University of Khartoum
Faculty of Engineering and Architecture
Department of Electrical and Electronic Engineering

Process Synchronization in Computer Networks


Prepared By:
Hiba Siddig Maki

Supervisor:
Dr. El Rashid Osman

June, 2003
Dedication

This work is dedicated to all who are interested in the field of synchronization in computer network.

It is a simple attempt to pour a drop in the main stream of knowledge.
ACKNOWLEDGMENT

First of all we thank Allah for his blessing & support

Then we would like to acknowledge the generous assistance & guidance of Dr. Alrasheed in all stages of the present work

It’s also a pleasure to acknowledge with gratitude the encouragement & love of my caring families

Next I find myself indebted to all my doctors: Dr. Sami, Dr. Iman, Dr. Mohammed Ali, Dr. Abubaker & Dr. Mergani, for their unlimited offer & help
Abstract

Synchronization is used in real life so as to organize the work to grantee its continuation and preventing what we call blocking, which means no one go on working. Synchronization has a technical meaning in computer field that is related to this meaning but somewhat different.

However, synchronization is increasingly used and being an important issue with the development of operating systems which improve the possibility for processes to cooperate with each other even in distributed systems. Processes can operate within one machine using shared memory or through multiple machines using message passing.

This thesis searches how to achieve process synchronization either on a single machine or multicomputer systems. In the former one, synchronization can be achieved using multiple methods such as semaphore and monitor, where as in the later one we can use centralized, distributed or token ring algorithms. Then we focus on one of synchronization problems, deadlock. Deadlock is a situation where two or more processes are all blocked and none of them can become unblocked until one of others become unblocked. Three methods for handling deadlock situation: prevention avoidance, and detection.

A C++ program has been designed using Message-Passing Interface (MPI) under LINUX operating system to execute a producer – consumer synchronization problem.
تُؤدِّي إلى استخدام النظام، حيث تُستخدم التزامن في حياتنا، ومن خلاله، التخصيص في الجداول الزمنية.

(Blocking) يُعرف بـ "Blocking". يُطلق عليه "Blocking". يُحتمل أن يكون تزامن عمليات تكون في التوزيع (distribution) للتعامل مع أهمية تخصيص الذاكرة. 

هذا يُشمل البرمجة بلغة C++، وتخصيص النظام في لينكس. (deadlock)

(producer-consumer problem)
# Table of Contents

<table>
<thead>
<tr>
<th>SUBJECT</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgment</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>ii</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td>v</td>
</tr>
<tr>
<td>Chapter 1 - Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 General</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2 - Operating System Overview</td>
<td></td>
</tr>
<tr>
<td>2.1 What Is an OS</td>
<td>4</td>
</tr>
<tr>
<td>2.2 OS Services</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Types of OS</td>
<td>5</td>
</tr>
<tr>
<td>2.4 OS Concept</td>
<td>6</td>
</tr>
<tr>
<td>2.4.1 Processes</td>
<td>6</td>
</tr>
<tr>
<td>2.4.2 Files</td>
<td>7</td>
</tr>
<tr>
<td>2.4.3 System calls</td>
<td>8</td>
</tr>
<tr>
<td>2.4.4 Shells</td>
<td>8</td>
</tr>
<tr>
<td>Chapter 3 - Process management</td>
<td></td>
</tr>
<tr>
<td>3.1 Concept</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Process States</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Process Control Block</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Cooperation Processes</td>
<td>12</td>
</tr>
<tr>
<td>3.5 Interprocess Communication</td>
<td>14</td>
</tr>
<tr>
<td>3.6 Type of IPC</td>
<td>17</td>
</tr>
<tr>
<td>Chapter 4 - Process synchronization techniques</td>
<td></td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Race Condition</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Critical Section</td>
<td>20</td>
</tr>
<tr>
<td>4.3.1 Hardware solution</td>
<td>20</td>
</tr>
<tr>
<td>4.3.2 Software solution</td>
<td>21</td>
</tr>
<tr>
<td>4.3.3 Mixed solution</td>
<td>21</td>
</tr>
<tr>
<td>4.4 Mutual Exclusion</td>
<td>22</td>
</tr>
<tr>
<td>4.4.1 Busy Waiting</td>
<td>22</td>
</tr>
<tr>
<td>4.4.2 Sleep &amp; Wake Up</td>
<td>24</td>
</tr>
<tr>
<td>4.4.3 Semaphore</td>
<td>26</td>
</tr>
<tr>
<td>4.4.4 Monitor</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Message Passing</td>
<td>31</td>
</tr>
<tr>
<td>4.5 Deadlock</td>
<td>33</td>
</tr>
<tr>
<td>4.5.1 Deadlock prevention</td>
<td>35</td>
</tr>
<tr>
<td>4.5.2 Deadlock detection and recovery</td>
<td>36</td>
</tr>
</tbody>
</table>
4.5.3 Ostrich method ........................................37

<table>
<thead>
<tr>
<th>Chapter 5-Process synchronization in distributed system</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction ........................................38</td>
</tr>
<tr>
<td>5.3 Communication ........................................38</td>
</tr>
<tr>
<td>5.4 Synchronization in distributed system ...............40</td>
</tr>
<tr>
<td>5.4.1 Introduction ........................................40</td>
</tr>
<tr>
<td>5.4.2 Event ordering ......................................40</td>
</tr>
<tr>
<td>5.4.2.1 Happened-before relation ........................41</td>
</tr>
<tr>
<td>5.4.2.2 Implementation ...................................41</td>
</tr>
<tr>
<td>5.4.3 Mutual Exclusion ....................................42</td>
</tr>
<tr>
<td>5.4.3.1 A Centralized Algorithm .........................42</td>
</tr>
<tr>
<td>5.4.3.2 Distributed Algorithm ............................44</td>
</tr>
<tr>
<td>5.4.3.3 Tokin Ring Algorithm .............................45</td>
</tr>
<tr>
<td>5.4.4 Election Algorithm ..................................46</td>
</tr>
<tr>
<td>5.4.1 The Bully Algorithm ................................47</td>
</tr>
<tr>
<td>5.4.2 A Ring Algorithm ...................................48</td>
</tr>
<tr>
<td>5.4.5 Distributed deadlock .................................49</td>
</tr>
<tr>
<td>5.4.5.1 Deadlock prevention ..............................50</td>
</tr>
<tr>
<td>5.4.5.2 Deadlock avoidance ...............................51</td>
</tr>
<tr>
<td>5.4.5.3 Deadlock detection ...............................51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6-Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction ............53</td>
</tr>
<tr>
<td>6.2 General MPI program ......53</td>
</tr>
<tr>
<td>6.3 Basic MPI function ......53</td>
</tr>
<tr>
<td>6.4 MPI implementation ......54</td>
</tr>
<tr>
<td>6.4.1 Compiling MPI program .57</td>
</tr>
<tr>
<td>6.4.2 Executing MPI program .57</td>
</tr>
<tr>
<td>6.4.3 Terminating LAM ......57</td>
</tr>
<tr>
<td>6.5 About our program ......58</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7 MPI program result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Result ....................60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8 Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion ..........64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A ........................................66</td>
</tr>
<tr>
<td>Appendix B ........................................71</td>
</tr>
<tr>
<td>Appendix C ........................................75</td>
</tr>
</tbody>
</table>

References ........................................78
Chapter 1

Introduction


1.1 General

Computer software helps the computer to engage in many valuable activities to earn its keep. It can be divided into two kinds:

- System programs: manage the operation of the computer itself.
- Application programs: solve the problems for their users.

The most fundamental of all the system programs is the operating system, which controls all the computer’s resources and provides the base upon which the application programs can be written. It run on kernel mode or user mode, not like the other portion of the system programs (editor, compiler, command interpreter) which run on user mode.

This thesis is organized in seven major chapters. This one serves as introduction to the entire thesis. A brief synopsis at the remaining chapters follows.

Introduction:

Because the computer system hardware without system program installed entire it; is a useless to the users, an operating system is an important and essential part at every computer system.

Chapter two explains what is an operating system and what are the services it can provide to the programs and to the users. In this thesis, we don’t concentrate on any particular operating system, but chapter two demonstrates the development and most common concepts of operating systems as general.

Process Management:
The key concept in all operating systems is a process, which is a program in execution. Notice that a program itself is not a process, the former one is a passive entity, for example, content of file on hard-disc, where as process is an active entity which can change its state and has a set of information stored into process control block.

Chapter three demonstrates all those above subjects and show how these processes can deal and communicate with each other.

**Process Synchronization:**

The process concurrency is the heart of modern operating systems. Due to processes cooperation which they may be fed out from the same section in the program, many problem can be occur. So, some sorts of synchronization schemes are needed.

Chapter four points out this problems and how it can be solved either by a software solution or hardware one.

**Process Synchronization in Computer System:**

Distributed system is a collection of processors that don’t share memory or a clock. Although it has many advantages and characteristics than a uniprocessor system, we need more complicated mechanism to perform network communication and to service all local requests as with uniprocessor.

Communication in distributed systems is carried out by passing message through the network. Also, process can communicate with each other, as one user can log in remotely on other computer. So, we need some sort of synchronization to avoid any problem can be occur due to process cooperation, for example, deadlock.

The goal at chapter five is to introduce the reader to the distributed system and how synchronization can be achieved. Also depict one of the synchronization problems, that is deadlock.
**MPI implementation**

Chapter six will demonstrate the implementation of the message passing interface (MPI) in C language in the first section. The appendix B includes the important procedures that describe the producer-consumer problem. Producer is used to produce a random items where as consumer consumes these items.

**MPI program results**

Chapter seven shows the results of C program that written to demonstrate how the producer-consumer problem is solved without deadlock is being happened and examine the total time needed by these processes.
Chapter 2

Operating System Overview
2.1 What is An Operating System?

Operating System (OS) is the most fundamental concept of all the system programs, which interface between user applications and computer hardware and creates an environment that allows application programs to run.

Operating System maintains some rules, that restrict running of each individual program and allocates resources (CPU time, memory space, file storage space, I/O devices, synchronization resources etc.) Also, it controls input/output (I/O) operations and provides safety in user-computer interaction.

Operating system has many functions to do summarized on protection and management. The protection operation need for all the resources within computer system just like memory, I/O, CPU, etc. The management is done for process, main memory, file, secondary storage, networking, I/O, etc.

2.2 Operating System Services

The operating system provides certain services to program and to the users of those programs. These services are provided for the convenience of the programmer, to make the programming task easier. The specific services provided will, of course, differ from one operating system to another, but there are some common classes identified below:

- It loads programs into main memory and execute them.
- It accomplish all I/O operations which can be done from file (eg word file) or other I/O device (printer, scanner, etc.).
- Allows to exchange information between processes, implemented as shared memory or massage passing.
Detect all the error that can be done by hardware, I/O or software and handle the error.

