International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 6, August - 2012

Concurrent Programming and Parallel distributed O.S

Mr. Talari Asish kumar
CMR TECHINAL CAMPUS, kandlakoya, Hyderabad-501401.

Abstract

This paper consists of two topics, one is Concurrent Programming & Parallel distributed O.S . In a concurrent program, several streams of operations may execute concurrently, each stream of operations executes as it would in a sequential program. While coming to parallel distributed O.S, A distributed operating system is the logical aggregation of operating system software over a collection of independent, networked, communicating, and physically separate computational nodes.

1. Introduction

The introduction of multi-core processors has renewed interest in parallel functional programming and there are now several interesting projects that explore the advantages of a functional language for writing parallel code or implicitly paralellizing code written in a pure functional language. These lecture notes present a variety of techniques for writing concurrent parallel programs which include existing techniques based on semi-implicit parallelism and explicit thread-based parallelism as well as more recent developments in the areas of software transactional memory and nested data parallelism.

In traditional sequential programming, the object-oriented model has gained wide acceptance. Concurrent and distributed programming remains, however, one of the last areas of software engineering where no single direction has been generally recognised as the preferred approach. In general, writing concurrent programs is extremely difficult because the multiplicity of possible inter leavings of operations among threads means that program execution is non-deterministic. For this reason, program bugs may be difficult to reproduce. Furthermore, the complexity introduced by multiple threads and their potential interactions makes programs much more difficult to analyze and reason about. Fortunately, many concurrent programs including most GUI applications follow stylized design patterns that control the underlying complexity.

We also use the terms parallel and concurrent with quite specific meanings. For a parallel program we have the expectation of some genuinely simultaneous execution. Concurrency is a software structuring technique that allows us to model computations as hypothetical independent activities (e.g. with their own program counters) that can communicate and synchronize.

The connection between programming languages and operating systems is especially close in the area of concurrent programming. First, threads are sometimes supported by the underlying operating system, so the language implementation needs to make use of those facilities, and the language designer may choose to present or to omit features, depending on the operating system and what it can do. For example, a thread can be modeled by a Unix process. Generally, Unix processes cannot share memory. However, some versions of Unix, such as Solaris, offer threads within a single address space; these threads do share memory. Second, operating systems themselves are often multithreaded.

2. Concurrent Programming

Architectural advances of recent years, coupled with the growing availability of networked computers, have led to a new style of computing, called concurrent programming, that allows multiple computations to occur simultaneously in cooperation with each other. Many people distinguish two classes of concurrent programming: Distributed programming refers to computations that do not share a common memory, and parallel programming refers to computations that share a common memory. This distinction is not always helpful, since it is possible to implement a distributed computation on a shared-memory computer, and to implement a parallel computation on a distributed-memory computer. It is up to the compiler and operating system to implement on the underlying architecture whatever concurrency style the programming language promotes. Terminology is less standard in the area of concurrent programming than elsewhere, so I will be somewhat arbitrary, but consistent, in my nomenclature. A parallel program is one which is written for performance reasons to exploit the potential of a real parallel computing resource like a multi-core processor. Concurrent programming has become a required component of ever more types of application, including some that were traditionally thought of as sequential in nature. It offers a comprehensive approach to building high-quality concurrent and distributed systems. The idea is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract , which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension indeed consists of just one keyword `separate`; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. Writing applications with this model is extremely simple, since programmers do not need to deal with low-level concepts typically used in concurrent programming.

2.1 Parallel Languages

Some parallel languages, like SISAL and PCN have found little favor with application programmers. This is because users are not willing to learn a completely new language for parallel programming. They really would prefer to use their traditional high-level languages (like C and Fortran) and try to recycle their already available sequential software. For these programmers, the extensions to existing languages or run-time libraries are a viable alternative.

