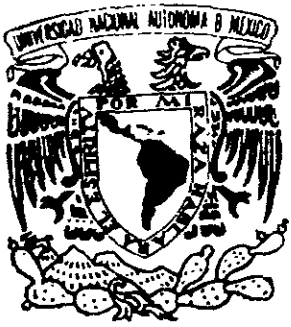

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

ACATLAN

SIMULACION DE UN SISTEMA MULTIPROCESADOR PARALELO EN UN SISTEMA MULTITAREA DE TIEMPO COMPARTIDO

T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A

MANUEL RAMIREZ ROJAS

ASESOR DE TESIS: M. EN S. I. ALMA LOPEZ BLANCO

NAUCALPAN DE JUAREZ, ESTADO DE MEXICO 1998

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

SIMULACIÓN DE UN SISTEMA MULTIPROCESADOR PARALELO EN UN SISTEMA MULTITAREA DE TIEMPO COMPARTIDO

Manuel Ramírez Rojas
Matemáticas Aplicadas y Computación
Escuela Nacional de Estudios Profesionales Acatlán
Universidad Nacional Autónoma de México

CONTENIDO

PRESENTACIÓN

Tema:

Sistemas multiprocesador para aplicaciones paralelas.

Título:

Simulación de un sistema multiprocesador paralelo en un sistema multitarea de tiempo compartido.

Objetivo:

Simular un sistema multiprocesador paralelo en el entorno del sistema operativo multitarea de tiempo compartido Unix, en el que sea posible ejecutar aplicaciones de programación en paralelo.

Hipótesis:

Con la simulación del entorno de un sistema multiprocesador en un sistema operativo multitarea de tiempo compartido es posible ejecutar aplicaciones de programación en paralelo.

Variables:

Independiente: Entorno del sistema operativo.

Dependiente: Aplicaciones de programación en paralelo.

INTRODUCCIÓN

Existe una gran demanda por sistemas de cómputo de alto rendimiento que tiende a crecer debido al amplio campo de aplicaciones que estos sistemas poseen en la actualidad, como la inteligencia artificial, la teoría fractal, las redes neuronales y otras aplicaciones científicas y técnicas. El éxito de este tipo de aplicaciones se relaciona en gran medida con la capacidad de respuesta de los sistemas de cómputo que se utilicen para su desarrollo, es por esto que no sólo deben contar con un desempeño de hardware adecuado, sino que también deben de poseer un diseño arquitectónico apropiado y adecuadas técnicas de procesamiento.

Como respuesta a esa demanda de velocidad de procesamiento, existe el *procesamiento paralelo*, en el que se aprovecha la ejecución concurrente de varias tareas para acelerar los tiempos de ejecución. Las computadoras que cuentan con capacidad de procesamiento paralelo, por lo tanto, son capaces de proporcionar un medio adecuado en el que se maximice el rendimiento de un sistema, y por consecuencia de las aplicaciones paralelas, mediante la realización concurrente de actividades. No obstante, este tipo de sistemas de cómputo para aplicaciones paralelas requieren de un sistema operativo que planifique y administre las actividades concurrentes y que además se encargue de la asignación eficiente y eficaz de los recursos con que cuente el sistema.

Por un lado, existen los llamados *sistemas de tiempo compartido*, en los que las tareas que se llevan a cabo son asignadas a un procesador único que es compartido por todas las actividades del sistema, a las que el sistema operativo asigna tiempos

variables de atención del procesador. Sin embargo, la mejor forma de ejecutar aplicaciones paralelas se obtiene en un *sistema multiprocesador* en el que, *en un mismo* intervalo de tiempo pueden suceder varias actividades en varios procesadores.

Hoy en día Unix es uno de los sistemas operativos que ha sido adoptado por la gran mayoría de las computadoras de alto rendimiento. En su diseño original, Unix fue concebido como un sistema operativo multitarea de tiempo compartido en el que se pueden ejecutar varias tareas a la vez en un mismo procesador. A pesar de ser un sistema operativo multitarea, el diseño original de Unix no proporcionaba un entorno *adecuado para la ejecución de aplicaciones paralelas,* debido a que solo explota a un procesador único que reparte entre las diferentes tareas que se lleven a cabo. Para lograr que el sistema operativo Unix sea capaz de proporcionar el entorno apropiado para la ejecución de aplicaciones paralelas en las actuales arquitecturas de sistemas con más de un procesador, ha sido necesaria una reestructuración de su diseño original.

El objetivo de este trabajo es estudiar y analizar el diseño original de Unix para obtener el modelo del sistema operativo que sea capaz de operar de manera óptima en un ambiente multiprocesador, en el que sea posible ejecutar aplicaciones de programación en paralelo, *para así aprovechar el gran potencial* que ofrecen los sistemas Unix y encauzarlo a la ejecución de este tipo de aplicaciones, que requieren de sistemas multiprocesadores para su adecuada realización.

El trabajo esta organizado en 4 capítulos. El capítulo 1 se titula "Sistemas de Cómputo de Alta Velocidad". Tomando como referencia el sistema operativo que emplean, los sistemas de cómputo han seguido un proceso evolutivo desde sus orígenes hasta la actualidad, pasando por distintas fases o *generaciones.* En este

capitulo se realiza un estudio de este proceso, iniciando con la definición del concepto de sistema operativo y sus principales características y funciones. Después de describe el proceso mediante el cual los sistemas de cómputo han evolucionado, diferenciando las características propias de cada una de las fases evolutivas, desde los sistemas de procesamiento por lotes hasta el actual desarrollo de sistemas distribuidos. Finalmente se desarrolla la última de las etapas de éste proceso evolutivo en la que se habla sobre *Transputers*, que son microprocesadores especializados empleados para el desarrollo de sistemas paralelos. Se analiza su arquitectura y su funcionamiento al operar con varios procesos concurrentes en varios procesadores.

El capítulo dos se titula "Los Sistema Paralelos". En este capítulo se estudia de forma específica aquellos sistemas que son capaces de ejecutar aplicaciones en paralelo, resaltando la importancia del concepto de multiprocesamiento asociado con el paralelismo. Primeramente se define el concepto de procesamiento paralelo y sus campos de aplicación en diversas áreas de la ciencia y la tecnología. Después se revisa la clasificación de los sistemas de cómputo de acuerdo a su multiplicidad de flujos, tanto de instrucciones como de datos, para distinguir aquellas que favorezcan el multiprocesamiento. Se describen las diferentes arquitecturas, organizaciones y estructuras de interconexión para sistemas multiprocesador, así como las formas en las que un sistema operativo puede administrar procesos concurrentes en varios procesadores.