Allocate the resources (CPU cycles, memory, disk, etc..) to processes as needed.

Accounting - keep track of resource usage, which one is used and how long it used.

2.3 Types of Operating Systems

Operating systems and computer architecture have had a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. In this section we will see how they have developed.

**Single-User, Single Task:**

Single task operating system is the one that do one job at a time. These ones allowed only one user to run a single program at a time. An example of single user operating system is Disk Operating System (DOS). A more modern example is the Palm OS for PalmPilot computers.

**Single-User, Multi-Tasking:**

It is the favorite operating systems to the most people that use them on their desktop and laptop computers today. Windows versions are examples of operating systems that allow a single user to have several programs in operation at the same time. For example, a Windows user can be writing a word file, while printing a term paper from a paint program, while listening to some music.

**Multi-User:**

These operating systems allow many users to have their own task simultaneously on illusion that this operating system is dedicated only for them as individual user. The operating system must make sure that the requirements of the various users are
balanced, and that the programs they are using have sufficient and separate resources so that a problem with one user doesn't affect the entire community of users. Linux is one example of a multi-users operating system.

**Real-Time Operating System:**

“Real-time operating systems are used to control machinery, meteorology, scientific instruments, and industrial systems. A real-time operating system typically has very little user interface capability and no end-user utilities. A very important part of a real-time operating system is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs or the data will be lost.” [2]

### 2.4 Operating System Concepts

The interface between the operating system and the user program is defined by the set of “extended instruction” that the operating system provided, which is known as *system call*. This one creates, deletes, and uses various software objects managed by the operating system.

The most important of these are *processes, files, shells, and system calls*.

#### 2.4.1 Process

It is a key concept in all operating systems; “it’s a program in execution”[1]. It consists of the executable program, the program’s data and stack, its program counter, and other register, and all the other information needed to run the program.

All these information except its own address space is stored in a *Process Table*, which is an array or linked list, when the process being suspended due to, for example, had more CPU time
### 2.4.2 Files

Directory is a multiple files that grouped all together; its entry may be file or other directory that gives arise of hierarchies (see fig 2.1).

![Diagram of a hierarchical file system](image)

**Fig 2.1 Example of hierarchical file system**

Process hierarchies are little differ of file one. Process hierarchies are not very deep and typically shorted – lived, also ownership and protection differ from process and file.

It is important to provide a protection means for each person’s files. In Unix, files and directories are protected by assigning to each one 9-bit binary protection code, consist of 3-bit field, one for owner, one for group and one for every one else. E.g. 

```
rwx  r-x  --x.
```
To perform I/O easier, many OS represent each I/O devices as special file. Two kinds of special file exist: Block special files, Character special files. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. Programs that do system maintenance often need this facility. Character special files are used to model devices that consist of character streams rather than fixed size randomly addressable blocks, such as terminates line printers and network interfaces. Pipe is a feature that related to both processes and files. It is a sort of pseudo-file that can be used to connect two processes together.

### 2.4.3 System Calls

It is the way that user program can communicate with the OS and request a services. Corresponding to each system call is a library procedure that user programs can call. This procedure puts the parameters of the system call in a specifies place, such as the machine registers and then issues a TRAP instruction to start the OS. After OS get the control, it examines the parameters to see if they are valid, if so, perform the work requested. When finish, puts the status code in a register and execute a RETURN FROM TRAP instruction, to return control back to the library procedure, which returns to the caller, returning the status code as a function value.

### 2.4.4The Shell

It is not part of the OS, but makes heavy use of many OS features and thus serves as a good example of how the system calls can be used. It is the primary interface between a user and the OS. A figure 2.2 below explains the shell as a level.[1]
Fig 2.2 A view of shell as level
Chapter 3

Process Management
3.1 Concept

The most central concept in any operating systems, and at the heart of it, is a *Process* which is an abstraction of a running program. [2]

A process is the unit of work in a system. Such a system consists of a collection of *concurrently* executed processes, some of which are operating-system processes (those are execute system code), and the rest of which are user processes (those that execute user code).

A process is more than the pogrom code (*text code*). It also includes the current activity, as mentioned in chapter 2. So, a program is a *passive* entity, such as the content of a file stored on disk, where as a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resource.

More than one process can exist at the same time, which are given time slices.

There are two types of processes:

- Normal
- Lightweight (threads)

**Normal Processes**

Processes exist in hierarchies. Parents fork another processes. Children can then load new programs. Processes have a complete environment. They are fully partitioned from each other.

**Threads**

A thread is a software object that can be dispatched and executes in an address space. Several threads can exist in the same address space. Threads vary in implementation. Usually, they are created with some sort of CreateThread function call. They are started as one function from the current process and have their own stack and register set but they share data. They are faster to switch.
3.2 Process States

As a process executes, it changes state as shown in fig 3.1. The current activity of the process is called the State of a Process. The states are:

New  The process is being created.
Running  Instructions are being executed by CPU.
Waiting  Process waits for some event to occur (I/O to be completed or signal to be received).
Ready  The process is waiting for it's turn to use a processor.
Terminated  The process has finished execution.

![Fig 3.1 Process States](image)

3.3 Process Control Block

Each process is represented in the operating system by a process control block (PCB), also called a task control block. It contain many pieces of information associated with a specific process, including: process state, program counter, CPU registers, CPU scheduling information, memory-management information, accounting information, I/O status information.
The PCB simply serves as repository for any information that may vary from process to another (see Fig 3.2). The figure below shows the PCB.

<table>
<thead>
<tr>
<th>Pointer</th>
<th>process State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Process number</td>
</tr>
<tr>
<td></td>
<td>Process counter</td>
</tr>
<tr>
<td></td>
<td>Register</td>
</tr>
<tr>
<td></td>
<td>Memory unit</td>
</tr>
<tr>
<td></td>
<td>List of open files</td>
</tr>
</tbody>
</table>

Fig 3.2 Process control block

3.4 Cooperating Processes

The concurrent processes executing in the operating system may be either independent or cooperating processes. Independent processes do not affect each other during execution, no shared data between them. On the other hand, cooperating processes can be effected or affected by another processes.

There are several reasons for providing an environment that allows process to cooperation including information sharing, computation speed up, modularity, and convenience.

Cooperation between two (or more) processes reside on same computer differ from that they are distributed into two computer in the network. In a single machine all
processes reside on same computer and there are unified memory, one (or more) location within it can implemented as shared ones between all processes and then the cooperation can hold out through these location. Here, if a failure occurs, the entire system fails (see fig 3.3).

Fig 3.3 two processes within one machine

Fig 3.4 Two processes within two machine
In distributed environment, each machine contains its own memory and the processes residing on different sites. Communication can be taken through the network communication links and a failure of one link dose not result of the failure of the system. (See fig. 3.4).

To illustrate the concept of cooperating processes, let us consider the producer/consumer problem: a producer process produces information that is consumed by the consumer process. To allow these processes to run concurrently, a buffer must be available for the item that filled and emptied by the producer and consumer respectively. The buffer may be unbounded-buffer, which places no particular limits on buffer size or bounded-buffer that assume fixed size of the buffer. The buffer either provided by the operating system through the use of interprocess communication (next section), or explicitly coded by the application programmer with the use of shared memory.

3.5 Interprocess Communication (IPC)

IPC is a full set of communications mechanisms for data sharing between applications. It is provided by the operating system to enable the sharing of data on shared memory segments, the exchange of information and data, and the synchronization access to shared resource between processes on the same system. IPC can be distributed across applications or even across the network through the use of these mechanisms. In most cases, one application acts as the server receiving request(s) from client applications for services, or to set up an initial agreement to exchange data between two programs.

There are many reasons to reply the question, why do processes intercommunicate? Often a problem is broken into several stages; each handled by a process that passes information to the next stage. So, multiple processes can be used to complete the same task. Also, sometimes a package is broken up into several parts (e.g. for an accounting package: inventory, credits, debits, invoicing, payroll). Each part will need to
pass/obtain information to/from another part (e.g. sales affect inventory etc.). In addition, some processes provide services to other processes.

For these cases, processes need to communicate. Instead of communicating through a repository, user processes can directly exchange information with each other using some form of interprocess communication (IPC).

Several forms of interprocess communication have been developed ranging from simple shared memory methods to message passing to remote procedure call (declare later). They enable a process to communicate data to another process, allow a process to declare its intention to get data from multiple processes, and notify a receiving process when the data arrives, either by unblocking the receiver if it is waiting or invoking an asynchronous callback, if it is not.

There are many methods of intercommunicating information between processes. The familiar ones are:

(i) Files

A file is a collection of related information that is stored on a disk and that has a unique name. Files are the most obvious way of passing information. One process writes a file, and another reads it later. It is often used for IPC.

(ii) Pipes

A pipe (also called a FIFO) is an object that can act like a file, that is can be written to and read from. We can write bytes to the pipe. When we read from a pipe, we get the same bytes that were written and in the same order as they were written. A pipe is useful when we want to write from one process and read from another. With pipe, there are no fixed – size blocks, for example, the process writing the pipe can write in unit of 100 bytes and the process reading the pipe can read in units of 20 byte (see Fig 3.5).
Many OS provide a method of connecting the output *stream* of data from one process to the input of another; this is known as a *pipe* under Unix. A pipe avoids the use of a temporary file. It is *unidirectional* or *bidirectional pipes*. With pipes, one of the most IPC problems can occur. This is a deadlock, which will discuss later. That because pipes can only hold a finite amount of data, both pipes may fill up and then both processes are blocked writing.

![Pipe buffer diagram](image)

**Fig 3.5 Pipe**

(iii) **Shared Memory**

It is the fastest IPC facility. The operating system maps some pages into the same logical locations in two or more processes as shown in Fig 3.6. It is easy to do with a paged architecture. However, the operating system needs to keep a *link count* (like for shared files), so that the page can be freed when the link count becomes zero.

![Shared memory diagram](image)

**Fig 3.6 Shared memory within two processor**
(iv) **Message Passing**

An alternate approach is message passing, which can be implemented as system calls and therefore can be made available to existing languages through libraries. Message passing is based on two primitives:

```c
send(destination, &message);
receive(source, &message);
```

Message passing is useful in multiprocessor and distributed systems as well as uniprocessor environments; the calls are the same, just the implementation changes.

### 3.6 Types of IPC

Basically, there are five types of IPC:

1) **Blocking Communication:**

A blocking communications is a communication that might block if a buffer is not available, the network is not available, the receiver is not ready to receive the message, or the receiver has not yet replied to the message. For the send, this means that the call won’t return until the message data have been buffered or sent for example, until the memory referenced in the call is available for re-use. For the receive, this means that the call won’t return until the data has been received into the memory referenced in the call.