2.2 Reasons for writing concurrent & parallel programs

Writing concurrent and parallel programs is more challenging than the already difficult problem of writing sequential programs. However, there are some compelling reasons for writing concurrent and parallel programs:

Performance: We need to write parallel programs to achieve improving performance from each new generation of multi-core processors.

Hiding latency: Even on single-core processors we can exploit concurrent pro- grams to hide the latency of slow I/O operations to disks and network de- vices.

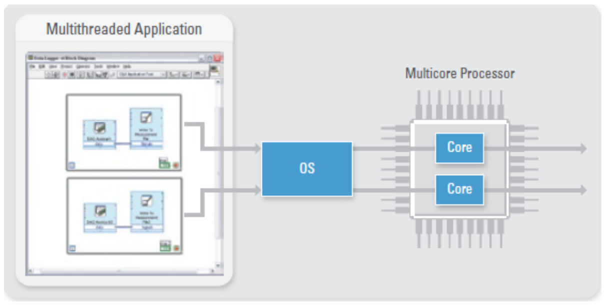
Software structuring: Certain kinds of problems can be conveniently represented as multiple communicating threads which help to structure code in a more modular manner e.g. by modeling user interface components as separate threads.

Real world concurrency: In distributed and real-time systems we have to model and react to events in the real world e.g. handling multiple server requests in parallel.

All new mainstream microprocessors have two or more cores and relatively soon we can expect to see tens or hundreds of cores. We cannot expect the performance of each individual core to improve much further. The only way to achieve increasing performance from each new generation of chips is by dividing the work of a program across multiple processing cores. One way to divide an application over multiple processing cores is to somehow automatically parallelize the sequential code and this is an active area of research. Another approach is for the user to write a semi-explicit or explicitly parallel program which is then scheduled onto multiple cores by the operating systems and this is the approach we describe in these lectures.

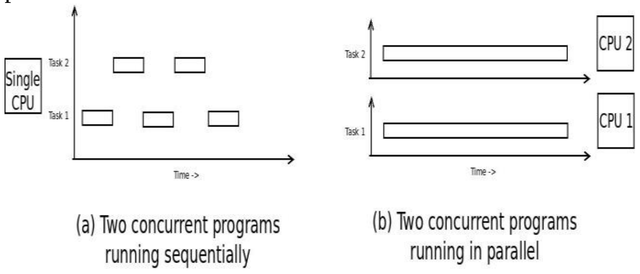
Parallel processing is much faster than sequential processing when it comes to doing repetitive calculations on vast amounts of data. This is because a parallel processor is capable of multithreading on a large scale, and can therefore simultaneously process several streams of data. This makes parallel processors suitable for graphics cards since the calculations required for generating the millions of pixels per second are all repetitive. GPUs can have over 200 cores to help them in this. The CPU of a normal computer is a sequential processor - it's good in processing data one step at a time. This is needed in cases where the calculation the processor is performing depends on the result of the previous calculation and so on; in parallel processing these kinds of calculations will slow it down, which is why CPUs are generally optimised for sequential operations and have only 1-8 cores.

Consider a traditional sequential language. In many cases, you must explicitly break your code into separate pieces called threads that can run on multicore processors. (The OS handles these threads after they are created.) Although the concept of threads is straightforward, working with them can be time-consuming and tedious. Each thread must be carefully managed – plus, data accessed by threads is very susceptible to race conditions if not protected carefully.



parallel sections of code and maps them into threads, which can take advantage of a multicore processor.

Let us consider both single processor and multi-processor machines (like the Intel dual-core, or high end processors that have tens of processors). The figure below illustrates two concurrent tasks (such as make and gcc) that can be scheduled sequentially or in parallel.


Single CPU
Task 2
Task 1
Time ->
(a) Two concurrent programs running sequentially
Task 2
Task 1
CPU 2
CPU 1
Time ->
(b) Two concurrent programs running in parallel

2.3 Advantages of using concurrent programming

It has many advantages such as:

- It lets the programmer decide which part of the code he wants to run concurrently.
- It creates and manages the threads for the programmer.
- Its compiler are available from a lager number of companies such as Intel and IBM to name a few.
- "It provides a set of pragmas, runtime routines, and environment variables that programmers can use to specify shared-memory parallelism in Fortran, C, and C++ programs." (Copty)
- OpenMP can run the code as a serial code.
- OpenMP is easier to program than other parallel programming languages such as MPI(Message Passing Interface).