El capítulo 3 se titula "Simulación de multitarea a multiproceso. Caso Unix System V". En este capítulo se analiza el problema del multiprocesamiento en ambiente Unix, ya que su diseño original se desarrolló para sistema uniprocesadores. Además se desarrollan las diferentes formas de organización del sistema operativo Unix en las que

ha sido factible operar en entornos de *multiprocesamiento* con varios procesadores activos a la vez, resaltando las ventajas y desventajas de cada una de ellas.

Finalmente, en el capítulo 4 titulado "Ejecución de aplicaciones paralelas", se ejemplifica el funcionamiento de los sistemas multiprocesador, resaltando su mayor capacidad de procesamiento y mayor velocidad de ejecución con respecto a sistemas de procesamiento secuencial. Para ello, se toma como referencia a uno de los casos de investigación más reciente en matemáticas: la *geometría fractal*. El capítulo inicia con la definición de los conceptos de fractal y de dimensión fractal, así como su vinculación con el mundo real. Después se realiza un estudio de las funciones cuadráticas complejas que son utilizadas para obtener, tanto al Conjunto Julia como el Conjunto de Mandelbrot, que son dos de los objetos más representativos de la geometría fractal. Se generan los algoritmos y programas necesarios para la representación de dichos fractales por computadora y se explica la forma en que este tipo de aplicaciones, con gran cantidad de procesos iterativos, suelen consumir gran cantidad de tiempo para ser ejecutadas. Por último, se realiza una evaluación del desempeño de varias organizaciones de cómputo, incluyendo un sistema multiprocesador, al ejecutar este tipo de aplicaciones. Es en esta evaluación en la que se observa la mayor capacidad de los sistemas multiprocesador para llevar a cabo ejecuciones más eficientes con respecto a los sistemas de procesamiento secuencial.

Al final se incluye un Apendice con el código fuente de los programas empleados en el capítulo 4, un glosario de terminología empleada y la bibliografía consultada.

CAPITULO 1
SISTEMAS DE CÓMPUTO DE ALTA VELOCIDAD.

1.1 Sistemas Operativos.

En la actualidad, los sistemas de cómputo han evolucionado de forma tal que su capacidad y velocidad aumentan a un ritmo muy acelerado, mientras que sus costos se ven reducidos. Debido a que los costos de los procesadores se han reducido paulatinamente, han nacido arquitecturas de sistemas con procesadores múltiples y en red que dan origen a una gran diversidad de oportunidades de desarrollo de nuevas formas de manipulación tanto el software como el hardware. Un ejemplo claro de estas nuevas tendencias tecnológicas puede verse en la forma en que los lenguajes de programación secuenciales están siendo desplazados por lenguajes de programación concurrentes, los cuales permiten especificar la ejecución de varias tareas de forma simultánea.

Por otro lado, cuando se tiene un equipo de cómputo, no sólo se tiene hardware, sino también los medios para satisfacer las necesidades de los usuarios del equipo mismo mediante software de aplicación, sin que éstos tengan que preocuparse sobre las operaciones internas de la computadora. Una parte muy importante del equipo de cómputo esta constituida por el programa intermediario entre el hardware y el usuario. Esta interfaz, denominada *sistema operativo,* es la responsable de hacer trabajar al equipo de cómputo en la forma más eficiente posible.

Así pues, de la misma forma en que las arquitecturas evolucionan, el desarrollo de sistemas operativos capaces de trabajar sobre dichos sistemas se convierte en una

necesidad fundamental para garantizar un correcto funcionamiento. El manejo de recursos con que cuenta un equipo de cómputo corre por cuenta de los sistemas operativos, por lo que éstos deben tener la capacidad suficiente para controlarlos de forma eficiente para el correcto desempeño de las nuevas arquitecturas de sistemas.

1.1.1 Definición de sistema operativo.

Inicialmente, un sistema operativo se podría definir como *el software que maneja el hardware*. Esto debido principalmente a que el concepto original de los sistemas operativos era el de elaborar un software con el objeto de manejar todos los componentes físicos de una computadora y que fuera capaz de controlar y administrar tales recursos. De esta forma, el sistema operativo se convertiría en la base sobre la cual los programadores y usuarios de un sistema de cómputo podían realizar sus programas y ejecutar sus aplicaciones sin la necesidad de verse involucrados con el modo de operación de los componentes de sus computadoras.

La esencia del concepto original es la misma, sin embargo la tendencia de los sistemas actuales es manejar al sistema operativo en forma de *microcódigo*, con lo que en algunos casos quedaría fuera de lugar concebirlo como software. De cualquier manera, un sistema operativo puede visualizarse como los programas, ya sea instalados en software o en *firmware*, que hacen utilizable el hardware. El hardware por su parte, proporciona la capacidad de cómputo del sistema y los sistemas operativos ponen dicha capacidad al alcance de los usuarios y administran el hardware para lograr el mejor rendimiento posible.

1.1.2 Funciones de un sistema operativo.

Los sistemas operativos son ante todo *administradores de recursos*. El principal recurso que administran es el hardware del sistema de cómputo:

- El procesador (o procesadores).
- Los dispositivos de almacenamiento.
- Los dispositivos de entrada/salida.
- Los dispositivos de comunicación.
- Los datos.

El propósito principal de un sistema operativo es el proporcionar un entorno adecuado a los usuarios del sistema en el que se puedan efectuar programas y aplicaciones. Para lograrlo, realiza varias funciones:

- Proporciona una interfaz a los usuarios para servir como intermediario entre el usuario del sistema y el hardware.
- Permite que los usuarios compartan entre sí el hardware y los datos.
- Evita que los usuarios interfieran recíprocamente.
- Planifica la distribución de los recursos.
- Facilita las entradas y salidas.
- Maneja la recuperación de errores.
- Contabiliza el uso de recursos.
- Maneja las comunicaciones.