2) **Nonblocking Communication:**

Is a communication; that will not block if there is no message to send or receive but instead will return with an error code. This means that the call to send receive may return before the operation completes. For example, if the machine has a separate communication processor, a nonblocking send could simply notify the communication processor that it should begin composing and sending the message.
3) **Reliable Blocking sends:**
It is a blocking communication likes, send call won’t return until the message data have been buffered or send. But the sender insist that the message will be buffered by the receiver’s kernel and then it can release.

4) **Explicit Blocking Send:**
Alternately reliable blocking send, in explicit blocking send, sender can release just after reception (never mind by which).

5) **Request and Reply:**
This is used with a client/server model, which is architecture for operating systems or any systems of processes. In this communication, it is usual to “have one process that “owns” the resource or service and is in charge in managing It.” [6] This process which is called a server process, accepts request from the other processes (client processes) and then allocate the resources to them as needed.
Chapter 4

Process Synchronization Techniques
4.1 Introduction

Concurrently running processes without any sort of data protection scheme can manipulate data and produce an incorrect result in addition to data inconsistency. Moreover, if the outcome of an execution depends on the particular order in which the access takes place, a **race condition** can occur. To safeguard against this, we need to implement some sort of synchronization of the processes.

The most important sort of problems that are related to interprocess communication and must be catered for are:

4.2 Race Condition

A race condition is a situation where two processes either communicate or share memory while they are running and the output of one or both of the processes depends on the relative speed of the two processes. The reasons the outputs are different depending on the relative speed is because in some cases the output is wrong. This situation may involve the whole system within big problem if it is not been prevented.

![Fig 4.1 Two processes want to access shared memory](image-url)
Figure 4.1 shows this situation where two processes P1 and P2 want to access shared memory. The last value of shared memory location depends on which of the process wins the race to update the variable.

A race condition may be prevented by finding some way to prohibit more than one process from reading and writing the shared data at the same time.

4.3 Critical Section

A critical section is a part of a program in which it is necessary to have exclusive access to shared data. Only one process may be in a critical section at any one time. This segments of code are called Critical Section (CS) and the task to make CS behave as atomic is known as Critical Section problem. We can avoid race conditions by defining critical sections.

Generally solutions of CS problem can be subdivided into three groups: Hardware solution, Software solution, and Mixed solution.

4.3.1 Hardware Solution

The main idea of the hardware solution is to disallow interrupts. By switching off interrupts (or more appropriately, by switching off the scheduler) a process can guarantee itself uninterrupted access to shared data. Disable interrupts has advantages of does not taking any programming effort and fast.

But it has many drawbacks: masking interrupts can be dangerous. There is always the possibility that important interrupts will be missed, also it is not general enough in a multiprocessor environment, since interrupts will continue to be serviced by other processors, so all processors would have to be switched off; and it is too complex which needs special hardware. We only need to prevent two programs from being in
their critical sections simultaneously if they share the same data. Also one of the hardware solutions to critical section is to use a method called test set and lock.

4.3.2 Software solution

This solution allow interrupts happen, but do not let two processes that shared the same data to enter their critical section at the same time. In addition to provide software routine, that will act like CPU for CS's (i.e. will let them be executed one at a time.)
The advantage of this method is centered on no need in special hardware, does not affect processes that are not concurrent and works for multiple processors. The main disadvantage is that, it is very slow and complex.

4.3.3 Mixed Solution

This is to allow interrupts, but create a set of special uninterrupted hardware instructions that will serve as an entry for CS.
This solution is useful because it does not affect processes that are not concurrent, works for multiple processors and faster and easier to program. But it is hardware dependent.

All the solutions of Critical Section problem should satisfy following requirements:

**Mutual Exclusion:** CS can be executed by only one process at a time.

**Progress:** If no process executes CS and a number of processes want to enter CS, then scheduling decision should choose a process among those that have not executed CS yet, and this decision should be made in the finite amount of time.
**Bounded Waiting**: If process Pi requests to enter CS, there exists the upper bound M, such that no more then M processes can enter CS before Pi.

Four criteria for a good solution to race conditions and other multiprocessing nastiest.

- No two processes can be in their critical sections at the same time.
- No assumptions can be made about the speed or number of CPUs.
- No process running outside one of its critical sections can block another process.
- No process should have to wait forever to enter one of its critical sections (starvation).

### 4.4 Mutual exclusion

Mutual exclusion is a mechanism for avoiding race conditions by preventing two processes from running in their critical section at the same time.

Processes running on the same system may want to access their critical sections, or shared resources, at the same time. A mutual exclusion policy would ensure that only one process at a time is in its critical section. “For instance, two processes, P1 and P2, are running on the same computer. Process P1 enters its critical section, and before it complete its work process P2 is scheduled to run. P2 runs until it reaches its critical section, but may go no further until P1 exit its critical section. In this case, P2 is blocked until P1 leaves its critical section.”[4]

There are several methods to enforce mutual exclusion. All of these methods emphasis to the process that want to enter its critical region must first of all to check if there are other processes have already been entered there critical section. These methods departed into two categories depending on the behavior of blocked processes. There are those that require *busy waiting*, and those that do not.
4.4.1 Busy Waiting

A process or processor is busy waiting if it is waiting for an event by inquiring, over and over again, if the event has occurred. For example, a processor can busy wait for memory cell to become non-zero by reading it again and again and checking if it is still zero.

A software solution of the mutual exclusion is to use a lock variable which must be tested by each process before it enters its critical section. “If another process is already in its critical section, the lock is set to logic 1, and the process currently using the processor is not permitted to enter its critical section. If the value of the lock variable is 0, then the process enters its critical section, and it sets the lock to 1. The problem with this potential solution is that the operation that reads the value of the lock variable, the operation that compares that value to 0, and the operation that sets the lock, are three different atomic actions or independent instruction. With this solution, it is possible that one process might test the lock variable, see that it is open, but before it can set the lock, another process is scheduled, runs, sets the lock and enters its critical section. When the original process returns, it too will enter its critical section, violating the policy of mutual exclusion.”[4]

The only problem with the lock variable solution is that the action of reading-then-writing variable is done in two separate instructions. An efficient way to handle the problem is to provide atomic read-write.

These steps can be combined, with a little help from hardware, into what is known as a TSL or TEST and SET LOCK instruction. A call to the TSL instruction copies the value of the lock variable and sets it to a nonzero (locked) value, all in one step. While the value of the lock variable is being tested, no other process can enter its critical section, because the lock is set. TSL is used with two operations, enter_region and leave_region, (see figure 4.2). If a process is interrupted after executing the TSL, there is no danger that another process might enter its critical section. This is so because the lock variable is set to a non-zero value while the original process is
testing it. Another advantage of the TSL instruction is that it works on machines with many processors as well. [4]

```
enter_region: (executed when a process wants to enter its critical section)

tsl register, lock       // copy lock to register and set lock to 1
cmp register, 0          // see if lock variable was set
jnz enter_region         // if lock was set, loop
ret                      // enter critical section

leave_region: (executed when a process wants to leave its critical section)

mov lock, 0              // store a 0 in lock variable
ret                       // done
```

**Fig 4.2 TSL example**

It was clear that from the previous pseudo code, the jnz instruction involve the process in testing loop unless the variable become zero and then has the ability to enter its critical region. Mutual exclusion policies that require busy waiting waste valuable processor time, and in some cases can lead to situations where a process will test the lock variable forever, a very wasting time of CPU.

### 4.4.2 Sleep and Wakeup

The previous solution is just a waste of the CPU time, so it is not workable solution. The better one is to eliminate this loop operation and if a process has no way to enter its critical region then go on idleness or blocking situation until being permitted. Two primitives, *Sleep* and *Wakeup*, are often used to implement blocking in mutual exclusion.
Essentially, when a process is not permitted to access its critical section, it uses a system call known as *Sleep*, which causes that process to block. The process will not be scheduled to run again, until another process uses the *Wakeup* system call. In most cases, Wakeup is called by a process when it leaves its critical section if any other processes have been blocked. In sleep and wake up example, a process wanting to enter its critical section must first check to see if any other processes are currently in their critical section. If so, the process calls the Sleep() system call, which causes it to block. Upon leaving its critical section, a process must check to see if any other processes have been put to sleep, waiting to enter their critical section. If a process is waiting, the process now exiting its critical section must wakeup the sleeping process. Figure 4.3 depict sleep and wakeup example.[4]

```c
var occupied;              // 1 if critical section is occupied, 0 if not.
var blocked;               // counts the number of blocked process

Enter_Region:              // process enters its critical section
{
    IF (occupied) {       // if critical section occupied
        THEN
            blocked = blocked + 1;   // increment blocked counter
            sleep();                   // go to "sleep", or block
        }
    ELSE occupied = 1;     // if can enter critical section, increment counter
}
...

Exit_Region:                // process exits its critical section
{
    occupied = 0;
}
```
IF (blocked)
{
    THEN wakeup(process); // if another process is sleeping wake the
    // process up
    blocked = blocked - 1; // decrement blocked counter
}

Fig 4.3 sleep and wakeup example

Now consider what would happen if two processes, A and B, are using this method for mutual exclusion. “Process A enters its critical section, and before it leaves the critical section, process B is scheduled. Now process B attempts to enter its critical section. It sees that process A is already there, so it increments the "blocked" variable in preparation to go to sleep. Just before it can call the Sleep primitive, process A is scheduled again. Process A exits its critical section. Upon exiting, it sees that the "blocked" variable is now 1, so it calls Wakeup on process B. But process B is not yet asleep. The Wakeup is lost, so when process B is scheduled again, it will call Sleep, and block forever.”[4]

Sleep and Wakeup is simple, and simple to implement. One call to wakeup will wakeup all sleeping processes on this particular address. But as we see above it has race hazards, difficult to use, and very low level process synchronization primitives. Below we will study three of implementation of mutual execution, which do not require busy waiting: semaphores, monitors, and message passing.

4.4.3 Semaphores

Semaphores are solution to mutual exclusion without busy waiting that proposed by E.W. Dijkstra in 1965. A semaphore is a synchronization primitive with two operations: wait(see fig 4.4) and signal(see fig 4.5). Semaphores have to be named in some way just like array and pipe. It will be an integer identifier (S) chosen by the
user processes and will be global to all processes that use the same semaphore. Modifications to the semaphore variables S in wait and signal operations must be executed indivisibly. That is, when one process modifies a semaphore value, no other process can simultaneously modify that same semaphore value. In each process wait(s) is executed just before its critical section signal(s) is executed just after leaving critical section.

```
function wait(S)
{
    while(S <= 0) ;        // no-op
    S--;                   
}
```

**fig 4.4 Classical Definition of Wait**

```
function signal(S)
{
    S++;                   
}
```

**fig 4.5 Classical Definition of Signal**

Semaphores allow a Sleep and Wakeup mutual exclusion policy to be implemented without the risk of loosing a Wakeup. Basically,
a semaphore is a new type of variable. Semaphores can have a value of 0 (meaning no Wakeups are saved), or a positive integer value, indicating the number of sleeping processes. Two different operations can be performed on a semaphore, \textit{DOWN} and \textit{UP}, corresponding to Sleep and Wakeup. \textit{DOWN} executed on semaphore variable \textit{S} before entering critical section, and \textit{UP} executed when leaving critical section. So, the state of the semaphores after these actions is changed.

