Pthreads:

1.) Define thread ref. variables.
- Ex) pthread_t thread0;

2.) Create entry point function.
- function must have void* return type
- function must have a single void* arg.
- Ex) void* entry_func(void* param);

3.) Create thread.
- Create using pthread_create(thread, attr, start_routine, arg)
  - thread: pointer to pthread_t variable
  - attr: Extra attributes (can just set to NULL)
  - start_routine: pointer to entry point function
  - arg: pointer passed to entry point function as parameter.
- Ex) pthread_create(&thread0, NULL, entry_func, &parameter);

☆ Might not be need 4.) Join threads back together
- join using pthread_join(thread, Status)
  - thread: pthread_t variable used when thread created. (not a pointer this ti
  - status: pointer to return value pointer (can just set to NULL)
- Ex) pthread_join(thread0, NULL);

Mutual Exclusion in Pthreads:
- Declare mutex variable: pthread_mutex_t mutex;
- Initialize mutex variable: pthread_mutex_init(mutex, attr); (attr can be set to NULL for default)
- Lock + Release when threads modify critical sections:
  - Ex)
```
pthread_mutex_lock(mutex);
critical_value += 1;
pthread_mutex_unlock(mutex);
```

MPI: Message Passing Interface:

- Communicator & Group objects:
  - define which collection of processes can communicate with each other.
  - Use MPI_COMM_WORLD when communicator required,
    - Predefined communicator w/ all all MPI processes included.
- Rank ("task ID"):
  - Unique integer identifier assigned when process initializes.
    - Contiguous
    - Begins at 0
  - Used to specify source/destination of messages
- MPI Execution Environment Setup:
  - MPI_Init(&argc, &argv):
    - Initialize environment
    - Must be called in every MPI program
    - Must be called before any other MPI functions
    - Must only be called once.
  - MPI_Comm_rank(comm, &rank):
    - Returns rank of a process in communicator
    - @param rank = location to store returned rank number
  - MPI_Comm_size(comm, &size):
    - Returns the amount of processes that were requested for job
    - @param size = location to store returned number of processes
  - MPI_Finalize():
    - Terminates MPI execution environment
    - Should be last MPI function called
- Point-to-Point Communication:
  - Blocking Send: • MPI_Send(buffer, count, type, dest, tag, comm)
  - Blocking Receive: • MPI_Recv(buffer, count, type, source, tag, comm, status)

MPI: Message Passing Interface (2):

- Point-to-Point Communication Cont'd:
  - Message Passing Routine (MPI_Send/MPI_Recv) Parameters:
    - Buffer: variable that is sent/received, usually passed by reference (&var)
    - Count: number of data elements of "type" to be sent
    - Type: MPI_CHAR, MPI_INT, MPI_LONG, MPI_FLOAT, etc (MPI_"type")
    - [MPI_Send] - Dest: Specifies the rank (Task ID) of the receiving process
    - [MPI_Recv] - Source: Specifies rank of sending process
    - Tag: Any arbitrary non-negative number to identify message
    - Comm: Communicator, usually MPI_COMM_WORLD
    - [MPI_Recv] - status: Indicates source of message and tag of message.

Consistency:

↳ Data-Centric Consistency Models:

↳ Sequential Consistency:

- When processes run concurrently, any valid interleaving of read/write operations is acceptable behavior, BUT all processes must see same interleaving of operations.
- Processes "see" writes from all other processes
- Process only "sees" it's own reads
- Statements must be executed in program order

↳ Causal Consistency:

- "Weakened" form of sequential consistency.
  - Makes distinction between events that are potentially causally related and those that are not
- If event b is caused or influenced by event a, causality requires that everyone else first see a, then see b.
- Vector timestamps can be used to construct/maintain dependency graph

↳ Data-Centric Consistency w/ Grouping:

- Use shared synchronization variables to grant access to Critical Sections (CS)
- ENTER_CS → Acquire relevant synch. var.
- LEAVE_CS → Release synch. var.
- Possible for multiple processes to own some synch var but only in nonexclusive mode; can read, but not write, the associated data.
- Required Criteria:
  - At an acquire, all remote changes to guarded data must be made visible. (ie. released)
  - Before updating shared data, process must enter CS in exclusive mode; no other process may hold synch. var. in any (exclusive/nonexclusive) mode.
  - If process wants to enter CS in nonexclusive mode, it must fetch most recent copy of shared data from current owner of synch. var.

Client Centric Consistency Models:

↳ Eventual Consistency: * Is not a client-centric model

- In absence of frequent updates, all replicas of data set converge towards identical copies.
- Only require that update eventually propagates to all replicas.
- Problem arises if client accesses different data replicas after updating another replica.
  - Alleviated through Client-Centric Consistency Models.

- Provides guarantees for a Single client concerning consistency of accesses to a data store. No guarantees for concurrent accesses by different clients.

