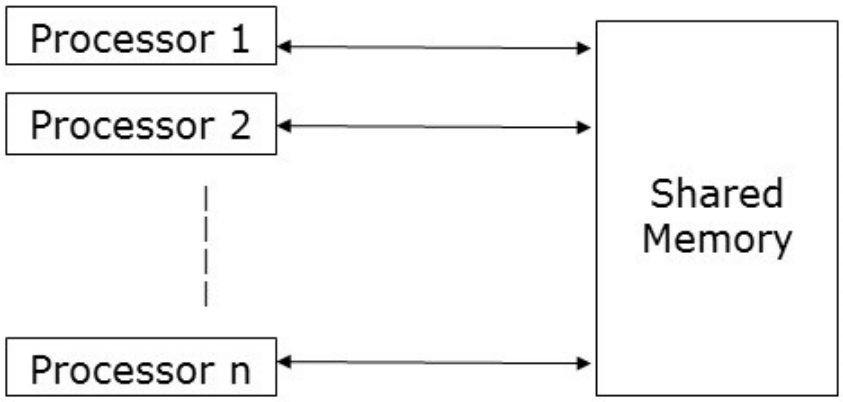
Shared Memory Model

Shared memory emphasizes on **control parallelism** than on **data parallelism**. In the shared memory model, multiple processes execute on different processors independently, but they share a common memory space. Due to any processor activity, if there is any change in any memory location, it is visible to the rest of the processors.

As multiple processors access the same memory location, it may happen that at any particular point of time, more than one processor is accessing the same memory location. Suppose one is reading that location and the other is writing on that location. It may create confusion. To avoid this, some control mechanism, like **lock / semaphore,** is implemented to ensure mutual exclusion.



Merits of Shared Memory Programming

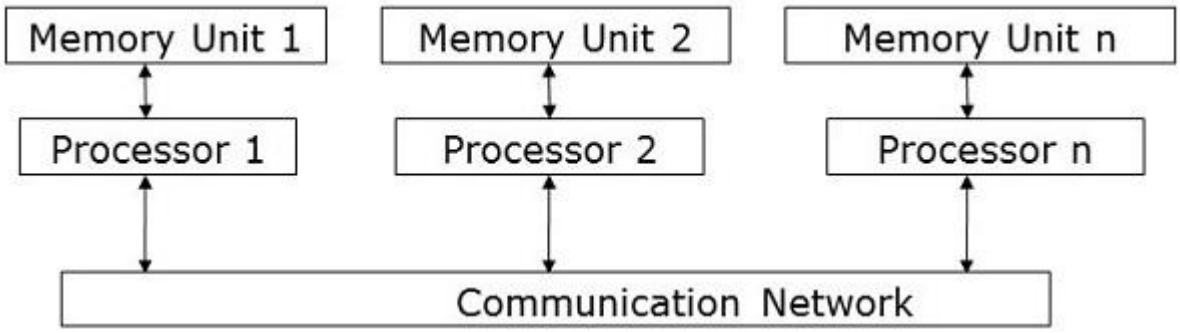
- Global address space gives a user-friendly programming approach to memory.
- Due to the closeness of memory to CPU, data sharing among processes is fast and uniform.
- There is no need to specify distinctly the communication of data among processes.
- Process-communication overhead is negligible.
- It is very easy to learn.

Demerits of Shared Memory Programming

- It is not portable.
- Managing data locality is very difficult.

Message Passing Model

Message passing is the most commonly used parallel programming approach in distributed memory systems. Here, the programmer has to determine the parallelism. In this model, all the processors have their own local memory unit and they exchange data through a communication network.



Processors use message-passing libraries for communication among themselves. Along with the data being sent, the message contains the following components −

- The address of the processor from which the message is being sent;
- Starting address of the memory location of the data in the sending processor;
- Data type of the sending data;
- Data size of the sending data;
- The address of the processor to which the message is being sent;
- Starting address of the memory location for the data in the receiving processor.

Processors can communicate with each other by any of the following methods −

- Point-to-Point Communication
- Collective Communication
- Message Passing Interface

Point-to-Point Communication

Point-to-point communication is the simplest form of message passing. Here, a message can be sent from the sending processor to a receiving processor by any of the following transfer modes −

- **Synchronous mode** − The next message is sent only after the receiving a confirmation that its previous message has been delivered, to maintain the sequence of the message.
- **Asynchronous mode** − To send the next message, receipt of the confirmation of the delivery of the previous message is not required.

Collective Communication

Collective communication involves more than two processors for message passing. Following modes allow collective communications −

- **Barrier** − Barrier mode is possible if all the processors included in the communications run a particular bock (known as **barrier block**) for message passing.
- **Broadcast** − Broadcasting is of two types −
    - **One-to-all** − Here, one processor with a single operation sends same message to all other processors.
    - **All-to-all** − Here, all processors send message to all other processors.