On semaphore, both tool variable and tool procedure are predefined. The predefined procedures are \texttt{initialize}(S,n), \texttt{wait}(S), \texttt{signal}(S), that can be implemented in different ways. Operating system/hardware make sure, that all these instructions are atomic and the variable value of semaphore can be modified by one process at a time. All synchronization code is put outside of semaphore.

Semaphores can be exist in two types:

\textbf{Binary}: binary semaphore is either busy or not and take value only of 0 or 1; so they are easier to implement. It is intended for mutual execution where only one process at a time can be in its critical section. Figure 4.6 below explains a pseudo code for this one. Semaphore S consists of integer Val and two binary Semaphores S1 and S2.

\begin{verbatim}
initialize(S,n):initialize(S1,1)
initialize(S2,0)
Val = n
wait(S):  wait(S1)
Val=Val-1
if(Val <0)
{
    signal(S1)
    wait(S2)
}\end{verbatim}
Chapter 4                                                                  Process Synchronization Techniques

}  

signal(S1)  

signal(S) : wait(S1)  

Val=Val+1  

if(Val <= 0)  

signal(S2)  

else  

signal(S1)  

fig 4.6 Binary semaphore  

Counting semaphore: there are some situations where one wants to limit the number of processes in the same state but greater than one. For example, if there are three printers, then you might allow three processes to be printing at the same time.  

Counting semaphores can be implemented in different ways:  

Naive (spinlock): are useful in multiprocessor environment and for many processes with short waiting for semaphore. No context switches is required. The process want to enter its critical section or use the shared resources, check the variable S, if it has value less than zero-which mean that the critical section now being used or all the shared resources was being already allocated- then the process go on waiting loop checking the S value. It requires busy waiting, which means that process is wasting the CPU time on empty loop cycles, this is the main disadvantage. The figure 4.7 below illustrates the implementation of naive semaphore.  

initialize(S,n):S=n  

wait(S): while(S<0) wait S = S-1  

fig 4.7Naive (spinlock) implementation  

Blocking(queue): (see fig 4.8)if process has to wait, it blocks itself and is placed to Semaphore Queue (it actually works like blocking for I/O).
Semaphore Queue (S.queue) usually uses FCFS (First Come First Save), but can use any other scheduling algorithm. Semaphore S consists of integer Val and semaphore queue SQ and instructions are implemented as illustrated below.

\[
\text{initialize}(S,n) : \text{Val}=n \\
\text{SQ.Empty()} \\
\text{wait}(S) : \text{Val}=\text{Val}-1 \\
\text{if(Val < 0)} \\
\quad \{ \\
\quad \quad \text{SQ.Insert(process)} \\
\quad \quad \text{block(process)} \\
\quad \} \\
\text{signal}(S) : \text{Val}=\text{Val}+1 \\
\text{if(Val <= 0)} \\
\quad \{ \\
\quad \quad \text{process = SQ.Extract()} \\
\quad \quad \text{wake-up(process)} \\
\quad \}
\]

**fig 4.8 Blocking(queue) implementation**

This implementation is useful in single processor environment when actual critical section is long and saves CPU time, but does not work for multiple processes.

Not like the previous one, context switches take time required, so need to turn off interrupts for quite a while to execute wait/signal.

As was declared, the value of a semaphore need not always be 0 or 1. Semaphores can have a value greater than one. The original value of the semaphore dictates the total number of processes that can enter a specific region at a time. For instance, a semaphore with an original value of 4 would allow four processes to enter the section of code they protect.
Its advantage is centered on making synchronization programming easier and faster, but sometimes is not flexible enough to solve difficult synchronization problem.

General problems related to this policy is founded on hardware level which can't make wait(S) atomic, it need software implemented CS to implement semaphore. This software is a low-level system calls that programmed by the user. So, it is venerable to be put into wrong order which may causes in a deadlock presence. What we need, is to use higher level implementation for mutual execution.

Also, deadlocks are possible by improper usage of wait(S) and signal(S) in application, starvation caused by improper implementation of wait(S). But we can get rid of starvation by putting the blocked processes in S.queue.

### 4.4.4 Monitors

They are a high level construct found in some programming languages which contain variables, procedures, and data structures. Monitors closely resemble the protected classes and modules, which are found in languages like C++ and Ada, in that they only allow, access to the variables they contain through the procedure they contain. Only one process may execute a procedure within a monitor at any one time. This is what allows monitors to enforce mutual exclusion.

On the monitor policy, the set of tool variables and tool procedures are defined by programmer. Monitor provides the programmer special type of variable: condition, which is in a way as semaphore tool variable and can be modified only by signal() and wait() atomic operations. Monitor structure ensures, that only one process at a time could be active within monitor.

Synchronization code should be supplied by programmer within monitor and makes use of conditional variables
Monitor is more flexible but has many drawbacks that are, more sophisticated and unsafe structure, has software overhead and uses lots of system resources through queue and condition variables.

The general problems related to this policy are the use of queue always that make processes suffer from improper queue scheduling, all synchronization is provided by programmer (even bigger risk of deadlock and starvation) and excessive use of condition variables.

When a process calls a procedure in a monitor that is already occupied, it performs a WAIT operation on e condition variable. This will cause the process to block, until the other process, which currently occupies the monitor, leaves the monitor and performs a SIGNAL on the condition variable.[4]

4.5 Message Passing

The different type of mutual execution solution is useful in the situation of presence of shared memory. But in the distributed systems with no common memory, we have to find another solution that has the ability to communicate through different machines on the network. Message passing has this capability.

Two primitives, SEND and RECEIVE are used in the message passing scheme. The SEND primitive sends a message to a destination process while the RECEIVE primitive receives a message from a specified source process. Message passing works on distributed systems because these messages can be sent from machine to machine through a network.

Typically, each process has a mailbox, which is a buffer that receives all the messages were sent to that process. The destination of the SEND and RECEIVE system calls is a process’ mailbox, not the process itself. SEND takes two arguments, the destination mailbox and the message. RECEIVE also takes two arguments, the identifier (ID) of the sender, and the message.

Several methods exist for implementing mutual exclusion with message passing. One method uses processes to regulate access to shared resources. A resource control
process has a variable which records the status of a resource (free or busy), and a buffer which contains request messages from other processes.

What happens when process P1 wants to access a shared resource? Three messages are sent here. Message (1) is a request for access to the resource. Process P1 then blocks, waiting for a reply from the resource control process. Message (2) is the reply sent by the control process. This message wakes up P1, and P1 now uses the resource. Message (3) informs the resource control process that P1 is done using the resource (see fig 4.9). [4]

```
P1:
    send(control_process, "Give me resource");
    receive(control_process, Message);  // block while waiting for a
    // reply
    {
        ...Critical Section...
    }
    send(control_process, "Done with resource");
```

**fig 4.9 Send\Receive Primitive**

So, no process can enter its critical region unless be given the permission to do this. And no permission will be given if there is a process exists within the critical section. By this scheme the work can be organized between all processes.

### 4.6 Deadlock

A deadlock is a situation where a group of processes are all blocked and none of them can become unblocked until one of others become unblocked. The simplest deadlock is two processes, each of them is waiting for a message from the other.

Fig 4.10 show an example of the application code that causes deadlock due to synchronization error is:
From the above example, we see that process P1 at the beginning of its job is waiting process P2 to send a signal, where as process P2 also waiting P1 to send a signal. Each of two processes wait another one to send a signal, so both of them will enter a blocking state and a deadlock will occur.

To understand this situation let us consider the European road rule, which says: “on minor roads one should always wait for traffic coming from the right. If four cars arrive simultaneously at a crossroads then, according to the rule all of them must wait for each other and none of them can ever move. This illustrates deadlock problem. It is the stale-mate of the operating system world.”[7]

Here is another real operating system example of deadlock occurrence. Deadlock can occur among group of two or more processes, that each process holds at least one resource while making a request on another. The request can never be satisfied because the requested resource is being hold by another process that is blocked waiting for the resource that the first process is holding. Figure 4.11 illustrates this deadlock situation among three processes on three resources of different type. Each
So, a deadlock can occur when a number of processes are waiting for an event which can only be caused by another of the waiting processes.

A deadlock occur if all of the following conditions hold simultaneously:

1. **Mutual exclusion**: there exists a resource that can be accessed by only one (actually limited number of processes) process at a time and this resource must be non-sharable. If the resource can be shared, there is no reason to wait.

2. **No preemption**: The processes can not be forced to give up the resources they are holding. The only two ways resource can be released is by process, that holds it or through process termination.

3. **Hold and wait**: there exists process that holds at least one resource and is waiting for another resource (generally some critical number of resources are held by processes, which are waiting for another resources.)

4. **Circular waiting**: there exists a set of processes \{P_1,...,P_n\}, such that each process holds at least one resource and requiring another one. Process P_1 is waiting for resource hold by P_2, P_2 is waiting for P_3, …., P_n is waiting for P_1.

There are likewise three methods for handling deadlock situations:

1. **Prevention**. We can try to design a protocol that ensures that deadlock never occurs.

2. **Recovery**. We can allow the system to enter a deadlock state and then recover.

3. **Ostrich method**.

### 4.6.1 Deadlock prevention

A deadlock can be prevented if we ensure that at least one of the four condition that lead to deadlock is never satisfied, then deadlock will be structurally impossible.
To attack mutual exclusion, no resource will ever assigned executively to a single process. We can do this by spooling everything. But, since not all devices can be spooling (e.g. disk space), this attacking has a main disadvantage.

To handle hold-and-wait condition, the process, which is holding resources, must be preventing from waiting for more ones. This can be done by requesting all resources initially. If one needs more, then it must release all the resources it own currently and begin its request from the beginning.

This is difficult to achieve, so as many processes don’t know how many resources they will need from the beginning. Furthermore, following this method, each process will hold all the resources until it finish. So, this is a resource waste. Also, this will put a burden on the programmer.

No preemption condition can be prevented by taking the resource away from the current process that own it and give it to requested one.

