Operating Systems Concurrency Exam Guide (c. Exercises)

This guide is designed to help you master the concurrency problems (typically labeled `c.1`, `c.2`, etc.) in your "Sistemi Operativi" exam. We will break down the three main paradigms—Semaphores, Monitors, and Message Passing—using the classic problems from your lecture notes and detailed walkthroughs of past exam questions.

A crucial part of this exam is adhering to a strict set of rules, which the professor calls the **"Ennalogo"**. We will cover these rules at the end, as following them is non-negotiable for passing.

Part 1: The Foundation - Why We Need Concurrency Control

All concurrency problems stem from a single issue: **race conditions**. This happens when multiple processes access and manipulate shared data, and the final result depends on the unpredictable order of their execution.

Your lecture notes (slide 17) provide the canonical example:

```
// Shared variable
int totale = 100;

// Function called by two processes
void modifica(int valore) {
    totale = totale + valore;
}

// Process P1 calls modifica(+10)
// Process P2 calls modifica(-10)
```

We expect the final result to be 100. However, the single C statement compiles to multiple assembly instructions:

1. `lw $t0, totale` (Load `totale` into a register)
2. `add $t0, $t0, $a0` (Add the value to the register)
3. `sw $t0, totale` (Store the register's new value back into `totale`)

If Process P1 is interrupted after step 2, P2 might run completely, setting `totale` to 90. When P1 resumes, it will execute step 3, overwriting the result with its own outdated value of 110. The result is wrong.

The part of the code that accesses shared resources is called a **Critical Section (CS)**. The goal of all concurrency paradigms is to ensure **mutual exclusion**: only one process can be inside its critical section at any given time.

Part 2: The Core Paradigms

The exam will require you to use one of three high-level paradigms to solve synchronization problems.

2.1 Monitors

Monitors are the most frequently tested paradigm in your exams. They are high-level synchronization constructs that simplify concurrent programming.

Key Concepts (from Lecture Notes):

- **Encapsulation:** A monitor is like a class or an object. It bundles shared data (variables) with the procedures that operate on that data.
- **Implicit Mutual Exclusion:** Only one process can be executing a procedure *inside* the monitor at any time. You do **not** need to use a mutex; the monitor handles it automatically.
- **Condition Variables:** For complex synchronization (e.g., "wait until the buffer is not full"), monitors use `condition` variables. They are not boolean flags; they are queues of waiting processes.
  - `c.wait()`: A process calling this is immediately blocked and placed in the queue for condition `c`. Crucially, it **releases the monitor lock**, allowing another process to enter.
  - `c.signal()`: If the queue for condition `c` is not empty, this unblocks **one** waiting process.
- **Signal and Urgent Wait (Hoare-style):** This is the policy used in your course. When Process A calls `c.signal()` and unblocks Process B, Process A is immediately suspended and put on a special "urgent" queue. Process B gets the monitor lock and runs. When B finishes or calls `wait()`, A is resumed from the urgent queue. This ensures the condition B was waiting for is still true when it resumes.

Common Exam Problem Patterns & Solutions

Monitors are excellent for managing shared resources with complex state-dependent rules. Let's analyze past exam problems.

Example 1: Strict Alternation (from `scritto-2023-02-15-soluzione.txt`)

Problem (`redblack`): Implement a monitor with a procedure `rb(color, value)`. Processes must alternate strictly between red and black. A call to `rb` should block if the last completed call had the same color. The function returns the average of values for that color.

Analysis: This is a classic state-based synchronization problem. We need to remember the color of the last process that was allowed to proceed.

- **State:** A single integer `last_color` and variables to calculate the mean (`sum`, `count`).
- **Condition:** A single condition variable is enough. A red process waits if `last_color` is red. When a black process finishes, it signals this condition to wake up a waiting red process (and vice-versa). However, since there are two groups of waiters (red and black), it's cleaner to use one condition variable per color.