Messages broadcasted may be of three types −

- **Personalized** − Unique messages are sent to all other destination processors.

- **Non-personalized** − All the destination processors receive the same message.
- **Reduction** − In reduction broadcasting, one processor of the group collects all the messages from all other processors in the group and combine them to a single message which all other processors in the group can access.

Merits of Message Passing

- Provides low-level control of parallelism;
- It is portable;
- Less error prone;
- Less overhead in parallel synchronization and data distribution.

Demerits of Message Passing

- As compared to parallel shared-memory code, message-passing code generally needs more software overhead.

Message Passing Libraries

There are many message-passing libraries. Here, we will discuss two of the most-used message-passing libraries −

- Message Passing Interface (MPI)
- Parallel Virtual Machine (PVM)

Message Passing Interface (MPI)

It is a universal standard to provide communication among all the concurrent processes in a distributed memory system. Most of the commonly used parallel computing platforms provide at least one implementation of message passing interface. It has been implemented as the collection of predefined functions called **library** and can be called from languages such as C, C++, Fortran, etc. MPIs are both fast and portable as compared to the other message passing libraries.

Merits of Message Passing Interface

- Runs only on shared memory architectures or distributed memory architectures;
- Each processors has its own local variables;
- As compared to large shared memory computers, distributed memory computers are less expensive.

Demerits of Message Passing Interface

- More programming changes are required for parallel algorithm;
- Sometimes difficult to debug; and
- Does not perform well in the communication network between the nodes.

Parallel Virtual Machine (PVM)

PVM is a portable message passing system, designed to connect separate heterogeneous host machines to form a single virtual machine. It is a single manageable parallel computing resource. Large computational problems like superconductivity studies, molecular dynamics simulations, and matrix algorithms can be solved more cost effectively by using the memory and the aggregate power of many computers. It manages all message routing, data conversion, task scheduling in the network of incompatible computer architectures.

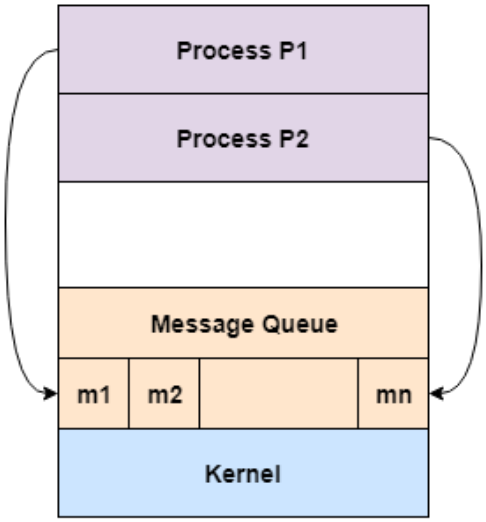
Features of PVM

- Very easy to install and configure;
- Multiple users can use PVM at the same time;
- One user can execute multiple applications;
- It’s a small package;
- Supports C, C++, Fortran;
- For a given run of a PVM program, users can select the group of machines;
- It is a message-passing model,
- Process-based computation;
- Supports heterogeneous architecture.

Message Passing Process Communication Model

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message passing model of process communication is given as follows −



Message Passing Model

In the above diagram, both the processes P1 and P2 can access the message queue and store and retrieve data.

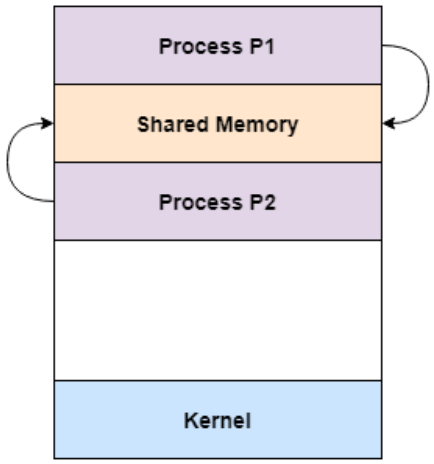
An advantage of message passing model is that it is easier to build parallel hardware. This is because message passing model is quite tolerant of higher communication latencies. It is also much easier to implement than the shared memory model.

However, the message passing model has slower communication than the shared memory model because the connection setup takes time.

Shared Memory Process Communication Model

The shared memory in the shared memory model is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

A diagram that illustrates the shared memory model of process communication is given as follows:



Shared Memory Model

In the above diagram, the shared memory can be accessed by Process 1 and Process 2.

An advantage of shared memory model is that memory communication is faster as compared to the message passing model on the same machine.

However, shared memory model may create problems such as synchronization and memory protection that need to be addressed.

Introduction

- In this chapter, we outline the basic concepts in message-passing computing.
  - We introduce the basic structure of message-passing programs and how to specify message passing between processes.
  - We discuss these in general, and then we outline two specific systems, PVM and MPI
  - Finally, we discuss how to evaluate message-passing parallel programs, both theoretically and in practice.

