Midterm Paper 2 Answers (WhatsAppImage2025-03-11at00.20.46_8f754

This document provides detailed answers to the questions from the second midterm paper image provided.

Question 1: Describe Flynn's taxonomy processing models for multiprocessor/multicomputer systems. (10 Marks)

Concept: Flynn's Taxonomy for Parallel Computer Architectures.

Syllabus Relevance: Module 1 (Introduction, Relation to parallel systems).

Answer:

Flynn's taxonomy, proposed by Michael J. Flynn in 1966, is a classification system for parallel computer architectures based on the number of concurrent instruction streams and data streams available in the architecture. It categorizes systems into four types:

1. **SISD (Single Instruction stream, Single Data stream):**
    - **Description:** This represents the traditional von Neumann architecture. There is one instruction stream being executed by a single processing unit (CPU), operating on a single data stream fetched from memory. Instructions are executed sequentially.
    - **Characteristics:** No parallelism in instruction or data streams. This is the model for conventional uniprocessor computers.
    - **Example:** Older personal computers, mainframes executing sequential programs.
    - **Diagram:** `+-----------------+ +-----------------+ +-----------------+ | Instruction Pool| ---> | Processing Unit | ---> | Data Pool | +-----------------+ +-----------------+ +-----------------+ (Single IS) (Single PU) (Single DS)`
2. **SIMD (Single Instruction stream, Multiple Data streams):**
    - **Description:** A single instruction is executed simultaneously by multiple processing units, each operating on a different data stream. A control unit

    broadcasts the instruction, and each processing unit applies it to its local data.
    - **Characteristics:** Achieves data-level parallelism. Suitable for problems involving operations on large arrays or vectors, like image processing or scientific computations.
    - **Example:** Vector supercomputers (e.g., Cray-1), array processors, modern GPUs (Graphics Processing Units) which have many cores executing the same instruction on different pixels or data elements.
    - **Diagram:** `+----------------+ +-----+-----+-----+ +-----+-----+-----+ | Instruction Pool| ---> | PU1 | PU2 | ... | ---> | DS1 | DS2 | ... | +----------------+ +-----+-----+-----+ +-----+-----+-----+ (Single IS) (Multiple PUs) (Multiple DS)`

3. **MISD (Multiple Instruction streams, Single Data stream):**

    - **Description:** Multiple instructions operate simultaneously on the same data stream. Different processing units execute different instruction sequences on the same data.
    - **Characteristics:** This model is rarely implemented in practice. Some argue that pipelined architectures or fault-tolerant systems (where multiple processors execute the same instructions on the same data for redundancy) might fit, but it's generally considered impractical or theoretical.
    - **Example:** No common commercial examples. Sometimes cited in the context of systolic arrays for specific tasks or fault tolerance.
    - **Diagram:** `+-----+-----+-----+ +-----+-----+-----+ +----------------+ | IS1 | IS2 | ... | ---> | PU1 | PU2 | ... | ---> | Data Pool | +-----+-----+-----+ +-----+-----+-----+ +----------------+ (Multiple IS) (Multiple PUs) (Single DS)`

4. **MIMD (Multiple Instruction streams, Multiple Data streams):**

    - **Description:** Multiple processing units execute different instruction streams independently, each operating on its own data stream. This is the most general and flexible model for parallel computing.
    - **Characteristics:** Achieves both task-level and data-level parallelism. Represents most modern multiprocessor and multicomputer systems.
    - **Subcategories:**
        - **Shared Memory MIMD:** Processors share a common address space (e.g., multi-core processors, Symmetric Multiprocessing systems - SMPs). Communication occurs implicitly through shared variables.
        - **Distributed Memory MIMD:** Each processor has its own private memory. Processors communicate explicitly by passing messages over a network (e.g., clusters, Massively Parallel Processors - MPPs). Distributed Systems fall under this category.

- **Example:** Multi-core CPUs, clusters of workstations, supercomputers like IBM Blue Gene, distributed systems.
- **Diagram:** `+-----+-----+-----+ +-----+-----+-----+ +-----+-----+-----+ | IS1 | IS2 | ... | ---> | PU1 | PU2 | ... | ---> | DS1 | DS2 | ... | +-----+-----+-----+ +-----+-----+-----+ +-----+-----+-----+ (Multiple IS) (Multiple PUs) (Multiple DS)`

Flynn's taxonomy provides a high-level classification but doesn't capture all nuances of modern parallel architectures (like heterogeneity or memory consistency models), yet it remains a fundamental concept for understanding parallel processing.