Solution:

```
monitor redblack {
    // State variables
    int last = -1; // -1: initial state, 0: red, 1: black
    double sum[2] = {0.0, 0.0};
    int count[2] = {0, 0};

    // One condition variable for each color to wait on
```

```
    condition ok2col[2];

    // The single entry procedure
    double rb(int color, double value) {
        // Wait if the last process had the same color
        if (color == last) {
            ok2col[color].wait();
        }

        // I have the lock and my turn is valid.
        // Update the state for the *next* process.
        last = color;

        // Update my color's statistics
        sum[color] += value;
        count[color]++;
        double mean = sum[color] / count[color];

        // Signal the *other* color that it's their turn
        ok2col[1 - color].signal();

        return mean;
    }
}
```

Takeaway: For problems requiring alternation, use a state variable to track whose turn it is. Use condition variables to make processes wait when it's not their turn.

Example 2: Group Synchronization / Barrier (from `scritto-2023-01-20-soluzione.txt`)

Problem (`fullbuf`): A monitor has `add(value)` and `get()`. The first `MAX` calls to `add` must block. `get` must wait until more than `MAX` `add` calls have been made (`Na > MAX`). When `get` runs, it returns the sum of values from the waiting `add` processes and unblocks the *first* process that was waiting on `add`. The system must ensure that after `get` completes, `Na >= MAX` is still true.

Analysis: This is a complex barrier problem. `add` processes accumulate until a threshold is passed, at which point `get` can act on them.

- **State:** We need a counter for waiting `add` processes (`Na`), a queue or list to store their values, and a way to signal them individually.
- **Individual Signaling:** Since `get` unblocks a *specific* (`FIFO`) `add` process, a single `condition` variable is not enough. We need a queue of condition variables, one for each waiting `add` process.
- **Logic:**
  - `add`: Increments `Na`. If `Na <= MAX`, it creates a new condition variable for itself, adds it to a queue, and waits on it. It also stores its `value` somewhere. If `Na > MAX` (meaning `get` has already run and unblocked one), it doesn't wait. It must also signal any waiting `get` process.
  - `get`: Waits until `Na > MAX`. When it proceeds, it calculates the sum, signals the first `add` process in the queue, decrements `Na`, and returns.

Simplified Solution Sketch (based on `scritto-2024-06-25 choicesem` which has a similar pattern):

```
monitor advanced_barrier {
    int Na = 0; // Number of waiting add() processes
    Queue<int> values;
    Queue<Condition> waiting_adds; // A queue of condition variables!

    Condition get_can_run; // For get() to wait on

    void add(int value) {
        Na++;
        values.enqueue(value);

        // Signal a potentially waiting get()
        if (Na > MAX) {
            get_can_run.signal();
        }

        // Create a personal condition and wait on it
        Condition my_turn = new Condition();
        waiting_adds.enqueue(my_turn);
        my_turn.wait();
    }

    int get() {
        // Wait until enough add() processes are ready
        if (Na <= MAX) {
            get_can_run.wait();
        }

        // Now Na > MAX is guaranteed.
        int sum = 0;
        // ... calculate sum from 'values' ...

        // Unblock the first waiting add() process
        Condition first_add = waiting_adds.dequeue();
        first_add.signal();
        Na--;

        return sum;
    }
}
```

Takeaway: If you need to signal a *specific* process (e.g., the Nth one, or one based on priority), you likely need a **data structure of condition variables** (queue, list, array). This is a powerful and frequently tested pattern.

2.2 Semaphores

Semaphores are a lower-level primitive than monitors. They are essentially counters with atomic operations.

Key Concepts (from Lecture Notes):

- A semaphore is an integer variable.
- `P(S)` or `wait(S)`:
    1. Decrements the semaphore value `S`.
    2. If the value becomes negative, the process blocks. (In another common definition, `if S > 0, S--` else block). Your course uses the `nP <= nV + init` invariant, meaning the semaphore value cannot be negative. So the logic is: "wait until `S > 0`, then decrement `S`."
- `V(S)` or `signal(S)`:
    1. Increments the semaphore value `S`.
    2. If one or more processes are blocked on `S`, unblock one (usually FIFO, as per your "fair" semaphores rule).
- **Crucial Difference from Monitors:** `V()` operations are "remembered". If you call `V()` on a semaphore with no one waiting, its value still increases. When a process later calls `P()`, it might proceed without blocking. A monitor's `signal()` with no one waiting has no effect.