1. Basics of Message Passing Programming

1.1 Programming Options

- Programming a message-passing multicomputer can be achieved by
  - **1.** Designing a special parallel programming language
  - **2.** Extending the syntax/reserved words of an existing sequential high-level language to handle message passing
  - **3.** Using an existing sequential high-level language and providing a library of external procedures for message passing

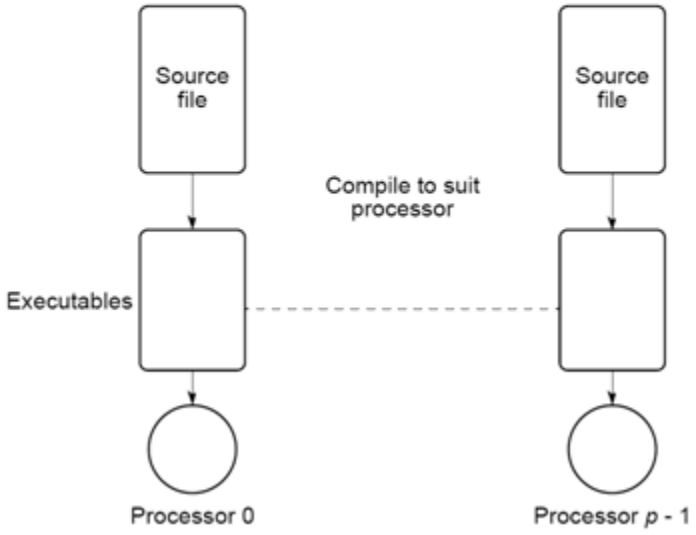
1. Basics of Message Passing Programming

- We will concentrate upon the third option. Necessary to say explicitly what processes are to be executed, when to pass messages between concurrent processes, and what to pass in the messages.

- Two primary methods are needed in this form of a message-passing system:
  - **1.** A method of creating separate processes for execution on different computers
  - **2.** A method of sending and receiving messages

1.2 Process Creation

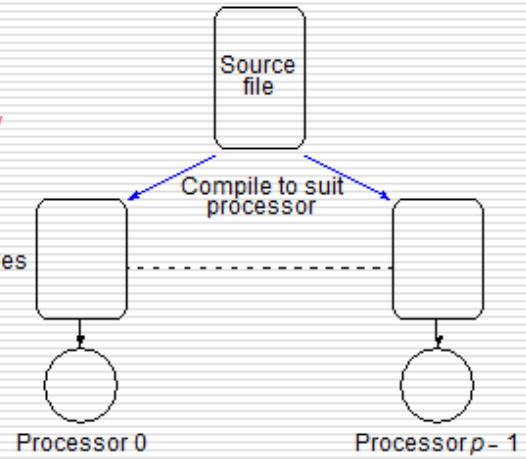
- **MPMD model**



1.2 Process Creation

- **SPMD model**
  - Different processes are merged into one program. Within the program are control statements that will customize the code; i.e. select different parts for each process

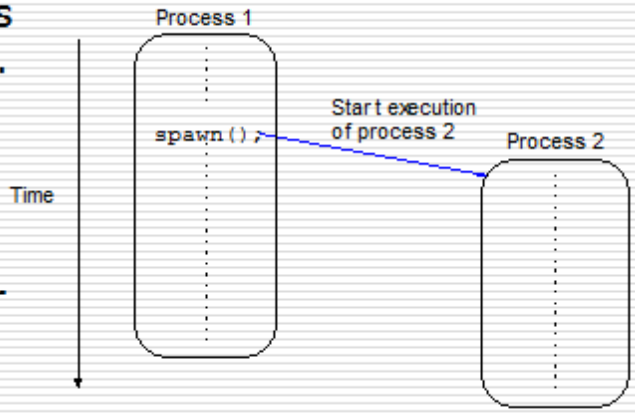
Basic MPI way



1.2 Process Creation

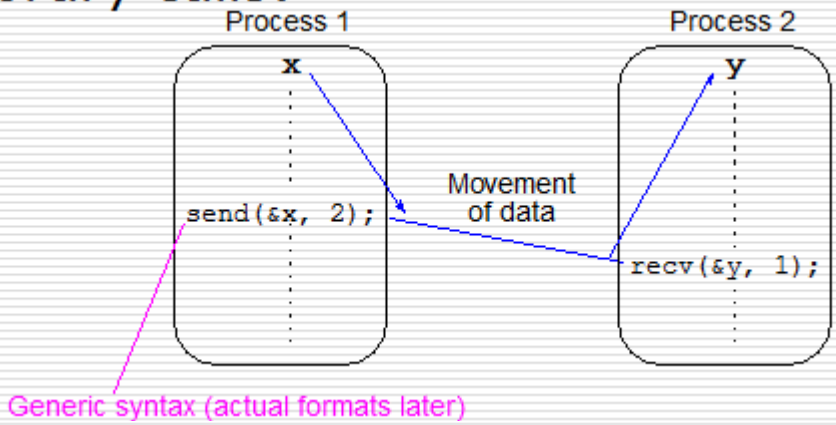
- **MPMD model**

Separate programs for each processor. One processor executes master process. Other processes started from within master process - *dynamic process* creation.