Question 2: In the above process execution figure, Is the cut, shown by curve X, a consistent cut? Why? (10 Marks)

Concept: Consistent Global State (Consistent Cut), Happened-Before Relation.

Syllabus Relevance: Module 1 (Global state, Cuts), Module 2 (Snapshot algorithms).

Answer:

A cut in a distributed execution space-time diagram represents a global state. A cut is **consistent** if it does not contain the receive event of any message without also containing the corresponding send event. Visually, this means no message arrow should cross the cut line starting from after the cut (the future) and ending before the cut (the past).

Let's examine the cut denoted by curve X in the provided diagram: * The cut passes through P1 after event e1^3. * The cut passes through P2 after event e2^3. * The cut passes through P3 after event e3^2.

The global state represented by cut X includes events {e1^1, e1^2, e1^3} on P1, {e2^1, e2^2, e2^3} on P2, and {e3^1, e3^2} on P3, along with the states immediately following these events.

Now, let's check the messages relative to this cut:

1. **Message from P1 to P2 (sent e1^1, received e2^1):** Both send and receive events are before the cut X. OK.
2. **Message from P1 to P3 (sent e1^2, received e3^1):** Both send and receive events are before the cut X. OK.
3. **Message from P2 to P1 (sent e2^2, received e1^3):** Both send and receive events are before or at the cut X. OK.

4. **Message from P2 to P3 (sent e2^3, received e3^3):** The send event (e2^3) is included in the cut (it's the last event on P2 before the cut). The receive event (e3^3) occurs after the cut on P3 (which cuts after e3^2). This message crosses the cut from past to future. This is allowed and does not violate consistency.
5. **Message from P3 to P1 (sent e3^2, received e1^4):** The send event (e3^2) is included in the cut (it's the last event on P3 before the cut). The receive event (e1^4) occurs after the cut on P1 (which cuts after e1^3). This message crosses the cut from past to future. This is allowed and does not violate consistency.
6. **Message from P3 to P2 (sent e3^1, received e2^2):** Both send and receive events are before the cut X. OK.

Crucially, we need to check if any message crosses from the future to the past. Let's re-examine: * Receive event e1^3 (on P1, included) corresponds to send event e2^2 (on P2, included). OK. * Receive event e2^1 (on P2, included) corresponds to send event e1^1 (on P1, included). OK. * Receive event e2^2 (on P2, included) corresponds to send event e3^1 (on P3, included). OK. * Receive event e3^1 (on P3, included) corresponds to send event e1^2 (on P1, included). OK.

All receive events included in the cut have their corresponding send events also included in the cut. No message crosses the cut line X from the future to the past.

Justification: The cut X is consistent because it adheres to the definition of a consistent cut. For every message `m` whose receive event `Receive(m)` is included in the global state represented by the cut, the corresponding send event `Send(m)` is also included in that global state. We have verified that no message violates this condition.

Therefore, the cut shown by curve X is a consistent cut.

Question 3: From the give asynchronous execution diagram, find crown and write the reason for it. (10 Marks)

Concept: Concurrency in Asynchronous Executions, Happened-Before Relation. (Assuming "crown" refers to identifying concurrent events, as it's not a standard widely used term in this specific context. It might be instructor-specific terminology or a typo for "concurrent set" or similar.)

Syllabus Relevance: Module 1 (Asynchronous executions, Global state), Module 2 (Message ordering).

Answer:

In an asynchronous execution, two events `a` and `b` are considered **concurrent** (denoted `a || b`) if neither event happened-before the other. That is, `a || b` if and only if `¬(a -> b)` and `¬(b -> a)`, where `->` represents the Lamport happened-before relation.

The happened-before relation (`->`) is defined as the smallest relation satisfying: 1. If `a` and `b` are events in the same process and `a` comes before `b`, then `a -> b`. 2. If `a` is the sending of a message `m` and `b` is the reception of `m`, then `a -> b`. 3. If `a -> c` and `c -> b`, then `a -> b` (transitivity).