- *Reactive programming:* User can interact with applications while tasks are running, e.g., stopping the transfer of a big file in a web browser.
- *Availability of services:* Long-running tasks need not delay short-running ones, e.g., a web server can serve an entry page while at the same time processing a complex query.
- *Parallelism:*Complex programs can make better use of multiple resources in new multi-core processor architectures, SMPs, LANs or WANs,e.g.…scientific/engineering applications, simulations, games, etc.
- *Controllability*:Tasks requiring certain preconditions can suspend and wait until the preconditions hold, then resume execution transparently.

➢ *Advantages for developers:*

There are some cautions that we should be aware of:

- Considering Overheads: Parallel execution doesn't come for free. There are overhead costs associated with setting up and managing parallel programming features. If you have only a small amount of work to perform, the overhead can outweigh the performance benefit.
- Coordinating Data: If your pieces of work share common data or need to work in a concerted manner, you will need to provide coordination. As a general rule, the more coordination leads, the poorer the performance of your parallel program.
- Scaling Applications: Adding a second core or CPU might increase the performance of your parallel program, but it is unlikely to double it. Likewise, a four-core machine is not going to execute your parallel program four times as quickly. You can expect a significant improvement in performance, but it won't be 100 percent per additional core, and there will almost certainly be a point at which adding additional cores or CPUs doesn't improve the performance at all.

3. Parallel distributed O.S

Distributed operating systems have many aspects in common with centralized ones, but they also differ in certain ways. This paper is intended as an introduction to distributed operating systems, and especially to *current university research about them. After a discussion of what constitutes a distributed operating system and how it is distinguished from a computer network, various key design issues are discussed. DISTRIBUTED OPERATING SYSTEMS will provide engineers, educators, and researchers with an in-depth understanding of the full range of distributed operating systems components. Researchers are exploring the world of distributed computing, in which users in various locations can work with the same set of geographically dispersed resources. These efforts have led to such high-profile technologies as peer-to-peer (P2P), pervasive, and nomadic computing. A critical part of this research is developing consistent approaches to distributed-computing operating environments, which must work consistently across many platforms and technologies. Major efforts in this area include Globe, Opus, and Project Oxygen. None of the three represents radically new technologies, but instead each applies existing technologies in a novel way.*

A **distributed system** is a collection of independent processors that do not share memory or a clock and appears to the users of a system as a single computer. Such a system must: (a) be able to support an arbitrary number of processes, the distribution of which should be transparent to the user; (b) provide an efficient communication facility; and (c) be integrated into a single virtual computer. The focus of this book is an analysis of concepts and practice in distributed computing. The intended audience is anyone who is interested in the design and implementation of modern computer systems, particularly the operating systems and distributed algorithms that are essential in supporting networking and distributed processing. A distributed system is a collection of processors that do not share memory or a clock. Each processor has its own local memory. must be in English. These guidelines include complete descriptions of the fonts, spacing, and related information for producing your proceedings manuscripts

3.1 Examples of Distributed Systems

Example 1:

The Internet: net of nets (Fig. 1.1)
– global access to "everybody" (data, service, other actor; open ended)
– enormous size (open ended) – no single authority
– communication types
• interrogation, announcement, stream
• data, audio, video

