Message Passing

Shared memory vs Message Passing

- Shared memory systems:
  - there is a (common) shared address space throughout the system.
  - Communication takes place via shared data variables and control variables
  - Semaphores and monitors

- All multicomputer systems that do not have a shared address space.

- Hence, message Passing is used for communication.
- Shared memory systems are easy to develop and use

- Communication via message-passing can be simulated by communication via shared memory and vice-versa

Emulating message passing on a shared memory system

- The shared address space can be partitioned into disjoint parts, one part being assigned to each processor.
- “Send” and “receive” operations can be implemented by writing to and reading from the destination/sender processor’s address space, respectively.
- A separate location can be reserved as the mailbox for each ordered pair of processes.
- A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox.
- Mailboxes can be assumed to have unbounded size.
- Synchronization primitives inform the receiver/sender after the data has been sent/received.

Emulating message passing on a shared memory system

- Each shared location can be modeled as a separate process;
  - “write” is emulated by sending update message to owner process;
  - “read” is emulated by sending query message to owner process
- Expensive operation
- Read and write operations are implemented by using network-wide communication
- Application can of course use a combination of shared memory and message-passing.
- Combination of shared memory system and message passing system can be used.
  - In multicomputer system, each “processor” may be a tightly coupled multiprocessor system with shared memory.
  - Between two computers, the communication is by message passing.

Primitives for distributed communication

- Broadly defined into two;
  - Blocking/non-blocking
  - Synchronous/asynchronous

- Message send and message receive communication primitives are denoted as:
  - **Send()** : at least two parameters – the destination, and the buffer in the user space, containing the data to be sent.
  - **Receive():** at least two parameters – the source from which the data is to be received and the user buffer into which the data is to be received

Primitives for distributed communication

- There are two ways of sending data when the **Send primitive** is invoked
  - *The buffered option :* Copies data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network.
  - *The unbuffered option:* the data gets copied directly from the user buffer onto the network.
- For the **Receive primitive**, the buffered option is required
  - The data may already have arrived when the primitive is invoked
  - Needs a storage place in the kernel.

Primitives for distributed communication

- Synchronous primitives:
  - Both the Send() and Receive() handshake with each other.
  - The processing for the Send primitive completes only after the invoking processor learns that the corresponding Receive primitive has also been invoked and that the receive operation has been completed.
  - The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.

- Asynchronous primitives :
  - A Send primitive is asynchronous if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.
  - It does not make sense to define asynchronous Receive primitives

Primitives for distributed communication

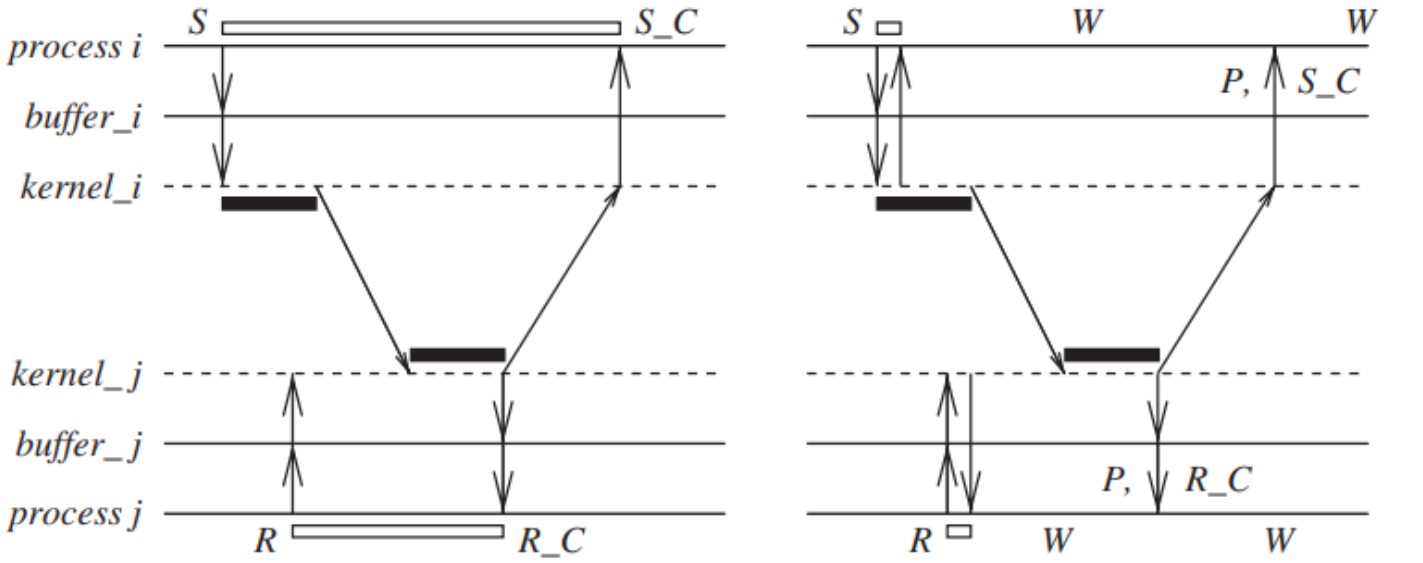
- Blocking primitives:
  - A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