En general, el objetivo principal que persigue un sistema operativo es lograr que el sistema se use de forma adecuada y que el hardware se emplee eficientemente

1.1.3 Componentes de un sistema operativo.

Aunque los sistemas operativos han evolucionado a la par de las arquitecturas de las computadoras, es posible distinguir sus componentes básicos. Estos componentes son principalmente programas de control, que incluyen a un programa

supervisor, un sistema de control de E/S, un controlador de comunicaciones de datos, el cargador inicial de programas y un programa de control de trabajos.

a) Programa *supervisor*. El supervisor dirige las actividades de E/S y maneja las condiciones de interrupción, la programación de trabajos, la recuperación de programas y la asignación en memoria principal.

b) Sistema de control de E/S. Este sistema maneja la programación de E/S, la corrección de errores de los dispositivos periféricos y otras funciones del manejo de datos.

c) Controlador de comunicaciones de datos. Este controlador esta compuesto por un conjunto de programas que se incluyen en los sistemas operativos que utilizan una red de canales de comunicación de datos y terminales remotas. Realizan actividades tales como entrada de datos, escrutinio automático, manejo de líneas de espera para los servicios y manejo de interrupciones para las terminales que compiten por los servicios, intercambio de mensajes, consultas y procesamiento de transacciones.

d) El cargador inicial de programas. Este cargador es un pequeño programa de control que carga al programa supervisor de control desde un dispositivo a la memoria principal cuando la computadora empieza a operar.

e) Programa de control de trabajos. La función de este programa es la de preparar al sistema de cómputo para el inicio del siguiente trabajo, ejecutando declaraciones del lenguaje de control de trabajos.

1.2 Proceso evolutivo de los sistemas de cómputo.

Los sistemas operativos han venido evolucionando a través de los años. Desde el punto de vista del sistema operativo que utilizan, los sistemas de cómputo han evolucionado a través de distintas fases o generaciones.

En sus inicios, las computadoras electrónicas carecían de sistema operativo, en estas máquinas usualmente se introducían los datos un bit a la vez desde tableros electrónicos y la programación se realizaba en lenguaje de máquina absoluto. Aún no se desarrollaba el lenguaje ensamblador. Posteriormente los programas se introducían en lenguaje de máquina mediante tarjetas perforadas y se desarrollaron los lenguajes ensambladores para acelerar el proceso de programación, lo que llevó a la necesidad de diseñar un programa administrador de los recursos con que cuenta un sistema. Fue así como fue concebido y desarrollado el concepto de un programa especial llamado sistema operativo.

1.2.1 Sistemas de procesamiento por lotes.

Los primeros sistemas operativos implantados como tales generalmente ejecutaban sólo una tarea a la vez y simplificaban la transición entre tareas para obtener la máxima utilización del sistema de cómputo. Estos sistemas se denominaron *sistemas de procesamiento por lotes de secuencia única*, ya que los programas y los datos se proporcionaban a la computadora en grupos o en lotes para su ejecución. La idea básica era reunir un conjunto de tareas para introducirlo a la computadora. Después de esto, el operador de la máquina cargaba un programa especial que leía el primer trabajo y lo ejecutaba. Una vez desplegada la salida del primer trabajo se ejecutaba la siguiente tarea. Así, una vez ejecutadas todas las tareas del lote, el operador proseguía a ejecutar el siguiente lote de tareas.

A partir de entonces, la tendencia en la evolución de los sistemas de cómputo llevó a la utilización de sistemas de procesamiento por lotes, los cuales podían aprovechar mejor los recursos del sistema mediante la ejecución de varias tareas simultáneamente. Este tipo de sistemas incluían usualmente dispositivos periféricos

(lectores y perforadoras de tarjetas, impresoras, unidades de cinta y unidades de disco), sin embargo los procesos rara vez utilizaban eficientemente dichos recursos, debido principalmente a que, mientras que una tarea esperaba a que se completara una operación de entrada o salida para poder seguir con la utilización del procesador, otras tareas quedaban *ociosas*. Por ese motivo, fue necesario tratar de evitar que el procesador permaneciera desocupado asignándolo a otras tareas que se encontraban en espera. Así, cuando una tarea estaba utilizando el procesador, otras podrían utilizar los diversos dispositivos de entrada y salida.

1.2.2 Multiprogramación.

Con lo anterior, la mejor forma de aprovechar al máximo los recursos del sistema parecía ser ejecutar una combinación de diversas tareas al mismo tiempo. Los diseñadores de sistemas operativos observaron que mientras una tarea esperaba a que se completara cierta operación de E/S para seguir utilizando el procesador, otra tarea podría aprovechar el procesador ocioso. De manera similar, cuando una tarea se encontraba utilizando el procesador, otras podrían utilizar los dispositivos de E/S.

De esta forma fue como nació el concepto de *multiprogramación*, en el cual varias tareas se encuentran a la vez en memoria principal y el procesador se traslada de una tarea a otra según fuera requerido para lograr que las tareas avanzaran en su ejecución mientras se mantenían en uso los dispositivos periféricos.

La esencia del funcionamiento de este tipo de sistemas con multiprogramación radica en la compartición del procesador por varios programas, con lo que se conseguía reducir los tiempos de ejecución totales con respecto a los sistemas de procesamiento por lotes. Sin embargo no siempre era así. En ocasiones sucedía que un programa con alta prioridad de ejecución podía ocupar el procesador demasiado

tiempo, impidiendo que otros programas pudieran compartirlo, con lo que se veía afectado el rendimiento en general del sistema. Este problema pudo solucionarse con un sistema operativo llamado de *tiempo compartido*.

1.2.3 Sistemas de tiempo compartido.

El concepto de *tiempo compartido* surgió como una extensión de la multiprogramación, en el que se asignan intervalos fijos o variables de tiempo del procesador a múltiples programas, es decir, proporciona igualdad de oportunidades a todos los programas y procesos que compiten por el procesador. Los tiempos de ejecución que se ahorran en este tipo de sistemas pueden ser incluso mayores que los que se obtienen en sistemas de procesamiento por lotes o de multiprogramación.

Al mismo tiempo, la compartición del tiempo de atención del procesador por parte de varios programas a la vez da origen al concepto de *procesadores virtuales*, ya que cada usuario del sistema, conectado desde una terminal, puede interactuar con el procesador de forma instantánea, por lo que se hace aparente que cada usuario posee su propio procesador.