1.3 Message-Passing Routines

- Passing a message between processes using send() and recv() library calls:



1.3 Message-Passing Routines

- **Synchronous Message Passing**
  - Routines that actually return when the message transfer has been completed.

Synchronous send routine

  - Waits until complete message can be accepted by the receiving process before sending the message.

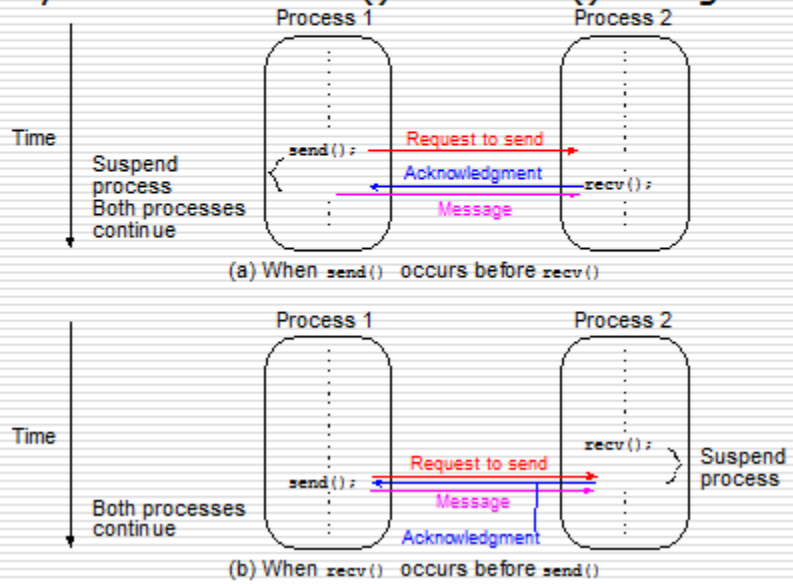
Synchronous receive routine

  - Waits until the message it is expecting arrives.

- Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

1.3 Message-Passing Routines

- Synchronous send() and recv() using 3-way protocol



1.3 Message-Passing Routines

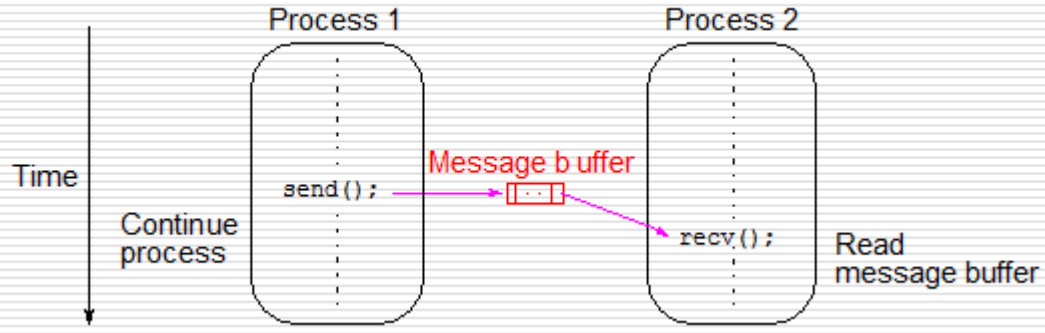
- Asynchronous Message Passing
  - Routines that do not wait for actions to complete before returning. Usually require local storage for messages.
  - More than one version depending upon the actual semantics for returning.
  - In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

1.3 Message-Passing Routines

- **Blocking and Nonblocking Message Passing**
  - **Blocking** - has been used to describe routines that do not return until the transfer is completed.
    - The routines are "blocked" from continuing.
    - In that sense, the terms *synchronous* and *blocking* were synonymous.
  - **Non-blocking** - has been used to describe routines that return whether or not the message had been received.
  - The terms *blocking* and *nonblocking* redefined in systems such as MPI:

1.3 Message-Passing Routines

- **How message-passing routines can return before the message transfer has been completed**
  - Generally, a *message buffer* needed between source and destination to hold message:

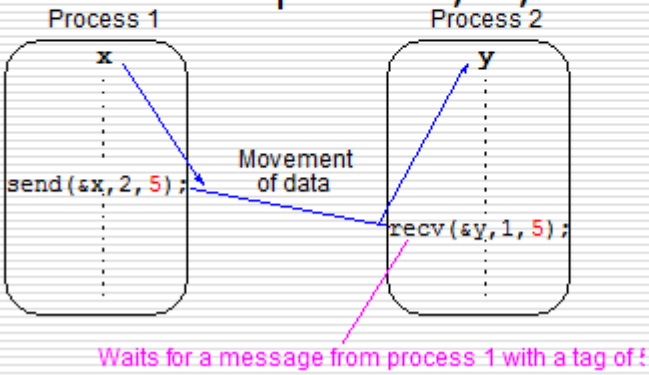


1.3 Message-Passing Routines

- Once local actions completed and message is safely on its way, sending process can continue with subsequent work.
- Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes re-available - i.e then routine behaves as a synchronous routine.

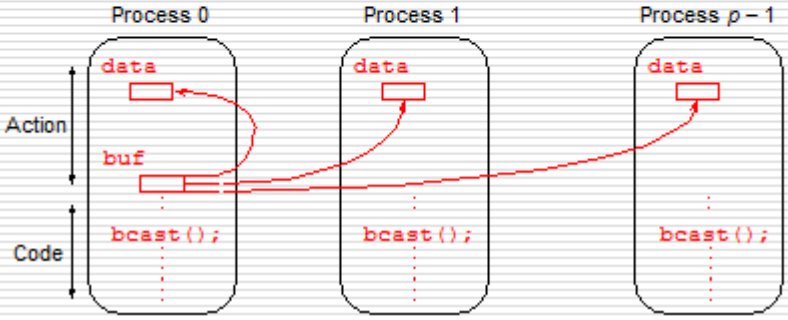
- Message Selection
  - **Message Tag**
    - Used to differentiate between different types of messages being sent.
    - Message tag is carried within message.
    - If special type matching is not required, a wild card message tag is used, so that the recv() will match with any send().

- Example:
  - To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y:

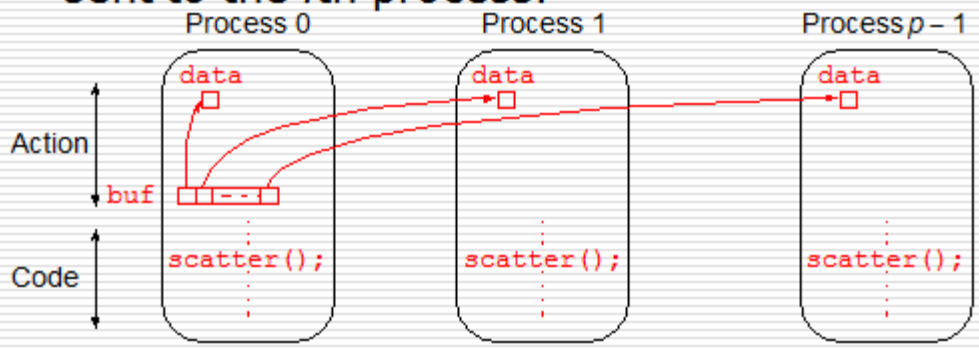


- "Group" message passing routines
  - Have routines that send message(s) to a group of processes or receive message(s) from a group of processes
  - Higher efficiency than separate point-to-point routines although not absolutely necessary.

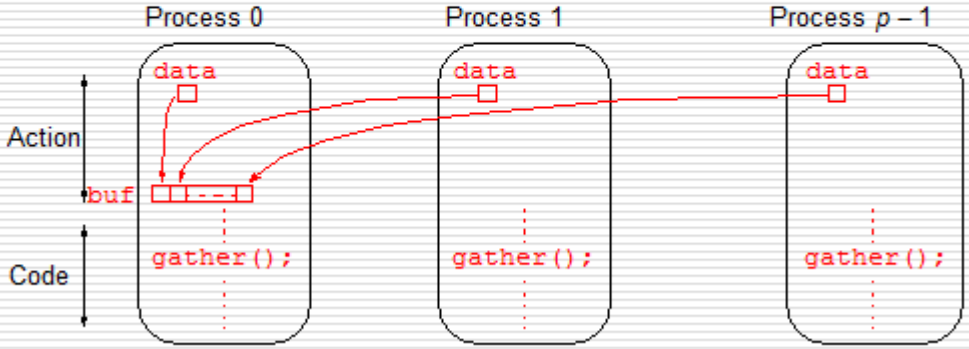
- Broadcast
  - Sending the same message to all the processes concerned with the problem.
  - M*ulticast* - sending the same message to a defined group of processes.