Common Exam Problem Patterns & Solutions

Semaphores are great for managing access to a pool of N resources or for simple signaling between processes.

Example 1: The `wait4` Problem (from `scritto-2023-01-18`)

Problem: Implement a `wait4` function using semaphores. The first three processes to call it must block. The fourth process must unblock all three and then proceed. The eighth process does the same for the 5th, 6th, and 7th, and so on. Do not use complex data structures, only counters and semaphores. Use "passing the baton" to ensure correctness.

Analysis: This is a barrier synchronization problem, perfect for semaphores.

- **State:** We need a shared counter `count` to track how many processes have arrived.
- **Mutual Exclusion:** Access to `count` must be protected by a mutex semaphore.
- **Blocking:** Processes that need to wait will do so on a second semaphore, let's call it `gate`.
- **Passing the Baton:** This is key. The fourth process must not just leave the critical section and let other new processes sneak in. After releasing the three waiters, it must also release the mutex, but in a way that gives priority to an already-waiting process. The standard `V(mutex)` does this if the semaphore is fair (FIFO).

Solution:

```
// Shared variables
int count = 0;
Semaphore mutex = new Semaphore(1); // For protecting count
Semaphore gate = new Semaphore(0);  // For blocking processes 1, 2, 3

void wait4() {
```

```
        mutex.P(); // Enter critical section

        count++;
        if (count < 4) {
            // I am one of the first three. I must wait.
            mutex.V(); // Release mutex so others can enter
            gate.P();  // Wait on the gate
        } else {
            // I am the fourth process.
            // Unblock the three waiting processes.
            gate.V();
            gate.V();
            gate.V();

            // Reset the counter for the next group of four.
            count = 0;

            // I can proceed. Release the mutex for the next process.
            mutex.V();
        }
    }
```

Takeaway: Use a mutex semaphore to protect shared counters. Use a second semaphore initialized to 0 to act as a gate or barrier for processes that need to wait.

2.3 Message Passing

This paradigm avoids shared memory entirely. Processes run in their own private address spaces and communicate by sending and receiving messages. Synchronization is a side effect of communication.

Key Concepts (from Lecture Notes):

- **Primitives:** `send(message, destination)` and `receive(source)`.
- **Asynchronous Send:** The `send` operation is non-blocking. The sender sends the message and continues its execution immediately. The OS buffers the message. This is the most common type in your exams.
- **Blocking Receive:** The `receive` operation is blocking. If no message from the specified source is available, the receiver waits until one arrives.
- **Synchronous Send:** The sender blocks until the receiver has received the message. This is called a **rendezvous**. You are often asked to build synchronous services on top of asynchronous ones.

Common Exam Problem Patterns & Solutions

Message passing problems are often about designing a communication protocol between processes.

Example 1: Building Synchronous from Asynchronous (from `scritto-2020-01-15`)

Problem (`mulsend`/`mulrecv`): Given an asynchronous message passing service (`asend`, `arecv`), implement a synchronous `mulsend` that sends a message `times` times to a destination and only

completes when all `times` messages have been received.

Analysis: "Synchronous" implies the sender must wait for confirmation. "mulsend" implies it must wait for `times` confirmations. The protocol is simple:

1. `mulsend` process sends the message.
2. The destination process, upon receiving, sends back an acknowledgement (ACK) message.
3. `mulsend` waits to receive `times` ACKs before it can complete.