Así, los sistemas de tiempo compartido resultan ser muy eficaces cuando se emplean en sistemas con un procesador conectado a varias terminales interactivas y se desarrollaron principalmente para permitir trabajar simultáneamente a un gran número de usuarios interactivos.

En su evolución, los sistemas de tiempo compartido lograron trabajar con múltiples modos de operación, ya que también realizan procesamiento por lotes y ejecución de aplicaciones de *tiempo real*, que se caracterizan por dar respuestas inmediatas a las diversas peticiones de los usuarios del sistema y por su disponibilidad permanente.

1.2.4 Sistemas Distribuidos.

Con el origen de los microprocesadores se da el gran desarrollo de las *computadoras personales* y las *estaciones de trabajo*. La tecnología de los microprocesadores, en los que se integran miles de transistores en un centímetro cuadrado, permite la construcción de computadoras de escritorio muy poderosas a bajo costo, con lo que un usuario puede tener su propia computadora dedicada para realizar la mayor parte de sus trabajos y puede utilizar las facilidades de comunicación para transmitir datos entre sistemas. De esta forma, el cómputo se *distribuye*, evitando el llevar a procesar datos a un centro de cómputo centralizado. Esta tendencia da origen a lo que se conoce como la *computación distribuida*, que permite que los cómputos se dividan en subcómputos que pueden ejecutarse en otros procesadores distribuidos.

En los sistemas operativos en red actuales, los usuarios tienen conocimiento de la existencia de múltiples computadoras y pueden ingresar en equipos remotos y reproducir archivos de un equipo a otro. Cada uno de estos equipos ejecuta su sistema operativo local y tienen un usuario (o usuarios) propio. En cambio, los *sistemas operativos distribuidos* se presentan ante los usuarios como sistemas uniprocesadores tradicionales, que en realidad están compuestos de múltiples procesadores. En un sistema distribuido real, los usuarios no tienen conocimiento de dónde se están ejecutando sus programas o dónde están ubicados sus archivos, todo eso se debe manejar de forma automática y eficiente por medio del sistema operativo.

Los sistemas operativos distribuidos son completamente distintos a los sistemas operativos uniprocesadores. Por ejemplo, los sistemas distribuidos a menudo permiten que los programas corran en varios procesadores al mismo tiempo, con lo que se

requieren algoritmos de planificación del procesador más complejos a fin de optimizar la forma en que se distribuye el trabajo.

La tolerancia a fallas es otra área donde los sistemas distribuidos son diferentes Es común que estos sistemas se diseñen para que operen aun si parte del hardware se encuentra dañado. Los diseños tolerantes a fallas son importantes sobre todo en sistemas en los que quizá no puedan intervenir los seres humanos para solucionar los problemas.

1.3. Transputers.

Como ya se mencionó, los sistemas actuales tienen tendencias claras hacia la computación distribuida, que involucra arquitecturas de procesadores múltiples y *paralelismo*. En la actualidad, existen lenguajes de programación que permiten que una aplicación sea descrita como una colección de procesos que operan de forma concurrente y que se comunican a través de distintos canales. En esta descripción, cada proceso representa el comportamiento de un componente en una aplicación, al igual que cada canal describe una conexión entre los diferentes componentes

Estas tendencias hacia el procesamiento distribuido también se han visto reflejadas en la arquitectura empleada por procesadores VLSI (*very-large-scale integrated*), en los que es posible que un gran número de dispositivos idénticos sean manufacturados a un bajo costo. Bajo este concepto es como nacen los microprocesadores construidos explícitamente para el procesamiento concurrente: *los transputers.*

Para generalizar, un Transputer es un dispositivo sencillo VLSI con memoria, procesador y ligas de comunicación para una conexión con otros transputers. El concepto es simple: colocar, en un solo chip, un procesador, memoria y cuatro ligas de

comunicación, para después emplear varios de estos chips juntos y hacerlos trabajar en *forma concurrente. De esta forma, sistemas concurrentes* pueden ser construidos con un conjunto de transputers que operan juntos y que se comunican entre si por medio de ligas.

1.3.1 Arquitectura del Transputer.

Un transputer consiste a su vez de un determinado número de transputers, cada uno de los cuales ejecuta un proceso y se comunica con los otros transputers. Como un proceso es ejecutado por un transputer, entonces puede consistir él mismo de un número determinado de procesos concurrentes que tiene que soportar el modelo de programación interno definido. Con el procesamiento concurrente con transputers es posible implementar un procesador compartido entre los diferentes procesos concurrentes que existan. La figura 1.1 ilustra la arquitectura del transputer.

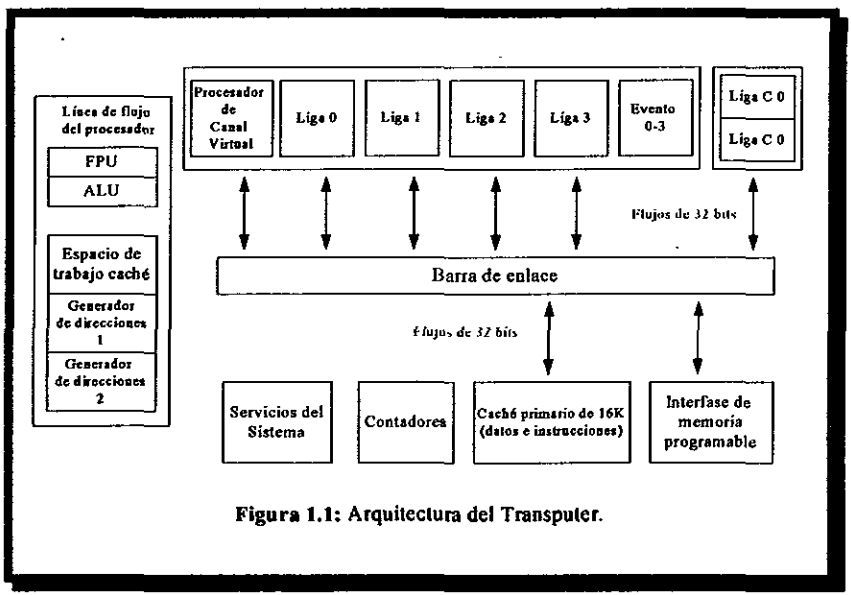

Figura 1.1: Arquitectura del Transputer.