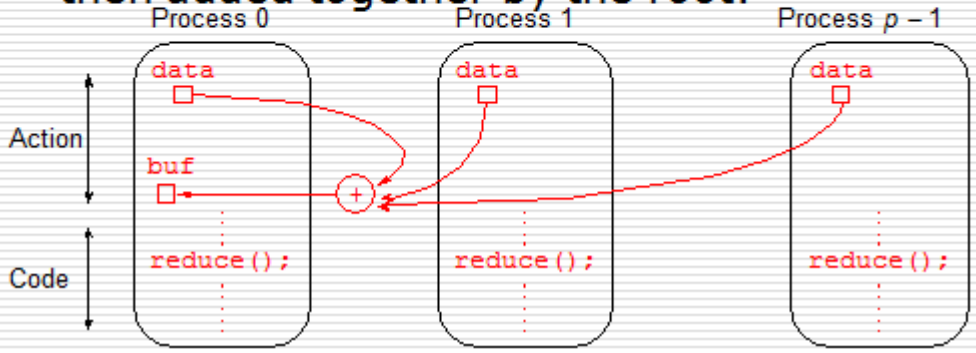
- Scatter
  - Sending each element of an array of data in the root to a separate process. The contents of the *i*th location of the array is sent to the *i*th process.



- Gather
  - Having one process collect individual values from a set of processes.



- **Reduce**
  - Gather operation combined with a specified arithmetic or logical operation.
  - Example, the values could be gathered and then added together by the root:



Leader-Election algorithm

What is Election?

- In a group of processes, elect a *Leader* to undertake special tasks.
- What happens when a leader fails (crashes)
  - Some (at least one) process detects this (how?)
  - Then what?
- Focus of this lecture: Election algorithm
  1. Elect one leader only among the non-faulty processes
  2. All non-faulty processes agree on who is the leader

System Model/Assumptions

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes can call an election simultaneously.
  - All of them together must yield a single leader only
- The result of an election should not depend on which process calls for it.
- Messages are eventually delivered.

Problem Specification

- At the end of the election protocol, the non-faulty process with the best (highest) election *attribute value* is elected.
  - Attribute examples: leader has highest id or address. Fastest cpu. Most disk space. Most number of files, etc.
- Protocol may be initiated anytime or after leader failure
- A *run* (execution) of the election algorithm must always guarantee at the end:
  - **Safety:** ∀ non-faulty p: (p's `elected` = (q: a particular non-faulty process with the best attribute value) or ⊥)
  - **Liveness:** ∀ election: (election terminates)
    & ∀ p: non-faulty process, p's `elected` is not ⊥

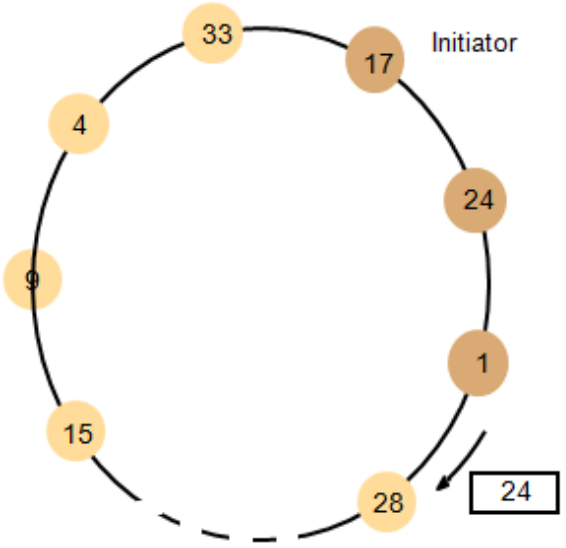
Algorithm 1: Ring Election

- N Processes are organized in a logical ring
  - p_i has a communication channel to $p_{(i+1) \bmod N}$
  - All messages are sent clockwise around the ring.
- Any process p_i that discovers the old coordinator has failed initiates an "election" message that contains p_i's own id:attr. This is the *initiator* of the election.
- When a process p_i receives an *election* message, it compares the attr in the message with its own attr.
  - If the arrived attr is greater, p_i forwards the message.
  - If the arrived attr is smaller and p_i has not yet forwarded an election message, it overwrites the message with its own id:attr, and forwards it.
  - If the arrived id:attr matches that of p_i, then p_i's attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an "elected" message to its neighbor with its id, announcing the election result.
- When a process p_i receives an elected message, it
  - sets its variable $elected_i \leftarrow$ id of the message.
  - forwards the message, unless it is the new coordinator.

Ring-Based Election: Example

In this example, attr:=id)

- In the example: The election was started by process 17.
  The highest process identifier encountered so far is 24.
  (final leader will be 33)
- The worst-case scenario occurs when the counter-clockwise neighbor (@ the initiator) has the highest attr.