Let's identify the events in the diagram: * P1: s1 (send m1), s3 (send m3) * P2: s2 (send m2) * P3: r1 (receive m1), r2 (receive m2), r3 (receive m3)

Let's establish the happened-before relationships: * Within processes: s1 -> s3; r1 -> r2 -> r3 * Message communication: s1 -> r1; s2 -> r2; s3 -> r3 * Transitivity examples: s1 -> r1 -> r2; s1 -> r1 -> r3; s1 -> s3 -> r3; s2 -> r2 -> r3

Now, let's find pairs of events that are concurrent (assuming "crown" asks for concurrent events):

- **s1 and s2:** Are they concurrent? s1 does not happen before s2 (no path). s2 does not happen before s1 (no path). **Yes, s1 || s2.**
- **s1 and r2:** s1 -> r1 -> r2. So, s1 happens before r2. Not concurrent.
- **s1 and r3:** s1 -> r1 -> r3 (or s1 -> s3 -> r3). So, s1 happens before r3. Not concurrent.
- **s2 and s1:** (Same as s1 and s2). **Yes, s2 || s1.**
- **s2 and s3:** Are they concurrent? s2 does not happen before s3 (no path). s3 does not happen before s2 (no path). **Yes, s2 || s3.**
- **s2 and r1:** Are they concurrent? s2 does not happen before r1 (no path). r1 does not happen before s2 (no path). **Yes, s2 || r1.**
- **s2 and r3:** s2 -> r2 -> r3. So, s2 happens before r3. Not concurrent.
- **s3 and s1:** s1 -> s3. Not concurrent.
- **s3 and s2:** (Same as s2 and s3). **Yes, s3 || s2.**
- **s3 and r1:** Are they concurrent? s3 does not happen before r1 (no path). r1 does not happen before s3 (no path, as s1->s3 but no path from r1 to s1). **Yes, s3 || r1.**
- **s3 and r2:** Are they concurrent? s3 does not happen before r2 (no path). r2 does not happen before s3 (no path, as s2->r2 but no path from s3 to s2). **Yes, s3 || r2.**
- **r1 and s2:** (Same as s2 and r1). **Yes, r1 || s2.**
- **r1 and s3:** (Same as s3 and r1). **Yes, r1 || s3.**
- **r2 and s1:** s1 -> r1 -> r2. Not concurrent.
- **r2 and s3:** (Same as s3 and r2). **Yes, r2 || s3.**

Identifying a "Crown" (Set of Mutually Concurrent Events): If "crown" refers to a maximal set of mutually concurrent events (an antichain in the poset of events ordered by happened-before), then we look for the largest set where no event in the set happened-before any other event in the set.

Let's list the concurrent pairs again: (s1, s2), (s2, s3), (s2, r1), (s3, r1), (s3, r2), (r1, s2), (r1, s3), (r2, s3)

Consider the set {s2, r1}. We know s2 || r1. This is a set of concurrent events. Consider the set {s2, s3, r1}. We know s2 || s3, s2 || r1, s3 || r1. All pairs are concurrent. This is a larger set of mutually concurrent events. Consider the set {s2, s3, r1, r2}. Is r1 || r2? No, r1 -> r2. So this set is not mutually concurrent. Consider the set {s1, s2}. They are concurrent. Consider the set {s1, s2, s3}. Is s1 || s3? No, s1 -> s3. Not mutually concurrent. Consider the set {s1, s2, r1}. Is s1 || r1? No, s1 -> r1. Not mutually concurrent.

It appears the largest set of mutually concurrent events (maximal antichain) in this diagram is **{s2, s3, r1}**.

Reason: Events s2, s3, and r1 are mutually concurrent because: * s2 || s3 (neither s2 -> s3 nor s3 -> s2) * s2 || r1 (neither s2 -> r1 nor r1 -> s2) * s3 || r1 (neither s3 -> r1 nor r1 -> s3) Furthermore, no other event can be added to this set while maintaining mutual concurrency. For example, adding s1 violates concurrency with s3 (s1->s3) and r1 (s1->r1). Adding r2 violates concurrency with s2 (s2->r2) and r1 (r1->r2). Adding r3 violates concurrency with s2 (s2->r3), s3 (s3->r3), and r1 (r1->r3).

(Disclaimer: If "crown" has a very specific, different definition provided by the instructor, this interpretation based on concurrency might need adjustment. However, identifying concurrent events is a standard analysis of such diagrams.)

Question 4: Brief Bagrodia Algorithm for binary rendezvous with necessary example diagrams. (10 Marks)

Concept: Bagrodia Algorithm, Rendezvous Synchronization.

Syllabus Relevance: Module 1 (Primitives for distributed communication, Synchronous versus asynchronous executions). Note: This specific algorithm is less common in standard textbooks and likely comes from specific lecture material or the provided PDFs. I will search the extracted text.