Solution (`mulsend` part): (The exam text implies `mulsend` is the complex part, `mulrecv` is likely just `arecv`)

```
// We assume an asynchronous service with:
// asend(destination, message)
// arecv(source) // source can be ANY

// The message format needs to be defined. Let's use a struct.
// For the data message: <type="DATA", payload=msg>
// For the ack message:  <type="ACK">

void mulsend(pid_t destination, T msg, int times) {
    // Send the message 'times' times
    for (int i = 0; i < times; i++) {
        asend(destination, create_data_message(msg));
    }

    // Wait for 'times' acknowledgements
    for (int i = 0; i < times; i++) {
        // Block here until an ACK is received from the destination
        T_ack ack_msg = arecv(destination);
        // We should check if ack_msg is actually an ACK, but for the exam
this is often sufficient.
    }
    // Now we are done.
}

// The receiver side would look like:
void receiver_process() {
    while(true) {
        T_data data_msg = arecv(ANY); // Receive from anyone
        // ... process the message ...
        pid_t sender = data_msg.get_sender();
        asend(sender, create_ack_message()); // Send ACK back
    }
}
```

Takeaway: To build synchronous behavior, the sender must `asend` and then immediately `arecv` to wait for a reply (an ACK).

Example 2: Protocol for Arbitrary Length Messages (from `scritto-2024-06-25`)

Problem: Given a message passing service where messages have a maximum length of 256 bytes, implement a service for messages of arbitrary length.

Analysis: This requires **fragmentation and reassembly**. The large message must be broken into smaller chunks that fit within the 256-byte limit. The receiver must reassemble these chunks in the correct order.

- **Protocol:** We need a header in each small message. The header could contain:
  - A sequence number.
  - A "more fragments" flag (e.g., 1 if more packets are coming, 0 for the last one).
- **Sender (`asend`):**
  1. Takes the large message.
  2. Loops, creating 256-byte chunks (or smaller for the last one).
  3. For each chunk, it prepends the header and sends it using the underlying `lasend`.
- **Receiver (`arecv`):**
  1. Needs to buffer incoming fragments for each sender.
  2. When `larecv` returns a fragment, it checks the header.
  3. It adds the fragment to a temporary buffer associated with the sender.
  4. If the "more fragments" flag is 0, it means the message is complete. It concatenates all fragments from the buffer, returns the full message, and clears the buffer for that sender.

Solution Sketch:

```
// Underlying service: lasend(dest, bytes), larecv(sender)
// Max payload size for lasend is e.g., 251 bytes to leave space for a
header.

// Map<pid_t, byte_array> reassembly_buffers; // Stores incomplete messages

void asend(pid_t dest, msg_t* msg) {
    byte* data = msg->data;
    int len = msg->length;

    while (len > 251) {
        // Create packet with "more fragments" flag set
        byte_packet packet = create_packet(data, 251, MORE_FRAGMENTS_TRUE);
        lasend(dest, packet);
        data += 251;
        len -= 251;
    }
    // Send the last fragment
    byte_packet last_packet = create_packet(data, len,
MORE_FRAGMENTS_FALSE);
    lasend(dest, last_packet);
}

msg_t* arecv(pid_t sender) {
    while (true) {
        // Receive a low-level packet
        byte_packet packet = larecv(sender);
```

```
        // Append payload to the reassembly buffer for this sender
        reassembly_buffers[sender].append(packet.payload);

        // If this was the last packet, the message is complete
        if (packet.is_last_fragment()) {
            msg_t* full_message =
create_message_from(reassembly_buffers[sender]);
            reassembly_buffers.remove(sender); // Clear buffer
            return full_message;
        }
        // Otherwise, loop to receive the next fragment
    }
}
```

Takeaway: Message passing questions can be about protocol design. Think about what information needs to be exchanged (headers, flags, ACKs) to achieve the desired behavior.