Las unidades principales que conforman la arquitectura del transputer son: el núcleo del procesador, formado por una Unidad Aritmética y Lógica (ALU) y una Unidad de Operaciones de Punto Flotante (FPU), memoria caché, cuatro ligas de comunicación serial, un procesador de canal virtual (VCP) y una interfase de memoria programable. También posee dos contadores, cuatro pares de canales de eventos para la sincronización interna de los procesos con los eventos externos y dos ligas de control que permiten el flujo de señales de control entre otros transputers. A lo largo del chip se encuentra una barra de enlace que controla los cuatro diferentes flujos de datos.

Fue una decisión de diseño el que el transputer debería de ser programado en un lenguaje de alto nivel para lograr la ejecución de procesos concurrentes. El conjunto de instrucciones fue diseñado para una compilación sencilla y eficiente, con un número pequeño de instrucciones, todas con el mismo formato, y con una representación compacta de las operaciones usadas más frecuentemente. El conjunto de instrucciones es independiente del tamaño de palabra del procesador, lo que permite que el mismo microcódigo pueda ser usado otros transputers con diferente tamaño de palabra en el procesador.

El diseño de las instrucciones es sencillo: cada instrucción consiste de un byte único, dividido en 4 pares de bits, los cuatro bits más significativos del byte son una función de código, y los 4 menos significativos son el valor del dato.

El diseño del transputer explota la disponibilidad de la velocidad de memoria contenida en el propio chip, teniendo un número reducido de registros. Esta pequeña cantidad de registros, junto con la simplicidad de su conjunto de instrucciones permiten al procesador tener un control lógico y rutas de datos veloces y relativamente simples.

Los seis registros son:

- El apuntador del espacio de trabajo, que apunta a un área de almacenamiento donde se alojan a las variables locales.
- El apuntador de instrucciones, que apunta a la siguiente instrucción a ser ejecutada.
- El registro del operador, que se usa en la formación de operandos de instrucciones.
- Los registros A, B y C, los cuales forman una pila de evaluación y son las fuentes o destinos para la mayoría de las operaciones aritméticas y lógicas.

Las expresiones son evaluadas en una *pila de evaluación*, y las instrucciones hacen referencia a la pila de forma implícita Con el uso de pilas se deshecha la necesidad de instrucciones para especificar la localización de sus operandos.

La figura 1.2 ilustra la localización de los registros.

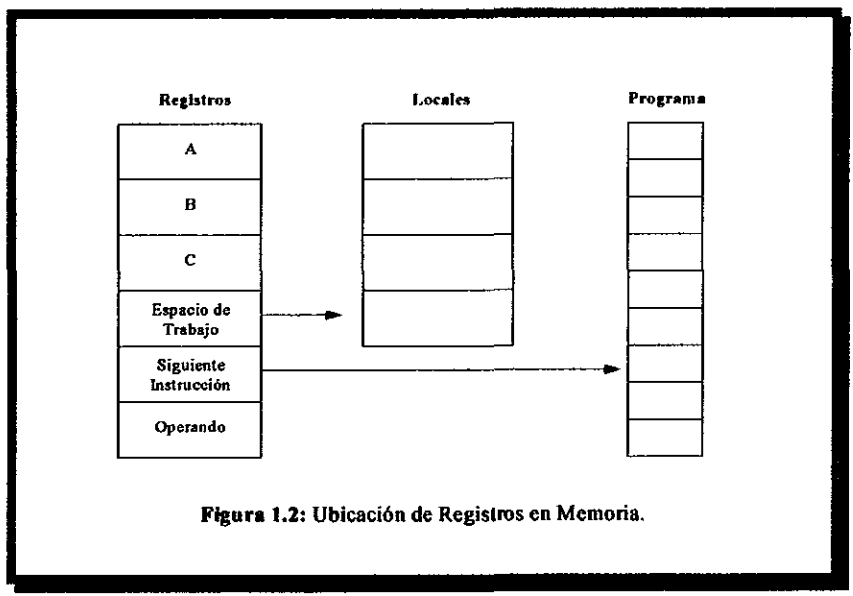

Figura 1.2: Ubicación de Registros en Memoria.

En un diseño como este no se requiere de ningún mecanismo de hardware para detectar más de tres valores que hayan sido cargados en la pila. Esto facilita al compilador asegurarse de que esto nunca sucederá.

1.3.2 El lenguaje Occam.

Occam es un lenguaje de programación concurrente usado por los transputers. Occam permite que un sistema pueda ser descrito como un conjunto de procesos concurrentes, los cuales se comunican unos con otros entre sí y además con dispositivos periféricos por medio de los canales. Los programas Occam son creados a partir de 3 procesos primitivos:

1. Asignar la expresión *e* a la variable *v*. (v:=e).
2. Desplegar la expresión *e* en el canal *c*. (c!e).
3. Introducir desde el canal *c* a la variable *v*. (c?v).

Estos procesos primitivos se combinan usando *construcciones*:

SEQuential: componentes que se ejecutan uno después de otro.

PARalell: componentes que son ejecutados en el mismo intervalo de tiempo.

ALTernative: el primer proceso en estar listo es ejecutado.

WHILE: ejecución repetida de procesos.

IF: procesos ejecutados de forma condicional.

Así, una *construcción* es por sí misma un proceso, que puede ser usado como el componente de otra *construcción*. Para programas secuenciales convencionales pueden ser expresados con variables y asignaciones, combinados en *construcciones secuenciales*. En el caso de programas concurrentes, estos pueden expresarse con canales, entradas y salidas, que son combinadas en *construcciones paralelas*. En Occam, un canal brinda una ruta de comunicación entre dos procesos concurrentes.

El lenguaje Occam ha demostrado que gran número de aplicaciones pueden descomponerse en varios procesos simples. Una vez que una aplicación ha sido descrita por medio del lenguaje Occam, es posible llevar a cabo varias implementaciones de la misma. De esta forma, un transputer consiste a su vez de un conjunto de transputers, cada uno ejecutando un proceso Occam.