Preventing circular waiting takes place by two ways: one way is to assign one resource at a moment. If process needs other recourses, it must first release the one it holds. This way is unacceptable, for example, if a process want to print out a file from tape through a printer, it must hold them both. The other way is to order all the resources numerically. Process request resources whenever they want, but the requesting resource must has a number greater than the number of the one it hold. If the process requests a resource with smaller number, this led a process to finish. It is a good solution but we find some difficulties. It is not easy to find an optimal ordering for the resources and satisfied every one.

4.6.2 Deadlock avoidance

Deadlock prevention requires a system overhead.
The simplest possibility for avoidance of deadlock is to introduce an extra layer of software for requesting resources in addition to a certain amount of accounting. Each time a new request is made, the system analyses the allocation of resources before granting or refusing the resource. The same applies for wait conditions.

The problem with this approach is that, if a process is not permitted to wait for another process - what should it do instead? At best the system would have to reject or terminate programs which could enter deadlock, returning an error condition.

Another method is the following. One might demand that all programs declare what resources they will need in advance. Similarly all wait conditions should be declared. The system could then analyze (and re-analyze each time a new process arrives) the resource allocation and pin-point possible problems.[7]

### 4.6.2 Deadlock Detection and Recovery

The detection of deadlock conditions is also a system overhead. At regular intervals the system is required to examine the state of all processes and determine the interrelations between them. Since this is quite a performance burden, it is not surprising that most systems ignore deadlocks and expect users to write careful programs.

To recover from a deadlock, the system must either terminate one of the participants, and go on terminating them until the deadlock is cured, or repossess the resources which are causing the deadlock from some processes until the deadlock is cured. The latter method is somewhat dangerous since it can lead to incorrect program execution. Processes usually wait for a good reason, and any interruption of that reasoning could lead to incorrect execution. Termination is a safer alternative.[7]
4.6.3 Ostrich method

This method deal with deadlock lie this: if any problem occur, ignore it or them all together and pretend that there is no problem at all. So, we have to pretend that deadlocks will never occur and live happily in our ignorance. This method used by most operating systems like Unix.
Chapter 5
Synchronization in Distributed System
5.1 Introduction

Distributed systems are a collection of loosely coupled processors that do not share common clock or memory. They all linked through communication links. Each machine has its own operating system and own memory. All the users of these machines can communicate with each other, sending mails and files and have the ability to share resources. These systems have many advantages over a centralized one, resource-sharing, speed up the computation and improve data reliability and availability. Reliability means that if one of machines crashed; the other will go on working. Availability denotes that all the users on their machine will show the data.

5.2 Communication in Distributed System

As was mentioned, distributed memory multicomputer system consists of multiple computers (nodes) interconnected by specific design network. Each node is an autonomous computer consisting of a processor, local memory and sometimes attached to disks or I/O peripherals. In these environments, systems must be able to synchronize and share data through some mechanism for remote interprocess communication (IPC).

Internode communication is carried out by passing messages through the network. Message passing demands special hardware and software support. The message-passing network provides point-to-point static connections among the nodes. All local memories are private and accessible only by local processors. Messages are a low-level mechanism; it is a piece of information that is passed from one process to another. A mailbox is the a place where messages are stored between the time they are sent and the time they are received (The source or destination of a message.)

Message passing has two basic operations (see Fig 5.1):
**send**: copy a message into a mailbox. If the mailbox is full, wait until there is enough space in the mailbox.

**receive**: copy message out of mailbox, delete from mailbox. If the mailbox is empty, then wait until a message arrives.

![The mailbox](image)

Fig 5.1 The mailbox

There is only one mailbox per process. Use the process name in send, no name in receive. This is simple but restrictive. No strict mailbox-process association, use mailbox name to designate recipient of a message. It allows multiple mailboxes per process, but trickier to implement (e.g., UNIX).

Communication can use buffering, a queue of messages waiting to be received. The buffer can be zero capacity, sender must wait (block) for receiver, bounded capacity, sender must wait for receiver if buffer is full or unbounded capacity, infinite length so sender never waits.

Many interesting systems have been built around the notion of message passing, most notably Mach and Microsoft Windows.

There are two general styles of message communication:

1-way: messages flow in a single direction (UNIX pipes, producer consumer.) and,

2-way: messages flow back and forth (remote procedure call, client/server).

We use messages due to many reasons, they fit application paradigm, e.g., UNIX filters, database server and the communicating processes can be isolated. The later one lead to many advantages, less error-prone because no invisible side effects: no
process has access to another's memory, processes might not trust each other (UNIX process and OS), and allow for independent development. Once the interface is agreed upon, anyone can go off and write clients and/or servers.

Also and since processes may be running on different machines, messages are extremely useful in networking.

There are two problems associated with using distributed memory systems: Low machine efficiency due to interprocessor communication and Difficulty to program. These two problems can be avoided by producing some methods that are: how to reduce communication overhead, increase the portability, and how to simplify distributed memory programming.

5.3 Synchronization in Distributed System:

5.3.1 Introduction:

In single CPU systems, all synchronization problems are generally solved using methods such as semaphores and monitors. These methods are not well-suited to use in distributed system because they need a shared memory. Synchronization in distributed system is more complicated than in centralized one because the former have to use distributed algorithm. In this section will examine back with single CPU, but now in the context of distributed systems.

5.3.2 Event Ordering

In many applications an ordering of events consider as one of the important issues; by which some specific programs decide to do additional task or not. For example, if there is a source code was already compiled, later on the user make the same
program-in the same file – more advance. Therefore, the compiles must know by some method this event was happened after the first compilation so as to recompile. In centralized system with single common memory and clock, it is easy to order the events occurrence. This is more complicated with distributed systems since there is no common clock and memory. It is some times impossible to determine which event happened before. Therefore, it is better to use happened-before relation which “is only a partial ordering of the event in distributed systems” [2].

5.3.2.1 The Happened – Before Relation:

The happened – before relation denoted by the operator (→). It can be defined on a set of events as follow: [2]

1. In the same process, A→B means A was executed before B.
2. With two processes, A→B means A an event of sending message, and B an event of receiving it.
3. Happened – before is transitive relation, so if A→B and B→C then A→C.

From above definition we can say a happened-before relation can be defined only with a cooperative processes. With independent processes [e.g. A B], which A is not affect on or affected by B, we can’t write A→B. In this case it is called concurrency processes. In this relation it is never mined which one occur first they are just independent events.

5.3.2.2 Implementation

Since each process in distributed systems has its own clock, all of these processes may be in different speed. So, we must implement the happened – before relation without using physical clocks. We can define a logical clocks (LC), which can be
implemented as a simple counter that is incremented by a constant rate, between each two successive events.

To define the happened – before relation we must associate a timestamp with each event, which is the logical clock of that event, then we can define the global ordering requirement:” For every pair of events A and B, if A→B, then the timestamp of A is less then the timestamp of B” [2]. The timestamp for each event is a unique number unless the events are in concurrency relation.

With two events (A and B) in same process (Pi), it is easy to assign a unique timestamp for every event. If A→B, the LCi (A)<LCi (B). But it is a little difficult with distributed systems. Suppose that we have two events, A is sending message in process P1, and B is receiving message in process P2. Naturally, the relation must be like this, LC1 (A) < LC2 (B). Unfortunately, in most cases this relation dose not be ensured because it may be that the processor for P2 is slower than processor for P1.

How do we resolve this problem?

When event B is occurred with timestamp t, the processor for P2 checks its logical clock; if it has LC2 (B) < t, then it must advance its logical clock so as to be LC2(B) = t+1. So, all we need a mechanism to advance the local logical clock when received a message has timestamp is greater than the current value of the local logical clock.

**5.3.3 Mutual Exclusion**

Systems involving multiple processes are easily using critical region, that if one process enter its critical region, no other one dares to do so. In single processor system, this is achieved by using semaphores, monitor and similar constructs. These methods can not be implemented in distributed systems that require shared memory. We will look now at an example of how a critical sections and mutual exclusion can be implemented in distributed systems.
5.3.3.1 Centralized Approach

The most straightforward way to achieve mutual exclusion in a distributed system is “to stimulate how it is done in a one-processor system. One process is elected as the coordinator [e.g. the one running on the highest network address].”[1] Whenever a process wants to enter a critical region, it sends a request message to the coordinator telling which critical region it wants to enter and asking for permission.

If another process asks for permission to enter the same critical region the coordinator just refrains from replying blocking second process or send a reply saying, “permission denied.”[1] Either way it queues the request from the second process until the first process finished. When process exists the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of given requests, sends that process an ok message. Fig 5.2 shows us how process 0 requests the coordinator C to enter its critical region and get an ok message. During its work process 1 ask the same request, so no permission given to it and being queued until process 0 exit.

**Fig 5.2 Centralized approach**

It is easy; the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order they are received. No starvation and easy to implement. It can be used for more general resource allocation.
This approach also has drawback. The coordinator is a single point of failure; if it is crash the entire system may go down. In addition, “in a large system, a single coordinator can become performance bottleneck.” [1]

5.3.3.2 Distributed Approach:

This approach works as follow: When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the time it requires within the critical section (timestamp). Then it sends to all other processes, including itself. The message assumed to be reliable, every message is acknowledged. If reliable group communication available, it can be used instead of sending individual message.

When a sending process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished.[1]

1. If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
2. If the receiver is already in the critical region, it does not reply. It queues the request.
3. If the receiver wants to enter the critical region, but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

After sending request, a process sits back and waits until everyone else has given permission. Upon existing critical region, it sends ok message to all processes on its queue, and deletes all of them from the queue. Figure 5.3 below describe this approach.
Processes P0 and P2 want to enter their critical region, so they send an individual message containing a time stamp of 5 and 7 respectively. P1, which do not need to enter its critical region, send an ok message to both of them. But P0- which wants to- wins; because it has the lesser timestamp. P2 was being queued until has a permission from P0.

With this approach mutual exclusion is guaranteed without deadlock or starvation. No single point of failure exists. But if one failed, no reply is given. This situation can be interpreted as denial of permission, thus blocking all subsequent requests. Another problem; it is best for small group of processes.

The drawbacks of using this approach are that it is slower, more complicated, more expensive and less robust.

5.3.3.3 A Token Ring Approach

Physically, in a ring network each process is assigned position in the ring and connected to exactly two other sites. Each process in the ring must know who is the next after it. The message travels from one process to another through the network until reaches the original creator.