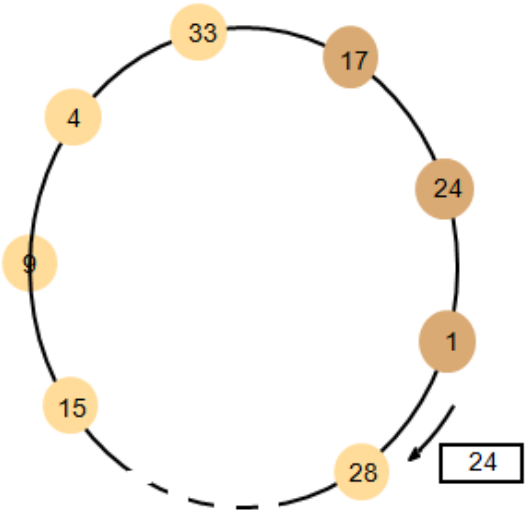


Ring-Based Election: Analysis

- **The worst-case scenario occurs when the counter-clockwise neighbor has the highest attr.**

In a ring of N processes, in the worst case:

- **A total of N-1 messages are required to reach the new coordinator-to-be (election messages).**
- **Another N messages are required until the new coordinator-to-be ensures it is the new coordinator (election messages – no changes).**
- **Another N messages are required to circulate the elected messages.**
- **Total Message Complexity = 3N-1**
- **Turnaround time = 3N-1**

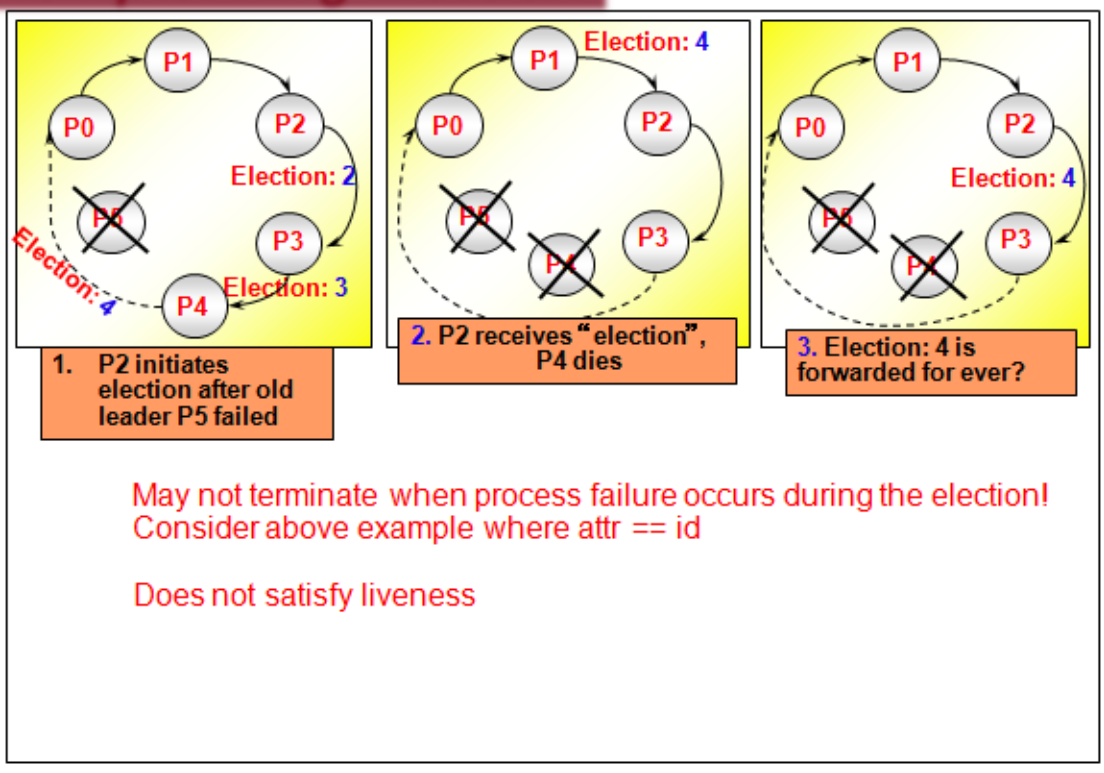


Assume – no failures happen during the run of the election algorithm

- Safety and Liveness are satisfied.

What happens if there are failures during the election run?