1.3.3 Mecanismos de comunicación entre procesos.

En un transputer, la comunicación entre procesos se consigue por medio de canales. La comunicación con lenguaje Occam es *punto a punto* y sincronizada, como resultado, un canal de comunicación no necesita ni cola de procesos, ni cola de mensajes, ni el uso de un buffer para mensajes.

Existen dos tipos de canales: internos y externos. Los primeros se emplean cuando la comunicación se realiza entre procesos ejecutándose en el mismo transputer y se implementa mediante un espacio de memoria (del tamaño de una *palabra*). El canal externo se emplea para comunicar procesos ejecutándose en transputers diferentes y se implementa mediante ligas *punto a punto*.

El procesador provee de un número de operaciones para soportar el flujo de mensajes, las más importantes son:

1. *Input message* (mensaje de entrada).
2. *Output message* (mensaje de salida).

Estas dos instrucciones usan la dirección del canal para determinar si el canal es interno o externo. Esto significa que la misma secuencia de instrucciones puede ser usada para ambos canales, permitiendo a un proceso ser escrito y compilado sin el conocimiento de cuáles son los canales conectados.

Además, el transputer cuenta con un *planificador* en microcódigo que permite que cualquier número de procesos concurrentes sean ejecutados juntos, compartiendo el tiempo del procesador. Con este planificador, se elimina la necesidad de un software dedicado para ello, ya que opera de forma tal que desactiva a aquellos procesos que no consumen tiempo del procesador.

De esta forma, el transputer tiene dos usos importantes, primeramente provee de un innovador sistema de construcción por bloques, el cual permite que el lenguaje Occam sea utilizado, tanto como un lenguaje para la descripción de un sistema concurrente como un lenguaje de programación, principalmente en aplicaciones diseñadas para la ejecución concurrente de procesos. En segundo lugar, el lenguaje Occam y el transputer pueden ser usados como prototipos de sistemas altamente concurrentes, en los que el procesamiento individual o secuencial es en lo último que se piensa al llevar a cabo la implementación de un sistema en un hardware dedicado.

CAPITULO 2
LOS SISTEMAS PARALELOS

2.1 El enfoque del procesamiento paralelo.

A medida que el hardware ha ido disminuyendo su tamaño y costo, también han ido apareciendo tendencias bien definidas hacia el multiprocesamiento, el procesamiento distribuido y el *paralelismo*. El incremento en la disponibilidad de microprocesadores a bajo costo ha permitido la producción de sistemas multiprocesadores, con lo que el concepto de desarrollo de actividades en forma secuencial se ha visto desplazado por lenguajes de programación concurrente, en los que es posible describir varias actividades en un mismo intervalo de tiempo, dando origen a los conceptos de *paralelismo* y *procesamiento paralelo.*

El término *paralelismo* involucra multiprocesamiento, es decir, el uso de múltiples procesadores para ejecutar en forma verdaderamente simultánea diferentes partes de una actividad o bien procesos concurrentes que se encuentran compartiendo una memoria común Varios procesos son *concurrentes* si existen simultáneamente. Estos procesos pueden funcionar en forma totalmente independiente unos de otros, o ser *asíncronos,* lo cual significa que en ocasiones requieren de cierta sincronización y cooperación.

2.1.1 Definición de procesamiento paralelo.

El procesamiento paralelo es una forma de procesamiento de información que favorece la explotación de los sucesos concurrentes en un proceso de computo. Es un término que se usa para denotar a un conjunto de técnicas empleadas para

proporcionar tareas simultáneas de procesamiento, con el fin de aumentar la velocidad de un sistema de cómputo. En lugar de procesar cada instrucción en forma secuencial, un sistema de procesamiento paralelo puede ejecutar procesamiento de datos en forma concurrente, para tratar conseguir el menor tiempo de ejecución.

Concurrencia a su vez implica paralelismo, simultaneidad y *solapamiento*. Los sucesos paralelos son los que pueden producirse en diferentes recursos durante el mismo intervalo de tiempo, a su vez, los sucesos simultáneos son los que pueden producirse en el mismo instante de tiempo y los sucesos solapados son los que pueden producirse en intervalos de tiempo superpuestos. El procesamiento paralelo exige la ejecución concurrente de varios programas, lo que contrasta con el procesamiento secuencial que sólo describe una actividad de cómputo a la vez. Los sucesos concurrentes involucrados en el procesamiento paralelo pueden darse en un sistema de cómputo en varios niveles de procesamiento:

- Nivel de trabajos o programas.
- Nivel de procedimientos o tareas.
- Nivel de inter-instrucciones.
- Nivel de intra-instrucciones.

2.1.2 *Niveles de paralelismo.*

El procesamiento paralelo puede abordarse en cuatro niveles de programación:

1. Nivel de programación o trabajos.

   Este es el nivel más alto del procesamiento paralelo, que se aplica a trabajos o programas múltiples a través de la multiprogramación, el *tiempo compartido* y el multiprocesamiento. En este nivel se requiere el desarrollo de algoritmos procesables en paralelo. La implementación de algoritmos en paralelo depende de

la asignación eficaz de los recursos, tanto de software como de hardware, a los múltiples programas que estén siendo utilizados para llevar a cabo determinada actividad.

2. Nivel de procedimientos o tareas.

   Este nivel de procesamiento paralelo se aplica a procedimientos o tareas dentro de un mismo programa. Esto supone la descomposición de un procedimiento o programa en múltiples tareas.

3. Nivel de inter-instrucciones.

   El tercer nivel trata de explotar la concurrencia entre múltiples instrucciones. En este nivel con frecuencia se realiza un análisis de dependencia de datos para detectar paralelismos entre instrucciones.

4. Nivel de intra-instrucciones.

   Por último, en el nivel inferior se implementa con frecuencia directamente por los medios de hardware. La partición del hardware se va incrementando desde los niveles superiores hasta los inferiores. Contrariamente, las implementaciones de software se incrementan desde los niveles inferiores hasta los superiores.