- Non-blocking primitives:
  - A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
  - For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer.
  - For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender
  - For non-blocking primitives, a return parameter on the primitive call returns a system-generated handle which can be later used to check the status of completion of the call.
  - Wait call.

Primitives for distributed communication

- Blocking synchronous send
- Non-blocking synchronous send
- Blocking asynchronous send
- Non-blocking asynchronous send
- Blocking receive
- Non-blocking Receive

Primitives for distributed communication



Primitives for distributed communication

- A synchronous Send is easier to use.
- Handshake between the Send and the Receive makes the communication appear instantaneous, thereby simplifying the program logic.
- Synchronous Send lowers the efficiency within the sender process
- The non-blocking asynchronous Send is useful for transferring large data.
- Programmer has to keep track of the completion of primitive operations

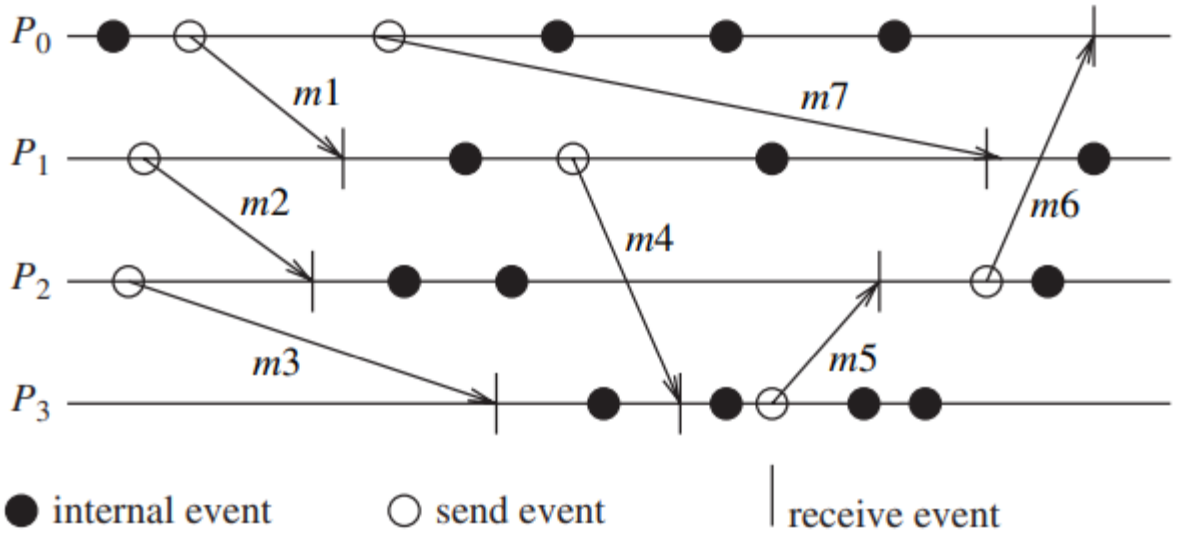
Processor Synchrony

- Classification of synchronous versus asynchronous processors.
- Indicates that all the processors execute in lock-step with their clocks synchronized.
- As this synchrony is not attainable in a distributed system, a large granularity of code is termed as a step.
- For steps the processors are synchronized.
- Ensures that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code.

