c);`	`while (condition) wait(c);`
Pros/Cons	Stronger logic, fewer context switches needed for checking.	Easier to implement, supports `broadcast` , more efficient context switching.

1.4. Implementation: Monitor using Semaphores

Implementing a Monitor using lower-level Semaphores is complex because we must handle the "handoff" of the lock between signalers and waiters.

Data Structures:

- `mutex` (Sem=1): Enforces mutual exclusion for entry.
- `next` (Sem=0): Queue for threads that signaled and are now waiting for the woken thread to finish.
- `next_count` (int): Number of threads waiting on `next` .
- `x.sem` (Sem=0): Queue for condition variable x.
- `x.count` (int): Number of threads waiting on x.

Procedure Entry/Exit Logic:

```
wait(mutex);
    // ... body of procedure ...
if (next_count > 0)
    signal(next); // Prioritize threads that signaled and suspended
else
    signal(mutex); // Allow new threads from entry queue
```

Condition Wait (`wait(c)`):

```
x.count++;
if (next_count > 0) signal(next); // Release lock to internal waiter
else                signal(mutex); // Release lock to external entry
wait(x.sem);                       // Sleep
x.count--;
```

Condition Signal (`signal(c)`):

```
if (x.count > 0) {
    next_count++;
    signal(x.sem); // Wake up the waiter
    wait(next);    // Suspend self (Hoare semantics behavior emulation)
    next_count--;
}
```

Part 2. Synchronization Algorithms & Barriers

Source Material: `2024-1-final.pdf`

2.1. The Bakery Algorithm (Critical Section)

Ensures mutual exclusion without hardware atomic instructions.

- **Concept:** Threads take a "ticket" number. The thread with the lowest number enters the Critical Section (CS).
- **Variables:**
    - `choosing[N]` : Boolean, true while a thread is picking a number.
    - `number[N]` : Integer, the ticket number (0 = inactive).
- **Logic (Acquire):**
    i. Set `choosing[me] = true` .
    ii. `number[me] = max(number[0]...number[N-1]) + 1` .
    iii. Set `choosing[me] = false` .
    iv. Loop through all other threads (j):
        - Wait while `choosing[j]` is true (prevent reading unstable numbers).
        - Wait while `number[j] != 0` AND (`number[j] < number[me]` OR tie-break with PID).
- **Optimization (Fetch-and-Add):**

  - If hardware supports atomic `fetch_and_add`, the `choosing` array is unnecessary because ticket assignment is instantaneous and unique.

2.2. Barrier Synchronization

A mechanism where threads stop at a point until N threads arrive.

Implementation 1: Using Semaphores (2 Threads)

```
sema_t s[2]; // Init to 0
void barrier2(int me) {
    int other = 1 - me;
    signal(s[other]); // Tell the other I arrived
    wait(s[me]);      // Wait for the other to signal me
}
```

Implementation 2: N-Thread Barrier (Mesa Monitor style)

- **Idea:** Last thread to arrive wakes everyone else.
- **State:** `count` (arrived threads), `mutex`, `cond`.

```
mutex_lock(&m);
count++;
if (count == N) {
    cond_broadcast(&c); // Wake everyone
    count = 0;          // Reset for next phase (reusable barrier)
} else {
    while (count < N)   // Mesa semantics requires while
        cond_wait(&c, &m);
}
mutex_unlock(&m);
```

Part 3. System Architecture & Costs

Source Material: `2024-1-final.pdf`

3.1. Operation Costs (Low to High)

1. **Branch/Jump:** CPU internal instruction (Cheapest).
2. **Procedure Call:** Push stack frame, jump, return.
3. **User-level Thread Switch:** Save registers, switch stack (User space only).
4. **Kernel-level Thread Switch:** Trap to kernel, scheduler runs, reload context (Most expensive).

3.2. Shared vs. Private Resources in Threads

- **Shared (Process-wide):** Heap memory, Global variables, Code memory, Open files (file descriptors), Parent PID, Signal handlers.
- **Private (Per-Thread):** Stack memory, Register values (PC, SP), Thread Priority, Errno.

Part 4. Storage & File Systems

Source Material: `2024-1-final.pdf`

4.1. Hardware Terminology

- **Spindle:** The motor spinning the platters.
- **Track Buffer:** Built-in RAM cache on the HDD to smooth speed differences.
- **Read Disturbance (SSD):** Reading a page frequently can cause bit flips in neighboring pages within the same block.

4.2. File System Layouts (FAT & FFS)

- **FFS (Fast File System):** Solved disk rotational latency by skipping blocks (interleaving) during layout so the head is over the correct block after processing the previous one.
- **fsync(fd):** System call that forces dirty data to disk for a specific file (persistence guarantee).

4.3. Directory Entry & Inode Math (Exam Logic)

- **Scenario:** Opening `/usr/bin/ls` .
- **Path Traversal:**
    i. Read Root (`/`) Inode $\longrightarrow$ Read Data (find `usr`).
    ii. Read `usr` Inode $\longrightarrow$ Read Data (find `bin`).
    iii. Read `bin` Inode $\longrightarrow$ Read Data (find `ls`).

    iv. Read `ls` Inode.
- **Cost Calculation:**
    - If `/usr` and `/usr/bin` have 100 entries, and data blocks are 4KB.
    - Cost depends on how many blocks the directory entries occupy.
    - *Formula:* $\lceil (\text{entries} \times \text{entry\_size}) / \text{block\_size} \rceil$.

4.4. Full Path Indexing

- **Concept:** A single massive directory ("Flat namespace") mapping full strings (`/usr/bin/ls`) directly to Inode numbers.
- **Time Complexity:**
    - **Normal FS:** $O(d \times n)$ where d is depth, n is entries per directory.
    - **Full Path:** $O(1)$ (Hash Map) or $O(\log N)$ (B-Tree) relative to the total number of files N.
- **Optimization:** Use a Bloom Filter or Hash Table to check existence quickly before searching the index.
- **Drawbacks:**
    - Hard Link complexity (aliasing).
    - Rename operations are expensive (must update prefixes of all children).
    - Large index size requires memory management.
- **Symbolic Links:** Store the target path as data. On lookup, if the result is a symlink, restart lookup with the new path string.

Part 5. Code Patterns to Memorize

5.1. Spin Lock

```
while (TestAndSet(&lock) == 1)
    ; // Spin (waste CPU)
// Critical Section
lock = 0;
```

- *Requirement:* Preemptive scheduler (otherwise single core hangs forever).

5.2. Atomic Fetch-and-Add (Lock-Free Logic)

Used to replace complex locking (like in Bakery algorithm).

```c
int fetch_and_add(int *v, int a) {
    int old = *v;
    *v = old + a;
    return old;
}
```

5.3. Monitor "Wait" Loop

Always use a while loop when using Mesa monitors (pthreads, Java, C++).

```c
// Correct
while (buffer_is_empty)
    wait(not_empty_cv);

// Incorrect (Race condition possible)
if (buffer_is_empty)
    wait(not_empty_cv);
```

5.4. Semaphore vs Monitor "Signal"

- **Semaphore `signal()`:** Increments value. If no one waits, value persists (history).
- **Monitor `signal()`:** Wakes one waiter. If no one waits, signal is lost (no history).