2.1.3 Aplicaciones del procesamiento paralelo.

En la actualidad existe una gran demanda de sistemas de cómputo veloces y eficaces en muchas áreas científicas y tecnológicas, para las cuales, la resolución de problemas científicos de gran escala aporta datos a dichos sistemas mediante el modelado y simulación de procesos que difícilmente son llevados a una práctica real. De esta forma, el procesamiento paralelo encuentra un gran campo de aplicación. Por ejemplo, la modelación multidimensional, aplicable en áreas tan variadas que van desde el medio ambiente hasta la econometría, se lleva a cabo mediante extensos

experimentos de simulación que frecuentemente acarrean cálculos a gran escala para obtener la precisión y el tiempo de respuesta deseados. En la actualidad, el desarrollo en la investigación en áreas tales como, las redes neuronales, la inteligencia artificial y la teoría fractal, ha requerido para su mejor entendimiento de sistemas paralelos para su mejor desempeño, ya que varias tareas que involucran estos temas pueden describirse como un conjunto de procesos simultáneos que pueden se ejecutados en sistemas multiprocesadores.

Los sistemas de procesamiento paralelo también han sido solicitados para la resolución de problemas de diseño en ingeniería tales como el análisis de elementos finitos, necesarios para diseños estructurales y experimentos para estudios aerodinámicos. En medicina, este tipo de sistemas son útiles en tomografía asistida, diseño de órganos artificiales para implantaciones humanas, estimación de daños cerebrales y estudios de ingeniería genética. En defensa militar son empleados para el diseño de armamentos, simulación de efectos y combates bélicos. Es decir, en la gran mayoría de las áreas de investigación científica básica, se necesita de sistemas de cómputo veloces y eficientes, como los sistemas de procesamiento paralelo, para avanzar en sus estudios y aprovechar las ventajas del modelado y simulación a través de sistemas de cómputo.

2.2 Multiplicidad de flujos de instrucciones y datos.

En esencia, el principal actividad de un sistema de computo es la ejecución de una secuencia de instrucciones sobre un conjunto de datos. Los sistemas multiprocesadores están constituidos por conjuntos de procesadores, con grupos de elementos de proceso de instrucciones que se conectan a los módulos de memoria y

periféricos a través de una *red* y su organización se basa en la forma en la que fluyen tanto los datos como las instrucciones.

El término *flujo* se emplea para denotar una secuencia de elementos, que pueden ser instrucciones o datos, que ejecuta o sobre los que opera un procesador único, es decir, este término es usado para referirse a una secuencia de datos o instrucciones que el sistema utiliza durante la ejecución de un programa. Por lo tanto, un *flujo de instrucciones* es una secuencia de instrucciones ejecutadas por el sistema, mientras que un *flujo de datos* es una secuencia de datos que incluye datos de entrada y los resultados parciales o totales solicitados o producidos por el flujo de instrucciones.

Las diferentes organizaciones de computadoras se caracterizan por la multiplicidad de hardware con que cuentan para atender los flujos de instrucciones y de datos y pueden clasificarse en cuatro categorías de acuerdo a su multiplicidad de flujos tanto de las instrucciones y como de los datos:

- Flujo simple de instrucciones-flujo simple de datos.
  (SISD: *Single Instruction Stream-Single Data Stream*).
- Flujo simple de instrucciones-flujo múltiple de datos.
  (SIMD: *Single Instruction Stream-Multiple Data Stream*).
- Flujo múltiple de instrucciones-flujo simple de datos.
  (MISD: *Multiple Instruction Stream-Single Data Stream*).
- Flujo múltiple de instrucciones-flujo múltiple de datos.
  (MIMD: *Multiple Instruction Stream-Multiple Data Stream*).

Este esquema para clasificar organizaciones de sistemas de cómputo considera la organización del sistema mediante la cantidad de instrucciones y unidades de datos

que se manipulan en forma simultánea. Cada categoría depende de la multiplicidad de los sucesos concurrentes que suceden en el sistema. Los diagramas de bloque que serán presentados ilustran estas cuatro organizaciones.

De forma conceptual, en los diagramas sólo son necesarios tres tipos de componentes:

- Módulo de memoria (MM).
- Unidad de Control (UC).
- Unidad de procesamiento (UP).

Las instrucciones y los datos son tomados del *módulo de memoria*. Las instrucciones se decodifican en la *unidad de control*, que envía el flujo de instrucciones decodificadas a las *unidades de procesamiento* para su ejecución.

Los flujos de datos circulan entre la unidad de procesamiento y la memoria de forma bidireccional, mientras que cada flujo de instrucciones es generado por una unidad de control independiente.

2.2.1 Flujo simple de instrucciones-Flujo simple de datos (SISD).

Este tipo de organización SISD representa a la mayoría de los sistemas de cómputo de la actualidad, en donde las instrucciones son ejecutadas de forma secuencial, pero pueden encontrarse solapadas en las etapas de ejecución

Un sistema con organización SISD puede tener más de una unidad funcional, pero todas las unidades funcionales se encuentran bajo la supervisión de una única unidad de control.

La figura 2.1 muestra el diagrama de bloques de esta organización.

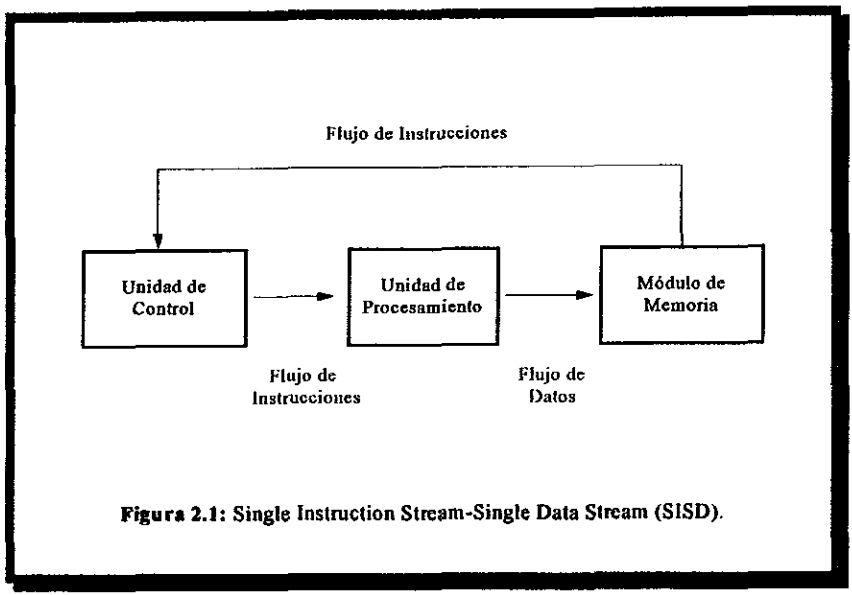

Figura 2.1: Single Instruction Stream-Single Data Stream (SISD).