Example: Ring Election



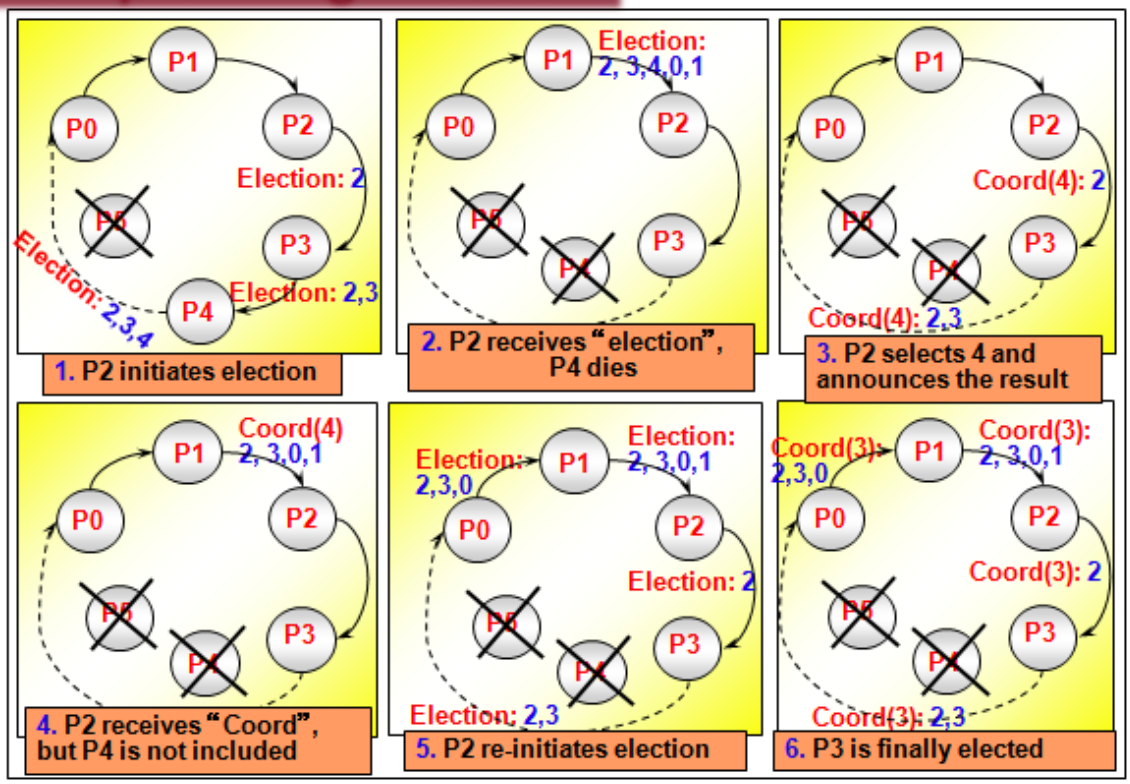
May not terminate when process failure occurs during the election!
Consider above example where attr == id

Does not satisfy liveness

Algorithm 2: Modified Ring Election

- Processes are organized in a logical ring.
- Any process that discovers the coordinator (leader) has failed initiates an "election" message.
- The message is circulated around the ring, bypassing failed processes.
- Each process appends (adds) its id:attr to the message as it passes it to the next process (without overwriting what is already in the message)
- Once the message gets back to the initiator, it elects the process with the best election attribute value.
- It then sends a "coordinator" message with the id of the newly-elected coordinator. Again, each process adds its id to the end of the message, and records the coordinator id locally.
- Once "coordinator" message gets back to initiator,
  - election is over if would-be-coordinator's id is in id-list.
  - else the algorithm is repeated (handles election failure).

Example: Ring Election

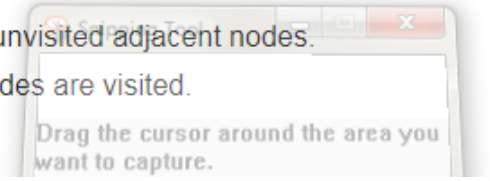


Breadth-First Search

Breadth-First Search (or BFS) is an algorithm for searching a tree or an undirected graph data structure. Here, we start with a node and then visit all the adjacent nodes in the same level and then move to the adjacent successor node in the next level. This is also known as **level-by-level search**.

Steps of Breadth-First Search

- Start with the root node, mark it visited.
- As the root node has no node in the same level, go to the next level.
- Visit all adjacent nodes and mark them visited.
- Go to the next level and visit all the unvisited adjacent nodes.
- Continue this process until all the nodes are visited.



Pseudocode

Pseudocode

Let **v** be the vertex where the search starts in Graph **G**.

```
BFS(G,v)

   Queue Q := {};

   for each vertex u, set visited[u] := false;
   insert Q, v;
   while (Q is not empty) do
      u := delete Q;

      if (not visited[u]) then
         visited[u] := true;
         for each unvisited neighbor w of u
            insert Q, w;
      end if

   end while

END BFS()
```

Data Parallel Model

In data parallel model, tasks are assigned to processes and each task performs similar types of operations on different data. **Data parallelism** is a consequence of single operations that is being applied on multiple data items.

Data-parallel model can be applied on shared-address spaces and message-passing paradigms. In data-parallel model, interaction overheads can be reduced by selecting a locality preserving decomposition, by using optimized collective interaction routines, or by overlapping computation and interaction.

The primary characteristic of data-parallel model problems is that the intensity of data parallelism increases with the size of the problem, which in turn makes it possible to use more processes to solve larger problems.

Example − Dense matrix multiplication.