When the ring is initialized, special process is given a token and then circulates around the ring from process to another in point-to-point massage. When the token is acquired, it checked to see if there is any attempting to enter a critical region. If so, the process enters the region and when have it, it passes the token along the ring. It is not permitted to have another chance to enter its critical section. The figure 5.4 below
shows how a ring is configured in a physical and a logical order. P1 holds the token, so it can enter its critical section if it want or pass it over the network through P2. With this approach, starvation never occurs since one process takes the token at instance and then circulates it among processes in a well-defined order. If a token is lost, it may be regenerated. The problem is how to detect the lost of token. Since there is no limited time between successive appearance of the token, the interoperation of lost may be as this “may somebody still be using it.”[1]

![Network line](image)

*An order of process in network*

**logical ring constructed in software**

| 0 | 1 | 2 | 3 | 4 |

Fig 5.4 Toke ring approach

There is no problem when a process is dead, recovery is easier. It detected when one passes the token to the next and fail. At this point the dead process remove from the group. The token passed over the dead process to the one after that. To do so, requires every one know the ring configuration.

### 5.3.4 Election Algorithms

As was declared on the previous section, various distributed algorithms required one process acts as coordinator, initiator, sequencer, or perform some special role. The algorithms that determine where a new coordinator should be restarted are called **election** algorithms.

For maximum flexibility, any process should be able to coordinate. Each process given an identification number, the highest one is a coordinator. Since any process can do the job, there is no a priori reason for choosing one over another. Hence some
arbitrary scheme must be applied: typically, each process is assigned a unique number, and the process with the highest number is made coordinator. When one coordinator dies, the problem is to determine which of the other processes are still alive, and which of those has the highest number.

Election Algorithm attempts to locate the process with the highest process number and designate it as coordinator.

The crash of the coordinator can be observed only when some other process sends a request to the coordinator and gets no reply. That process is responsible for initiating the election. Of course, two processes may start an election concurrently, so any election algorithm must take that into account.

It has to be assumed that each process knows the identities of all others involved in the algorithm.

5.3.4.1 Bully Algorithm:

When a process notices that the coordinator not responds, it initiates an election message. Initiating process sends #election# to each process with a higher number than itself. If all of those that are still alive will send a reply. If the process receives no replies, it can assume that it is the highest-numbered, living process and should become coordinator. It announces its new role with a coordinator message broadcast to all processes in the group. A process receiving an election message must have a higher number than the sender and so has more right to be coordinator. By sending a reply, it removes the sender from the possible candidates. “It can then repeat the election process for itself, until eventually the mantle of coordinator passes to the process with the highest number.” [8]

Whenever a process restarts after a crash, it holds an election just in case it has become the highest-numbered process in the group. This might take away the coordinator’s role from some other process, hence the name bully algorithm.

A process, P, holds an election as follow: [1]
1. P sends an ELECTION message to every process with a higher number
2. If no one responds, P wins the election and becomes coordinator.
3. If one of the higher ups answer, it takes over and P’s job is finished.

5.3.2.2 Ring Algorithm:

This are based on the use of rings and does not use a token. Processes arranged in logical ring. Initiator passes election message to neighbor. Message passed around ring collecting processes numbers. When message returns, contains all live processes. Initiator broadcasts coordinator message.

By details, this algorithm work as follow: When the coordinator is not working, process which notice that builds an ELECTION message and pass it to its neighbor, containing its own process number. The message is circulated around the ring, each process adding its own number before passing the message on. By the time the message has gone full circle, it will contain the numbers of all processes still alive. The initiating process then selects the highest number and circulates a coordinator message to indicate that process is the new coordinator. At each step, the sender adds its own process number to the list in the message.

Each process must know the complete configuration of the ring, so that if a process gets no acknowledgement from its neighbor (If the later process is crash,) it can pass the message on to the next process in the ring, or the one after that, and so on. Upon the arrival of the message to original generator, it notice this event and then build COORDINATOR message and circulate it within the ring informing who is the coordinator and who are now within the ring.

“If two processes notice absence of coordinator, simultaneously build ELECTION message, and convert them self into COORDINATOR message, both will be removed. No harmful to have extra message circulating but waste a little bandwidth. The algorithm involves fewer messages than the BULLY algorithm.” [1]
Communication delay may again interfere with the algorithm in that a heavy loading of a process's node or the communication path to it may cause it to be bypassed. In both cases the algorithms are specified at a higher level than this.

### 5.3.5 Distributed Deadlock

Deadlock can occur whenever two or more processes are competing for limited resources and the processes are allowed to acquire and hold a resource (obtain a lock). Thus preventing others from using the resource while the process waits for other resources.

Two common places where deadlocks may occur are with processes in an operating system (distributed or centralized) and with transactions in a database. The concepts discussed here are applicable to any system that allocates resources to processes.

This is a deadlock: process A is waiting for a resource held by process B and process B is waiting for a resource held by process A. No progress will take place without outside intervention. Several processes can be involved in a deadlock when there exists a cycle of processes waiting for each other. Process A waits for B which waits for C which waits for A.

As with traditional system, four conditions must hold for deadlock to occur:[9]

1. *Exclusive use* – when a process accesses a resource, it is granted exclusive use of that resource.
2. *Hold and wait* – a process is allowed to hold onto some resources while it is waiting for other resources.
3. *No preemption* – a process cannot preempt or take away the resources held by another process.
4. *Cyclical wait* – there is a circular chain of waiting processes, each waiting for a resource held by the next process in the chain.
Chapter 5                                                              Synchronization in Distributed System

The structure of the system may allow additional complexities in the deadlock problem. The simplest model, *single-resource*, requires that a process have no more than one unfulfilled request. Thus a blocked process is waiting for only one other process and can be involved in at most one deadlock cycle.

In the multiple-resource model, also called AND model, a process is allowed to make several resource requests, and it is blocked until all of the requests are granted. Processes in this model can be involved in several deadlock cycles at once.

In the communication model, also called the OR model, a process makes several requests and is blocked until any one of them is granted. The AND-OR model allows a combination of request types, such as a request for resource X and either Y or Z.

The problem of deadlocks can be handled in several ways: Prevention, Avoidance, and Detection. “In prevention, some requirement of the system makes deadlocks impossible so that no runtime support is required. Avoidance schemes require decisions by the system while it is running to insure that deadlocks will not occur. Detection requires the most sophisticated runtime support: the system must find deadlocks and break them by choosing a suitable *victim* that is terminated or *aborted* and restarted if appropriate.”[9]

**5.3.5.1 Deadlock Prevention**

Prevention is the name given to schemes that guarantee that deadlocks can never happen because of the way the system is structured. One of the four conditions for deadlock is prevented, thus preventing deadlocks. One way to do this is to make processes declare all of the resources they might eventually need, when the process is first started. Only if all the resources are available is the process allowed to continue. All of the resources are acquired together, and the process proceeds, releasing all the resources when it is finished. Thus, *hold and wait* cannot occur.
The major disadvantage of this scheme is that resources must be acquired because they might be used, not because they will be used. Also, the pre-allocation requirement reduces potential concurrency.

Another prevention scheme is to impose an order on the resources and require processes to request resources in increasing order. This prevents cyclic wait and thus makes deadlocks impossible.

One advantage of prevention is that process aborts are never required due to deadlocks. While most systems can deal with rollbacks, some systems may not be designed to handle them and thus must use deadlock prevention.[9]

5.3.5.2 Deadlock Avoidance

In deadlock avoidance the system considers resource requests while the processes are running and takes action to insure that those requests do not lead to deadlock.

Avoidance based on the banker's algorithm, sometimes used in centralized systems, is considered not practical for a distributed system. Two popular avoidance algorithms based on timestamps or priorities are wound-wait and wait-die. They depend on the assignment of unique global timestamps or priority to each process when it starts. Some authors refer to these as prevention.

In wound-wait, if process A requests a resource currently held by process B, their timestamps are compared. B is wounded and must restart if it has a larger timestamp (is younger) than A. Process A is allowed to wait if B has the smaller timestamp. Deadlock cycles cannot occur since processes only wait for older processes. In wait-die, if a request from process A conflicts with process B, A will wait if B has the larger timestamp (is younger). If B is the older process, A is not allowed to wait, so it dies and restarts.

In timeout based avoidance, a process is blocked when it requests a resource that is not currently available. If it has been blocked longer than a timeout period, it is
aborted and restarted. Given the uncertainty of message delays in distributed systems, it is difficult to determine good timeout values. These avoidance strategies have the disadvantage that the aborted process may not have been actually involved in a deadlock.[9]

5.3.5.3 Deadlock Detection

Deadlock detection attempts to find and resolve actual deadlocks. These strategies rely on a Wait-For-Graph (WFG) that in some schemes is explicitly built and analyzed for cycles. In the WFG, the nodes represent processes and the edges represent the blockages or dependencies. Thus, if process A is waiting for a resource held by process B, there is an edge in the WFG from the node for process A to the node for process B.

In the AND model (resource model), a cycle in the graph indicates a deadlock. In the OR model, a cycle may not mean a deadlock since any of a set of requested resources may unblock the process. A knot in the WFG is needed to declare a deadlock. A knot exists when all nodes that can be reached from some node in a directed graph can also reach that node.

In a centralized system, a WFG can be constructed fairly easily. The WFG can be checked for cycles periodically or every time a process is blocked, thus potentially adding a new edge to the WFG. When a cycle is found, a victim is selected and aborted. [9]
Chapter 6
Implementation


6.1 Introduction

A common approach to process cooperation in computer network is to use a message
passing library, where a process uses the library calls to exchange message
(information) with another process. This message passing allows processes running
on multiple processors to cooperate in solving problems. One of the standardized
interfaces intended for use in programs that exploit the existence of multiple
processors by message passing is **Message Passing Interface (MPI)**.

MPI was developed in 1993-1994 by a group of researchers as well as vendors. The
main design goals for MPI were to establish a practical, portable, efficient, and
flexible standard for message passing.

“MPI is a library of functions and macros that can be used in C, Fortran, and C++
programs. It specifies point-to-point communication as well as collective one, in form
of various sends and receives calls; the ability to define derived (complex) data types,
and the ability to define virtual topologies to ease communication.” [3]

6.2 General MPI Program:

Every MPI program must contain the preprocessor directive.

```c
#include “mpi.h”
```

This file, mpi.h contains the definitions, macros, and function prototypes necessary
for compiling an MPI program. Before any other MPI functions can be called, the
function MPI_Init must be called, and it should only be called once. Its arguments
are pointers to the main function’s parameters. After program has finished, it must
call MPI_Finalize. This cleans up any “unfinished business “ left by MPI[3].

MPI provides the functions MPI_Comm_rank, which returns the rank of a process in
its second argument and MPI_Comm_size for determining the number of processes
excluding the program. Their syntaxes are:

```c
MPI_Comm_size(MPI_COMM_WORLD, int size)
```
And

\[ \text{MPI\_Comm\_rank(MPI\_COMM\_WORLD, \text{int rank})} \]
respectively.