(Self-correction: Searching extracted text... The provided PDFs (unit1, unit1a, module2, uni3a, unit3b, module4a, module4b, module5a, module5b) and the extracted book pages do not seem to contain a specific description of the "Bagrodia Algorithm for binary rendezvous". This algorithm might be from a specific lecture slide (like the mentioned "Physical clock synchronization -read from ppt") or another source not provided. Without the specific source material defining this algorithm, providing an accurate answer as expected by the instructor is difficult. However, I can describe the general concept of binary rendezvous and common approaches, acknowledging the specific algorithm name might have details I cannot access.)

Answer:

(Note: The specific "Bagrodia Algorithm" for binary rendezvous was not found in the provided PDF materials or standard textbook sections outlined. The following describes the general concept of binary rendezvous and a plausible mechanism, which may or may not align perfectly with the specific Bagrodia algorithm taught.)

Binary Rendezvous: Rendezvous is a synchronization mechanism in message passing where both the sender and receiver must be ready to communicate before the message transfer occurs. It acts as a synchronization point. In binary rendezvous, only two processes are involved: one sender and one receiver.

The core idea is that the first process to arrive at the communication point (either the sender ready to send or the receiver ready to receive) must wait for the other process to arrive before the communication (message transfer) can proceed. This ensures tight synchronization between the pair.

General Mechanism (Illustrative, not necessarily Bagrodia's): A common way to implement binary rendezvous involves handshake messages:

1. **Sender Initiates:**
    - Process P1 (Sender) wants to send message `M` to Process P2 (Receiver) via rendezvous.
    - P1 sends a `READY_TO_SEND(M)` message to P2.
    - P1 blocks, waiting for an acknowledgment from P2.
2. **Receiver Arrives:**
    - When P2 is ready to receive, it checks if a `READY_TO_SEND` message has arrived from P1.
    - If yes: P2 accepts the message `M`, performs the receive operation, and sends an `ACK_RECEIVE` back to P1.
    - If no: P2 blocks, waiting for the `READY_TO_SEND` message.
3. **Completion:**
    - When P1 receives the `ACK_RECEIVE` from P2, it knows the rendezvous is complete and the message has been transferred. P1 unblocks and continues execution.

- ◦ P2, having received the message, also continues execution.

(Alternative: Receiver Initiates) A similar handshake can occur if the receiver reaches the receive point first, sending a `READY_TO_RECEIVE` message and waiting for the sender.

Diagram (Conceptual Handshake):

Case 1: Sender arrives first

```
 Time ->
P1 (Sender): ---READY_TO_SEND(M)--> | (Blocks)
                              |           |
                              |           |<--ACK_RECEIVE---
                              |           |              |
P2 (Receiver): --------------(Ready to Receive)-------------> (Receives M, Unblocks P1)
```

Case 2: Receiver arrives first

```
 Time ->
P1 (Sender): --------------(Ready to Send)-------------> (Sends M, Unblocks P2)
                              |           |              |
                              |<--READY_TO_RECEIVE---
                              |           |
P2 (Receiver): ---READY_TO_RECEIVE--> | (Blocks)
```

Key Aspects: * **Synchronization:** Ensures both sender and receiver are ready before transfer. * **Blocking:** The first process to arrive typically blocks. * **Atomicity:** The message transfer and synchronization appear as a single atomic event to the participating processes.

Without the specific details of the Bagrodia algorithm, this general explanation covers the core principles of binary rendezvous.

Question 5: How distributed mutual exclusion can be implemented? Detail with system model. (10 Marks)

Concept: Distributed Mutual Exclusion (DME) Implementation Approaches, System Model.

Syllabus Relevance: Module 3 (Distributed mutual exclusion algorithms, System model).

Answer:

(This question is identical to Question 5 from Midterm Paper 1. Please refer to the detailed answer provided for Midterm Paper 1, Question 5, which covers the system model and the three basic approaches: Non-Token-Based (Permission-Based), Token-Based, and Quorum-Based, with examples like Lamport, Ricart-Agrawala, Suzuki-Kasami, and Maekawa, along with conceptual diagrams.)

[Content from Midterm Paper 1, Question 5 Answer would be inserted here]

Summary of the three approaches: 1. **Non-Token-Based:** Request permission from others (e.g., all others), use timestamps for ordering (Lamport, Ricart-Agrawala). 2. **Token-Based:** Circulate a unique token; only the holder enters CS (Suzuki-Kasami, Raymond's Tree). 3. **Quorum-Based:** Request permission from a subset (quorum) with intersection property (Maekawa).

Each approach requires specific assumptions about the system model (process behavior, network reliability, message ordering, synchrony) and offers different performance and fault-tolerance characteristics.