Figure 1.1 A typical portion of the Internet

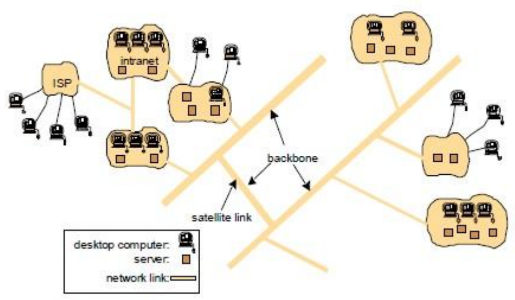
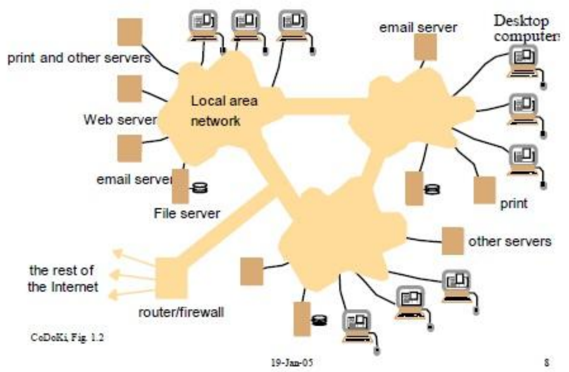


Fig 1.1

Example 2:

Intranets (Fig. 1.2)
– a single authority
– protected access
• a firewall
• total isolation
– may be worldwide
– typical services:
• infrastructure services: file service, name service
• application services

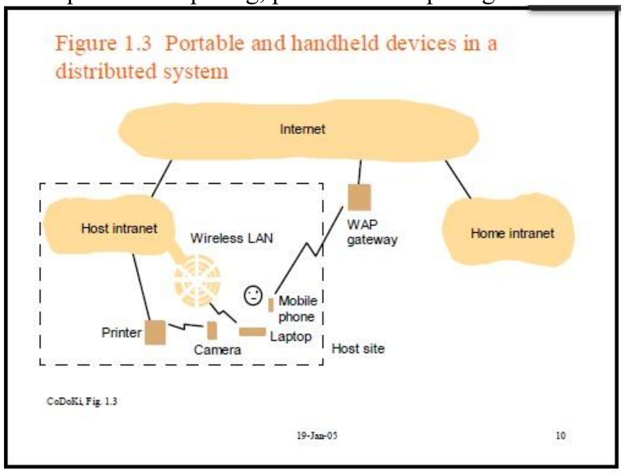
Figure 1.2 A typical intranet



Example 3:

Mobile and ubiquitous computing (Fig 1.3)
• Portable devices
– laptops
– handheld devices
– wearable devices
– devices embedded in appliances
• Mobile computing
• Location-aware computing

• Ubiquitous computing, pervasive computing



Operating system - a program that acts as an intermediary between a user of a computer and the computer hardware.

Operating system goals:

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.
- To hide details of hardware by creating abstraction.
- To allocate resources to processes (Manage resources).
- Provide a pleasant and effective user interface.

operating-System Services:

Program execution - system capability to load a program into memory and to run it.

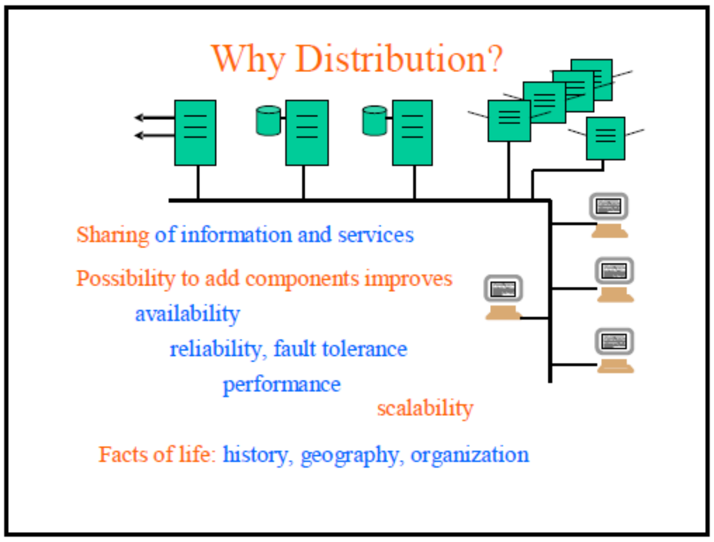
I/O operations - since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

File-system manipulation - program capability to read, write, create, and delete files.