The first argument is a communicator. It is a collection of processes that can send message to each other. So a typical MPI program has the following layout.

```
#include "mpi.h"

main (int argc, char *argv [])
{

MPI_Init (& argc, & argv);
MPI_Comm_size(MPI_Comm comm, int size);
MPI_Comm_rank(MPI_Comm comm, int rank);

MPI_Finalize();
}
```

### 6.3 Basic MPI functions:

The actual message-passing in our program is carried out by the blocking MPI function MPI_Send and MPI_Recv. The first command sends a
message to a designated process. The second receives a message from a process. Their syntaxes are:

MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)

The contents of the message are stored in a block of memory referenced by the argument message. The next two arguments, count and datatype, contain a sequence of count values, each having MPI type datatype. The arguments dest and source are, respectively, the rank of the receiving and the sending processes. The arguments tag and comm are, respectively, the tag and communicator. The tag is a user specified int that can be used to distinguish messages received from a single process. For example, suppose process A is sending two messages to process B, both messages contain a single float. One of the floats is to be used as data, while the other is to be used as request. In order to determine which is which, A can use different tags for the two messages. MPI guarantees that the integers 0 – 32767 can be used as tags. For the communicator argument, we can get away with using the predefined communicator MPI_COMM_WORLD. There is a wildcard, MPI_ANY_TAG that MPI_Recv can use for the tag. There is no wildcard for the communicator.

The last argument of MPI_RECV, status, returns information on the data that was actually received.

There are also a non-blocking MPI functions for send and receive. Their syntaxes are:
MPI_Isend (buf, count, datatype, dest, tag, comm, request)
MPI_Irecv  (buf, count, datatype, source, tag, comm, request)

All the argument except last one is the same with MPI_Send and MPI_Recv. The last one is predefine at the beginning of the code.

MPI_Request request;

MPI supplied many collective calls that enable a programmer to give commands to a subgroup of the processors in the virtual machine. Collective communication makes some tasks easier to be performed such as process synchronization, global summation, scattering and gathering data. Here we use MPI_Barrier calls, which perform the task of process synchronization. It blocks the caller until all group members have called it. The returns at the process only after all group members have entered the call. Its syntax is:

MPI_Barrier(MPI_COMM_WORLD);

In our main program we use another MPI function that check if there is a message from specific source. This is MPI_Iprobe function. Its syntax is:

MPI_Iprobe(source,tag, comm, & flag, & status);

The input parameters are source, tag and comm. The outputs ones are flag and status. Flag return 1 if the message is ready to be received or 0 if it is not. This function dose not actually receive a message, it only indicate that a message matching the signature specified is ready to be received.

The last function we use in our program is MPI_Wtime, which calculate the current time. We can calculate the start time of the processing and the last one. To get the total processing time, we can get the difference between them.

6.4 MPI implementation
There are many available implementation of MPI user can go on with them. These are *MPICH, CHIMP/MPI, and LAM*. Users can use any one of them. Here we deal with LAM implementation, which is available on our laboratory. LAM, denoted for local area multicopmuter, is an MPI programming environment and development system for a message – passing parallel machine. “With LAM a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one computer-intensive problem.” [3] LAM features a full implementation of the MPI communication standard. MPI need to be started before use it. To start it the user must creates a file listing the participating machine’s IP addresses in the communicator, [ emacs proc ]. Each machine will be given a node identifies (nodeid) starting with 0 for the first listed machine creation MPI started using lamboot. [lamboot proc].

**6.4.1 Compiling MPI programs:**

For C compiler mpicc must be used. It links the LAM library and sets up header and library search directory. The compiled program put into another file differ from the C file. For example if user wrote a program and save it as ABC.c, it must be compiled on another file [ e.g. ABCD]. The syntax will be:

```
mpicc -o ABCD ABC.c
```

ABCD is the executable file from where running will occur.

**6.4.2 Executing MPI Programs:**

An MPI application is started by one invocation of the mpirun command. The syntax:

```
mpirun -np 3 ABCD
```
Mpirun command run MPI program on LAM nodes additionally, mpirun will send the name of the directory where it was involved on the local node to each of the remote nodes an attempt to change to the directory. np <x> option means that run this program on the given x node.

6.4.3 Terminating LAM:

After finishing from running programs, LAM must be also being terminated. The (lamhalt) tool removes all trace of the LAM session on the network.

6.4 About our program

In this program we will build two processes that communicate among their function in different way. One process is producer and the other is consumer. Each of them need two buffers to accept data, private buffer (buffer[ ]) as well as temporary one (temp[ ]). No process can access the private buffer of the other one, but need to work/print from private buffer and copy it to from the temporary buffer (see fig 6.1). This is because of many things; one of them, it may be there is a difference on computing speed between two processes (one slower than the other), this will cause to stop the others job waiting the other one to complete its work. Also, the private buffer may be shared by many processes or threads within one machine, so it is better to empty the buffer for another uses.

![Diagram of producer and consumer processes](image-url)
Chapter 6

Implementation

Network line

Fig 6.1 Producer and consumer processes

The procedure begins the process by sending a message to the consumer notifies it by its coming. Then begin to produce specific number of random items from any location of the memory and deposits it to its private buffer if it is not full. The maximum number of produced item is predefined by giving specific size to the buffer. Here, a buffer size of 32 is to be chosen. We will use (%) operator to ensure that data buffer never overflow.

After production task, the producer checks the consumer if it is busy or ready to receive information. If it is, producer first passes the produced items to the temporary buffer, from which we send to the consumer. Then begin the sending to the consumer process by-first- send the number of items, which is denoted by a counter (int counter), followed by the data. Consumer begins by consuming every item in its private buffer before accept more data or request another data from producer. But if producer has terminated, it is stop the work and not proceed, otherwise, it accept data from producer into its temporary buffer and print it, computing the items printed. As consumer is busy consuming data and producer has completed, the later notifies the former one by its completion and terminate normally. Consumer process then prints the items within its private buffer.
Chapter 7

MPI program Result
Synchronization mechanism must provide continuously to the cooperating processes. Sending and receiving can achieve synchronization between two processes by sending and receiving messages in such way so as to avoid the blocking and a deadlock.

The main program in this thesis solves the producer – consumer problem using message-passing. This program with its existence result; synchronization is achieved; since there is a result and no blocking or deadlock is occurred. There are three steps to get the result.

**Step one:**
Create a text file (host file) that contains the producer and consumer IP using emacs editor. The first process is given rank 0, and the second one is given rank 1, this is the responsibility of MPI library. Notice that the consumer and producer ranks are predefined within the code.

![Fig 7.1 proc file](image)

**Step two:**
Starting LAM to create the network topology using lamboot instruction. The topology will be constructed contain two processes having two different IP addresses. The third show the us the version of LAM we used.

![Fig 7.2 Lamboot result](image)

**Fig 7.2 Lamboot result**

**Step Three:**

1. Insure that the program errorfree by compilation. If not, some errors will be appear on the screen.
2. Run the main program

The figure below shows the running of the program
Chapter 7

MPI Program Result

This program defines two processes, PRODUCER and CONSUMER.

Hello CONSUMER I am the PRODUCER, my rank is 0.

The items produced are:

Hello CONSUMER I am the PRODUCER, my rank is 1.

I am the consumer, I finish my task. The items will be consumed are:

The number of items consumed is 32.

Total execution time in second = 0.002793
Step Four:
This one is for LAM so as to terminates it.
Fig 7.4 LAM Halt result
Chapter 8

Conclusion
Synchronization of cooperating-processes is being an important issue in operating systems, that avoid the blocking occurrence on computer systems. It can take place in centralized systems as well as distributed one but with somewhat different scheme. This difference is due to existence and non-existence of common shared memory and common clock in centralized systems and distributed environments, respectively.

In distributed systems, which the processors are loosely coupled, we need some communication model that provide machine-to-machine communication capability. This is carried out by passing message through the network.

Race condition can occurs when two parallel processes are modifying or execute the same variables. The general solution is to prevent these processes from modifying at the same moment. Doing this is called mutual exclusion. It makes execution a critical section and atomic action. This can be done by a hardware solution which is very fast and dose not involve busy waiting, but is not efficient enough because it works only with single processor. Also, we can use a software solution that no need for special hardware and work with multipleprocessors, but requires busy waiting. From these two solutions, it is better to prevent race condition with one need no busy waiting and base on hardware, uninterruptible instructions.

In distributed systems environment, mutual execution can be implemented in different ways. In centralized approach, one of the processes in the system is chosen to coordinate the entry to the critical section. In fully distributed approach, the decision making is distributed across the entire system. A ring-structure network uses the token-passing approach. The fail of one process is possible, so one of the election algorithms must be chosen to create another coordinator. Two algorithms can be used to elect a new coordinator in case of failure, bully algorithm and ring algorithm.
A deadlock state occurs simply, when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. Deadlock in traditional systems is more easy to deal with, the important thing is to realized that deadlock is possible and then to have some methods to prevent, avoid, and recover this situation. The primary method for dealing with deadlock in distributed system is deadlock detection. The main problem is deciding how to maintain the wait-for graph (WFG).

Inorder to implement the synchronization in distributed system over specific problem, you must first chose a standard interface that has the ability to pass the messages through the network. MPI on these standards. One must be careful about his send/receive functions in the program, that the number of sending function is the same receiving ones, and all must be in specific order. This must be done perfectly or the program will be blocked. A nonblocking send/receive operation. But a blocking one ensure that the message is received by a meant receiver, so it is more reliable.
Appendixes
Appendix A

Main program

#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"
#define PRODUCER  0
#define CONSUMER 1
#define REQUEST  49
#define SIZE  157
#define DATA  17
#define TERMINATE 250
#define produce(x) x=rand()
#define consume(x)
#define BUFFER_SIZE 32

main (int argc, char *argv[])
{
  char  buffer[BUFFER_SIZE], temp[BUFFER_SIZE],c;
  int  nextc, rank, i, idle, consumed,nextp;
  int  counter, start, in, out, stop;
  double  start_time, end_time, proc_time;
  //variable require by MPI function

  int flag;
  MPI_Status status;
  MPI_Request request;
  /* MPI main function */
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &processes);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank)
  printf("This program define two processes, PRODUCER and  CONCUMER.");
  printf(" 
");
  /*begin initilization */
start = in = out = next = counter = consumed = 0;
/* empty tow buffer(private & temporary) */
for (i = 0; i < BUFFER_SIZE; i++)
buffer[i] = temp[i] = 0;
/* end initialization */
/*block the caller until every one arrive */
MPI_Barrier(MPI_COMM_WORLD);
/* calculate the initial time*/
start_time = MPI_Wtime();