↳ Monotonic Read Consistency:

- Guarantees that if a process has seen a value X at time t, it will never see an older version of x at later time.
- Interested in operations carried out by a single Processor P.

↳ Monotonic Write Consistency:

- A write operation by a process on data item x is completed before any successive write operations on x by the same process

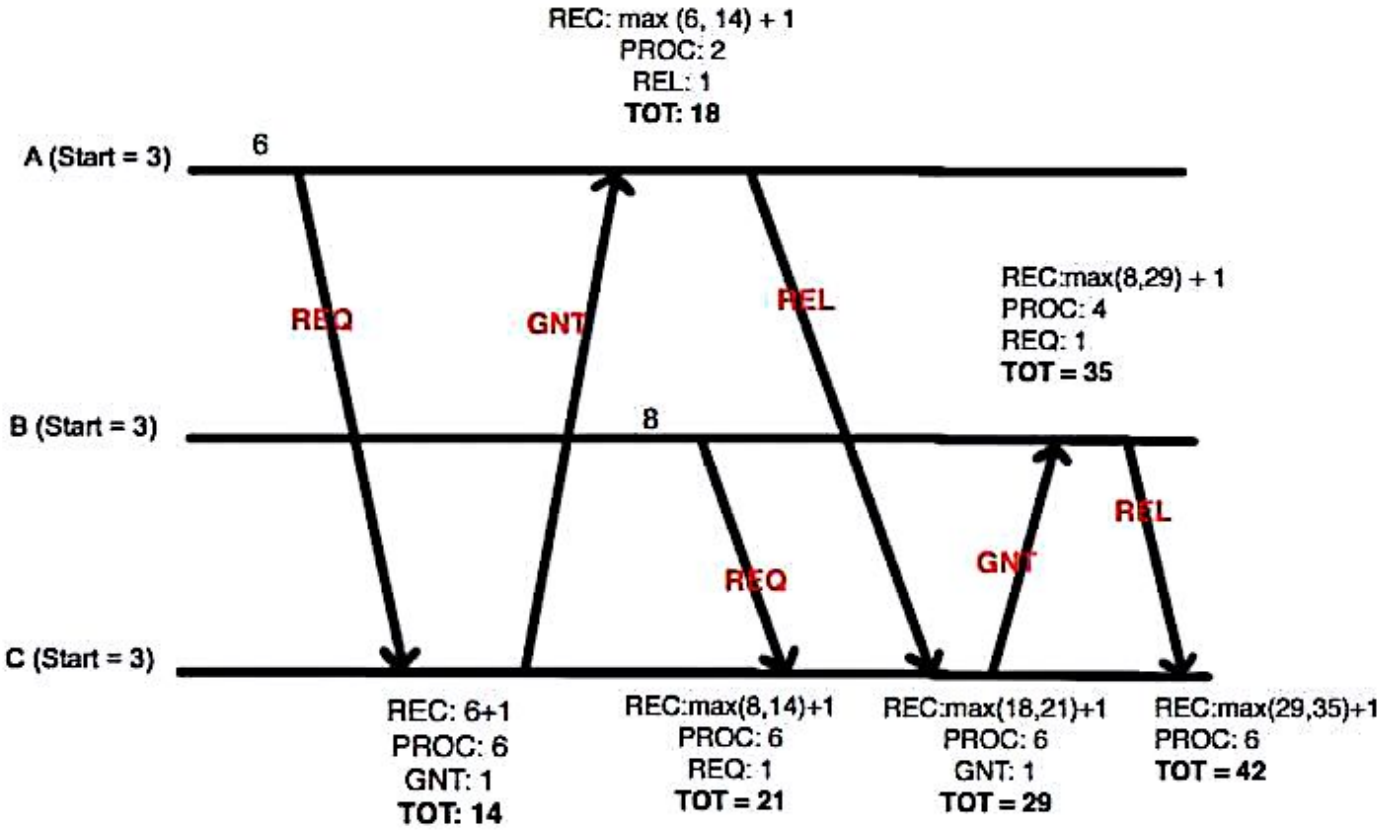
↳ Read-Your-Writes Consistency:

- Effect of write operation by a process on x will always be seen by a successive read operation on x by same process

↳ Write-Follow-Read Consistency:

- Write op. by a process on X following a previous read op on X by the same process is guaranteed to take place on same/more recent value of x that was read.

4. **Processing Times: A = 2, B = 4, C = 6**
Every message takes + 1 (REQ, REL, GNT)



It is assumed that Process A&B perform calculations before a request. Each REQ/REL messages increments their clock by 1. PROC time is shown above for each process. Each GNT message sent from Process C has to increment the clock by 1. TOT is all the time taken from REQ to GNT.

Im not sure whether you need to process the last one. Unprocessed = 36, Processed = 42