Part 3: The Ennalogo - The Unbreakable Rules of the Exam

Your professor has a very specific set of rules. Violating them is an automatic fail. Here is a summary of the rules from slides 202-206. **MEMORIZE THESE.**

General Rules (All Paradigms)

1. **No `busy-waiting`**. High-level paradigms like semaphores and monitors were invented to *avoid* busy-waiting (`while(condition){}`). Using it is a major error.
2. **No Mixing Paradigms**. If the exercise asks for a monitor, use only monitor primitives. Do not use semaphores inside a monitor or vice-versa.
3. **Initialize Your Variables**. Using a variable before it has been initialized is a fundamental error.
4. **No Fantasy Calls**. Do not invent functions like `block()`, `wakeup()`, or `restart()`. Use only the primitives provided by the paradigm.
5. **Data Structures:** You can assume standard data structures (Queues, Stacks, Lists) exist, but they must **not** contain concurrency primitives themselves. Their parameters must contain all the information needed to implement them (e.g., a queue for a monitor should not implicitly handle locking).

Rules for Semaphores

1. **Primitives:** The only operations are `S.P()` and `S.V()` (object-oriented) or `P(S)` and `V(S)` (procedural). They take no other parameters and have no return value (in the standard model).
2. **Initialization is Mandatory.** A semaphore *must* be initialized to a non-negative value.
3. **Correct Usage:** A program that only ever calls `P` or only ever calls `V` on a semaphore is fundamentally wrong. They must be used in pairs (though not necessarily in the same function).
4. **Fairness:** Unless stated otherwise, assume semaphores are "fair" (FIFO). Processes are unblocked in the order they were blocked.
5. **Mutual Exclusion:** When using semaphores to protect a critical section on shared data, *all* access to that data must be within the `P()`/`V()` block.

Rules for Monitors

1. **Primitives:** The only synchronization operations are `c.wait()` and `c.signal()` on a `condition` variable. They have no parameters and no return value.
2. **No Deadlock Inside Monitor:** Calling a blocking operation (like `busy-wait`, or a `P()` on a semaphore if you were mixing paradigms) inside a monitor procedure leads to deadlock, as no other process can enter to signal you. This is a critical error.
3. **Condition Variables are Private:** The wait queues and urgent stack are internal to the monitor implementation. You cannot access them directly.
4. **No `wait`/`signal` in Constructor.** The initialization block of a monitor cannot contain `wait` or `signal` calls.
5. **Policy is Signal-Urgent:** The Hoare-style policy is the default.
6. **Correct `wait`/`signal` Usage:** A condition variable that is only ever waited on or only ever signaled is useless and indicates a logical error in your program.
7. **`wait` is Not First:** A monitor procedure that has `wait()` as its very first instruction is almost always wrong. There should be a condition (`if` or `while`) checked before waiting.

Rules for Message Passing

1. **Use the Right Paradigm.** The exam will specify which type to use (synchronous, asynchronous, completely asynchronous). If it just says "asynchronous," it means `asend` is non-blocking and `arecv` is blocking.
2. **No Shared Memory.** This is the core principle. Processes cannot share global variables. All data exchange must be via messages.
3. **Correct `send`/`receive` Usage:** A process that only sends and never receives, or vice-versa, is likely wrong (unless it's a "fire-and-forget" scenario, which is rare in these problems).
4. **Server Processes:** Only use a dedicated server/dispatcher process if the problem explicitly asks for it or if it's impossible to implement the service otherwise (e.g., to serialize requests or manage a shared resource in a no-shared-memory environment).
5. **FIFO:** Unless stated otherwise, assume message delivery is FIFO between any given sender-receiver pair.

Final Study Advice:

1. **Master the Concepts:** Before looking at solutions, be sure you understand the difference between `V(S)` and `c.signal()`. Understand why monitors have implicit mutexes.
2. **Identify the Pattern:** When you read a problem, ask yourself: "What is the core synchronization pattern here?" Is it simple signaling? Is it managing a resource pool? Is it a barrier? Is it strict alternation?
3. **Choose Your State Variables:** Decide what minimal state you need to track. (e.g., `last_color`, `num_waiting`, `is_buffer_full`).
4. **Write the Logic:** Write pseudocode, paying close attention to the `if/while` condition before any `wait()` call. Think about who needs to be signaled and when.
5. **Review Against the Ennalogo:** This is the most important step. Go through your solution and check it against every rule. This will catch most common errors.