Communications - exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via shared memory or message passing.

Error detection - ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs. Operating System Concepts, Addison-Wesley Ó 1994 3.10 Silberschatz & Galvin Ó 1994 Additional operating-system functions exist not for helping the user, but rather for ensuring efficient system operation.

Resource allocation - allocating resources to multiple users or multiple jobs running at the same time. g *Accounting* - keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics. g *Protection* - ensuring that all access to system resources is controlled. Operating.



Why Distribution?

Sharing of information and services

Possibility to add components improves

availability

reliability, fault tolerance

performance

scalability

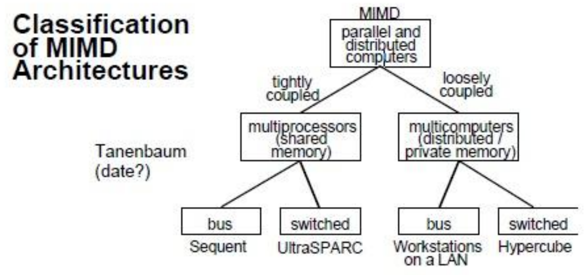
Facts of life: history, geography, organization

3.2 Classification of the OS

Multiprocessor Operating System

- Tightly-coupled software (single OS) running on tightly-coupled hardware.
- A process can run on any processor F Single ready queue!
  All memory is shared
- File system similar to that on non-distributed systems n Network Operating System
  Loosely-coupled hardware
  Loosely-coupled software
- Each computer runs its own OS
  User knows which machine he/she is on
  Goal: share resources, provide global (network) file system
  Typical utility programs: rlogin, rcp, telnet, ftp
- "True" Distributed Operating System
  Loosely-coupled hardware
  No shared memory, but provides the "feel" of a single memory
- Tightly-coupled software
  One single OS, or at least the feel of one
  Machines are somewhat, but not completely, autonomous

- distributed OS (like any OS) makes the use of many machines more easy and efficient


Classification of MIMD Architectures
Tanenbaum (date?)
MIMD
parallel and distributed computers
tightly coupled
loosely coupled
multiprocessors (shared memory)
multicomputers (distributed / private memory)
bus
switched
bus
switched
Sequent
UltraSPARC
Workstations on a LAN
Hypercube

3.3 KEY FEATURES AND ADVANTAGES OF A DISTRIBUTED SYSTEM

The following are the key features of a distributed system:

- They are Loosely coupled
- remote access is many times slower than local access
- Nodes are autonomous
- workstation resources are managed locally.
- Network connections using system software remote access requires explicit message passing between nodes messages are CPU to CPU.
- protocols for reliability, flow control, failure detection, etc., implemented in software.
- the only way two nodes can communicate is by sending and receiving network messages this differs from a hardware approach in which hardware signalling can be used for flow control or failure detection.

Advantages of Distributed Systems over Centralised Systems

- Better price/performance than mainframes• More computing powero sum of the computing power of the processors in the distributed system may be greater than any single processor available (parallel processing)
- Some applications are inherently distributed• Improved reliability because system can survive crash of one processor
- Incremental growth can be achieved by adding one processor at a time
- Shared ownership facilitated.

Advantages of Distributed Systems over Isolated PCs

- Shared utilisation of resources.
- Communication
- Better performance and flexibility than isolated personal computers
- Simpler maintenance if compared with individual PC's.

3.4 Disadvantages of Distributed Systems

Although we have seen several advantages of distributed systems, there are certain disadvantages also which are listed below

- Network performance parameters.
- Latency: Delay that occurs after a send operation is executed before data starts to arrive at the destination computer.
- Data Transfer Rate: Speed at which data can be transferred between two computers once transmission has begun.
- Total network bandwidth: Total volume of traffic that can be transferred across the network in a give time
- Dependency on reliability of the underlying network.
- Higher security risk due to more possible access points for intruders and possible communication with insecure systems
- Software complexity.