/*now begin the work .go to processes if it is the producer 0 then begin to produce an item and pass it to its private buffer */
switch (rank)
{
  case PRODUCER:
    printf("hello CONSUMER i'am the PRODUCER my rank is: %d\n",rank);
    /*send signal to consumer so as to be ready included my rank*/
    MPI_Isend(&rank,1,MPI_CHAR,CONSUMER,REQUEST,MPI_COMM_WORLD);
    while (counter < BUFFER_SIZE)
    {
      /* produce random items & put them into buffer*/
      produce (nextp);
      buffer[in] = nextp;
      /*increase the pointer to next location from the buffer and ensure that it is never overflow*/
      in = (in + 1) % BUFFER_SIZE;
      counter++;
    }
    printf("the items produced are :\n");
    for (i = 0; i < BUFFER_SIZE; i++)
Appendixes

printf ("%c ", buffer[i]);
printf ("the number of produced items is: %d\n", counter);
printf("///////////////////////////////////////// \n");
/*after process 0 produce an item see if there is any requested message from the consumer process if not wait*/
idle = 0;
do
{
    MPI_Iprobe(1,REQUEST, MPI_COMM_WORLD,&flag,&status);
    if(flag)
    {
        MPI_Recv(&out,1,MPI_CHAR,1,REQUEST,MPI_COMM_WORLD,&status);
        idle=1;
    }
} while( !idle);
/*send data to a waiting consumer */

if (idle)
{
    for (i = 0; i < counter; i++)
    {
        /*pass the produced items to the temporary buffer*/
        temp[i] = buffer[i % BUFFER_SIZE];
    }
    MPI_Send (&counter, 1, MPI_CHAR, 1, SIZE, MPI_COMM_WORLD);
    MPI_Send (temp, counter,MPI_CHAR,1,DATA,MPI_COMM_WORLD);
    counter = 0;
}
break;
default:
/* check if there any message from the producer */
MPI_Iprobe(0,REQUEST,MPI_COMM_WORLD,&flag,&status);
if (flag)
{
    MPI_Recv(&out,1,MPI_CHAR,0,REQUEST,MPI_COMM_WORLD, &status);

    stop=0;
}
if(!stop)
    printf ("Hello PRODUCER i'am the CONSUMER my rank is: %d\n", rank);
    /*empty the private buffer befor accept more */
    while (counter > 0)
    {
        nextc = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        consume (nextc);
    }
    /*request data from producer and wait reply */
    MPI_Isend(&rank,1,MPI_CHAR,PRODUCER,REQUEST,MPI_COMM_WORLD);
    MPI_Recv (&counter, 1, MPI_CHAR, PRODUCER, MPI_ANY_TAG, MPI_COMM_WORLD,&status);

    /*if producer has terminated then stop the work other wise, accept data from producer & update local buffer */
    if (!counter)   stop = 1;
else
{
    MPI_Recv (temp, counter, MPI_CHAR, PRODUCER, DATA, MPI_COMM_WORLD,&status);
    for (i = 0; i < counter; i++)
    {
    }
buffer[i % BUFFER_SIZE] = temp[i];
consumed++;
}
}
break;
}
/*producer notifies consumer that it has completed */
if (rank == PRODUCER)
{
    stop=1;
    MPI_Send (&stop, 1,MPI_INT,1,TERMINATE,MPI_COMM_WORLD);
}
/*wait here until every one arrive */
MPI_Barrier (MPI_COMM_WORLD);

/*calculate the end time*/
end_time = MPI_Wtime ();
/*calculate the total process time*/
proc_time = end_time - start_time;
if (rank)
{
    printf("i'am the consumer. I finish my task. The items will be consumed are:\n");
    for(i=0;i<BUFFER_SIZE;i++)
    printf("%c ",buffer[i]);
    printf ("The number of items consumed is = %d \n", consumed);
}
printf(" ////////////////////////////////////////////  
" Total execution time in second = % IF\n",proc_time);
MPI_Finalize ();
return 0;
Appendix B

Definition of Distributed System

In mid of 1980s, tow advances in technology began. These are development of powerful of microprocessor and invention of high-speed local area network (LAN). The later one, which is called, distributed system, allowed dozen or even hundreds of machine to be connected in such way so as to give specific design issue for the user. These tow technologies are feasible and easy to construct. But distributed system required different software than do centralized systems. Distributed systems such as client-server applications and cluster-based parallel computation are an important part of modern computing. Distributed computing allows the balancing of processing load, increases program modularity, isolates functionality, and can provide an element of fault tolerance.

Distributed computer systems contain software program and data resources dispersed across independent and different computers connected through a communication network. To operate as a system, there should be certain system-wide standards for interoperability. This set of standards composes a Network Operating System.

They made for computer system that have more then one CPU and are loosely coupled (i.e. each processor has it's own local resources and processors communicate to each other through message passing). These collections of autonomous computers linked by a network using software to produce an integrated computing facility.

If the number of the hosts not exceed than 10’s hosts, this is a Local Area Network (LAN). Metropolitan Area Network (MAN) has a network of 100’s host and more than this, 1000’s or 1,000,000 is Wide Area Network (WAN). The last one which we are called Internet.
A distributed system allows users of individual, networked computers to share data and processing power, often over long distances. Distribution can also enhance availability, reliability and performance.

Most people prefer this type of systems over centralized one due to many reasons. The important one is economics and speed. Microprocessors offer a better price/performance and may have more total computational speed than a mainframe. Also, some application involves spatially separated machines, this which is called inherent distribution. In addition to, reliability and increment growth. If one machine crashes, the system as whole can still work and the computing power can be added in small increment.

Distributed systems also have many advantages over a personal computer PC. For one thing, many users need to share data and device. For example, to access a common database and to share expensive peripherals like color printer.

Although these systems have their strength, they also have their weakness and disadvantages. Little software exists at present for distributed systems and the network can saturate or cause other problem. Also, there is a problem in security that is easy to access and applies to secret data.

On a distributed system we need mechanisms to perform network communication so as be able to write to the network and process incoming messages from the network, also determine location of channel buffers as a first pass that channel names will include machine id and channel id for that machine. We need too, mechanisms for service all local requests as with uniprocessor, service local requests for remote channels and service incoming requests for local channels. Deal with blocking on channel creation and remote communication is also needed.

**Key characteristics of distributed systems**

“Distributed systems have many important characteristic optimized on resource sharing, openness, concurrency, scalability, fault tolerance and transparency. Below, more detail about this.” [2]
Resource Sharing
We main by resource in hardware concept are disks and printers….etc. In the software are files, windows, and data objects.
Hardware sharing needed for convenience and reduction of cost. And data sharing for consistency (compilers and libraries), exchange of information (database) and cooperative work (groupware.)

Openness
Distributed systems are either opened or closed with respect to hardware or software. In an open system, it is easy to add new hardware drivers. Applications were hardware independent and IPC allowed extension of services and resources. They published specifications and standardization of interfaces.

Concurrency
We mean by concurrency, multi-programming and multi-processing. Distributed system allow parallel executions that is many users using the same resources, application interactions and many servers responding to client requests.

Scalability
Scalability handles growth of machine in a network. Small system with two computers and a file server on a single network can be extended to large one or current Internet. The software not changes to support growth. To support scalability we must avoid centralization, choose the naming or numbering scheme carefully and handle timing problems with caching and data replication.

Fault Tolerance
For one instant computer may fail, therefore we need hardware redundancy and software recovery.
Transparency

How to achieve the single system image in distributed system and everyone not think as multiple machines but as a single processor timesharing system is called transparency.

Two levels of transparency: to users, to processes. Users are easier to fool than processes. Process transparency means that it must be easy to write programs that don't know where they execute or where other things live. Transparency must be of access, location, concurrency, replication, failure, migration and performance.

To achieve all these entire characteristics, designers must put some goals in front of their eyes when designing distributed systems. These goals are: high performance, reliability, scalability, consistency and security.

From another point of view, networks are usually characterized in terms of two things latency: the minimum time to get the minimum amount of information between two sites and Bandwidth: once information is flowing, how many bits per second can be transmitted (i.e., the marginal cost per bit).
Appendix C

Glossary

In this glossary a brief description of the most important and terminology are given.

**Busy Waiting:**
A process or processor is busy waiting if it is waiting for an event to occur by inquiring, over and over again, if the event has occurred.

**Communicator**
A collection of processes that can send message to each other.

**Condition variable:**
Is a variable used in a monitor to represent a condition or event.

**Cooperating process**
Is one that can affect or be effected by the other processes executing in the system.

**Critical section:**
Is a section of code in a program that accesses a set of shared variables.

**Deadlock:**
Is a situation where a group of processes are all blocked and none of them can become unblocked.

**Directory:**
A directory is a data structure in a file system that maps names into objects. The objects mapped are either files of other directories.

**Distributed operating system**
Is an operating system that run on a network of computers but give the users the illusion that they are running on a single large system with one operating system.
**Distributed systems**

Collection of processes that do not share memory or clock.

**FIFO:**
FIFO stands for first – in, first – out. In a FIFO queue, the item that has been in the queue the longest removed next. The term FIFO is also used for a pipe that is named in the file naming system.

**File:**
A file is a collection of related in format on that is stored on a disk and that has a unique name.

**IPC:**
Stands for inter-process communication. Some comment forms of IPC are: files, pipes and message.

**Message:**
Is a content of communication between two processes. Message passing is a common form at interprocess communication.

**Monitor:**
Is a mechanism for process synchronization. It consist of a collection of producers the monitor enforces mutual exclusion between them, that is monitor only allow one process at a time to be executing in any of the producers in the monitor.

**Multi computer:**
Is a computer system with two or more computers each with its own memory and with a communications network between the computers.

**Mutual exclusion:**
Is a mechanism for avoiding race conditions by preventing two processes from running in their critical section.

**Pipe:**
A pipe is a method of inter-process communication. It is accessed like a file. One process writes to the pipe and when another process reads from the pipe and it will read the same bytes in the same sequence.
Appendixes

Process

Unit of work in system.

Process table:
Is a data structure in an operating system that contain the process descriptor of each process in the system.

Program:
Is a set of instruction that can be used to control process.

Race Condition:
Is a situation where two processes either communicate or share memory while they are running and the output of one or both of the processes depend on the relative speed of the two processes.

Semaphore:
Is a synchronization primitive with two operations: wait and signal, an implement with two data structures: a counter and a queue at blocked processes.

Starvation:
Is a resource management problem where a process dose not get the resource, it need for a long time because the resources are being allocated to other processes.

Synchronization:
Is the activity whereby two processes coordinate their activities so that they can work together on a task.

System call:
Is a request by a process for the operating system to perform an operation for the process.

WWW

World Wide Web
References
References

Books
1. Andrew s. Tanenbaum, “Modern Operating System”.

Paper
3. “A User’s guid to MPI”
   Peter S.pacheco, Department of Mathematics, University of San Fransisco
   San Fransisco, CA 94117

www pages
5. http://nt.server-0.npu.edu/cs505b/lect3/printversion.html