Synchronous vs asynchronous executions

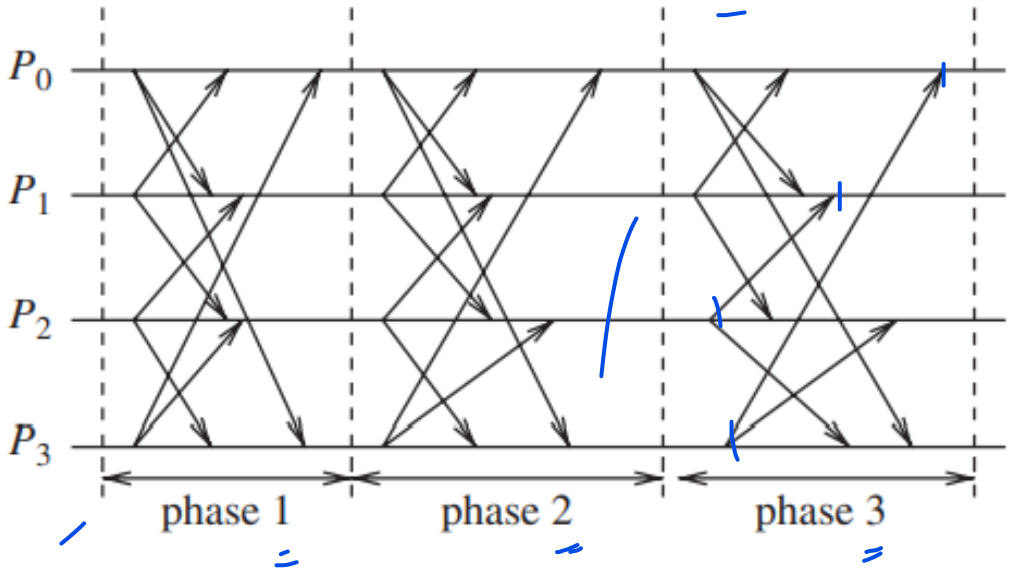
- An asynchronous execution is an execution in which :

  (i) there is no processor synchrony and there is no bound on the drift rate of processor clocks,
  (ii) message delays (transmission + propagation times) are finite but unbounded
  (iii) there is no upper bound on the time taken by a process to execute a step.



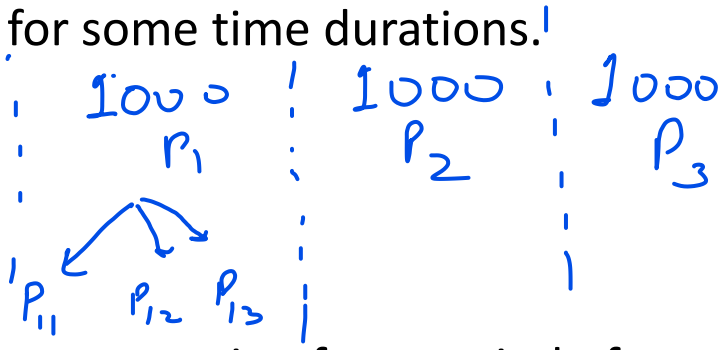
Synchronous vs asynchronous executions

- A synchronous execution is an execution in which :
  - (i) processors are synchronized and the clock drift rate between any two processors is bound
  - (ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round
  - (iii) there is a known upper bound on the time taken by a process to execute a step.



Synchronous vs asynchronous executions

- Easier to design and verify algorithms with synchronous executions.
- Practically difficult to build a completely synchronous system.
- Will involve delaying or blocking some processes for some time durations.



- **Virtually synchronous executions:**
  - Processors are allowed to have an asynchronous execution for a period of time and then they synchronize.
  - Within each round/phase/step, there may be a finite and bounded number of sequential sub-rounds (or subphases or sub-steps) that processor executes.
  - Each sub-round is assumed to send at most one message per process; hence the message(s) sent will reach in a single message hop.

Emulations

- An asynchronous program can be emulated on a synchronous system
  - Synchronous system is a special case of an asynchronous system
  - All communication finishes within the same round in which it is initiated

- A synchronous program can be emulated on an asynchronous system using a tool called synchronizer.