4. Conclusion

A distributed operating system takes the abstraction to a higher level, and allows hides from the application where things are. The application can use things on any of many computers just as if it were one big computer. A distributed operating system will also provide for some sort of security across these multiple computers, as well as control the network communication paths between them. A distributed operating system can be created by merging these functions into the traditional operating system, or as another abstraction layer on top of the traditional operating system and network operating system.Any operating system, including distributed operating systems, provides a number of services. First, they control what application gets to use the CPU and handle switching control between multiple applications. They also manage use of RAM and disk storage. Controlling who has access to which resources of the computer (or computers) is another issue that the operating system handles. In the case of distributed systems, all of these items need to be coordinated for multiple machines.

As systems grow larger handling them can be complicated by the fact that not one person controls all of the machines so the security policies on one machine may not be the same as on another.Some problems can be broken down into very tiny pieces of work that can be done in parallel. Other problems are such that you need the results of step one to do step two and the results of step two to do step three and so on. These problems cannot be broken down into as small of work units. Those things that can be broken down into very small chunks of work are called fine-grained and those that require larger chunks are called coarse-grain. When distributing the work to be done on many CPUs there is a balancing act to be followed. You don't want the chunk of work to be done to be so small that it takes too long to send the work to another CPU because then it is quicker to just have a single CPU do the work, You also don't want the chunk of work to be done to be too big of a chunk because then you can't spread it out over enough machines to make the thing run quickly.

5. References

1) Taubenfeld, Gadi (2006). Synchronization Algorithms and Concurrent Programming. Pearson / Prentice Hall. p. 433. ISBN 0-13-197259-6.
2) Filman, Robert E. Daniel P. Friedman (1984). Coordinated Computing: Tools and Techniques for Distributed Software. New York: McGraw-Hill. p. 370. ISBN 0-07-022439-0.
3) Practical Concurrent Programming for Parallel Machines D. B. Skillicorn.
4) Concurrent Programming and Robotics Ingemar J. Cox AT&T, Bell Laboratories Murray Hill, New Jersey 07974
5) Concurrent Programming: Principles and Practice, Greg Andrews
6) Tanenbaum, van Steen: Distributed Systems, Principles and Paradigms; Prentice Hall 2002
7) Coulouris, Dollimore, Kindberg: Distributed Systems, Concepts and Design; Addison-Wesley 2001
8) Andrew S. Tanenbaum and Albert S.Woodhull, Operating Systems Design and Implementation, 2/e, Prentice Hall, New Delhi.
9) Distributed Operating Systems ANDREW S. TANENBAUM and ROBBERT VAN RENESSE Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
10) OPERATING SYSTEM CONCEPTS Silberschatz Department of Computer Sciences University of Texas at Austin
11) Roosta, Seyed. Parallel Processing and Parallel Algorithms, Springer-Verlag, 1999.

BIOGRAPHY :

Mr **Talari Asish kumar** is currently pursuing bachelors degree in computer science engineering from CMR TECHNICAL CAMPUS, Kandlakoya, Hyderabad. It is one of the five colleges in CMR GROUP OF INSTITUTIONS.Which is one of the premier educational institutions dedicated to impart quality education and promoting excellence in academic pursuits in the field of Science.

Acknowledgement:

I have taken efforts in this project. However, it would not have been possible without the kind support and help of individuals and organization. I would like to extend my sincere thanks to all of them. I am highly indebted to the faculty of cse department of CMR Technical Campus for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project. I would like to express my gratitude towards my parents & members of CMR Technical Campus for their kind co-operation and encouragement which help me in completion of this project. I would like to express my special gratitude and thanks to industry persons for giving me such attention and time. My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.