Este tipo de organización, con un solo flujo de instrucciones y de datos, representa la organización de una computadora única que contiene una unidad de control, una unidad de procesamiento y una unidad de memoria.

Esta arquitectura es la más común y basa su organización esencialmente en la arquitectura de sistemas de cómputo de un sólo procesador que procesan una instrucción a la vez, es decir, los sistemas con organización SISD procesan datos de un flujo único de datos.

2.2.2 Flujo simple de instrucciones-Flujo múltiple de datos (SIMD).

La organización SIMD, ilustrada en la figura 2.2, corresponde sistemas con procesadores matriciales, en donde existen múltiples unidades de proceso supervisadas por la misma unidad de control. Todos estas unidades de proceso reciben la misma instrucción emitida por la unidad de control, pero operan sobre diferentes conjuntos

de datos procedentes de flujos distintos. El subsistema de memoria compartida puede contener múltiples módulos de memoria para que pueda comunicarse con todos los procesos simultáneamente.

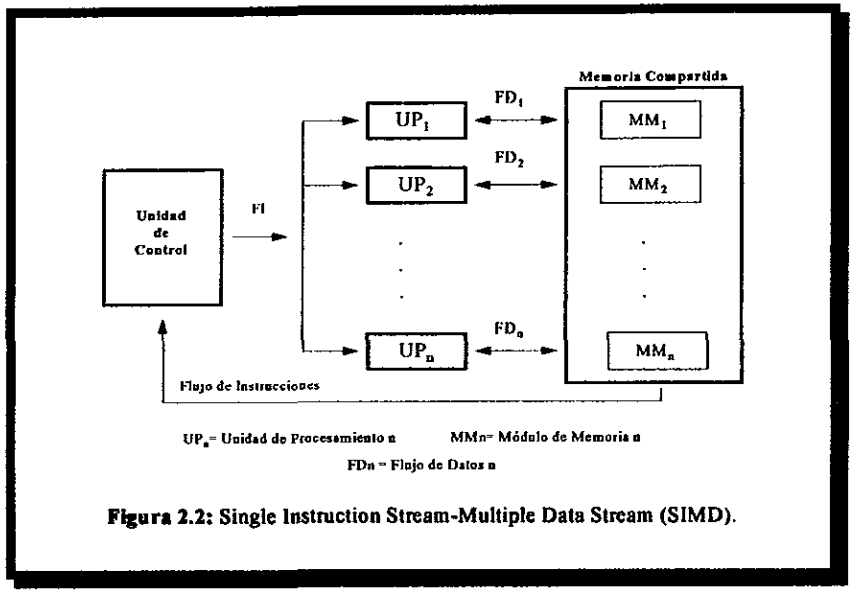

Figura 2.2: Single Instruction Stream-Multiple Data Stream (SIMD).

Generalmente los sistemas con procesadores matriciales ejecutan cálculos sobre grandes arreglos de datos con una organización de instrucción única y datos múltiples. Las unidades de procesamiento operan de forma sincronizada para ejecutar la misma operación bajo el control de una unidad común, por lo que proporciona una organización de flujo de instrucciones único con un flujo múltiple de datos ejecutando la misma operación simultáneamente en cada elemento de un arreglo.

Los sistemas con un flujo simple de instrucciones y flujo múltiple de datos son altamente especializados y muy convenientes para resolver problemas numéricos que pueden expresarse en forma de vector o matriz.

2.2.3 Flujo múltiple de instrucciones-Flujo simple de datos (MISD).

En el concepto de la organización de los sistemas con organización MISD existen *n* unidades procesadoras, cada una de las cuales recibe distintas instrucciones que operan sobre el mismo flujo de datos y sus derivados, como se muestra en la figura 2.3.

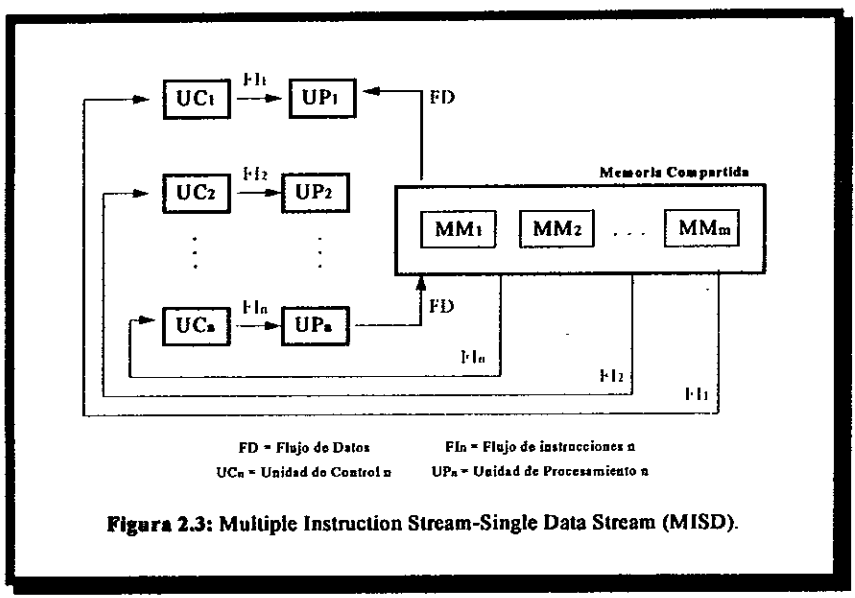

Figura 2.3: Multiple Instruction Stream-Single Data Stream (MISD).

En este caso, los resultados generados por las unidades procesadores (es decir, las salidas) se transforman en la entrada (es decir, los operandos) del siguiente procesador en el esquema.

Esta organización de sistemas de cómputo es muy poco utilizada y poco práctica en la actualidad, de hecho no existe ninguna materialización real de esta categoría de computadoras. La ejecución concurrente de procesos en un sistema con

un flujo múltiple de instrucciones da origen a la organización de los sistemas conocidos como *multiprocesadores*: la organización MIMD.

2.2.4 Flujo múltiple de instrucciones-Flujo múltiple de datos (MIMD).

Dentro de la categoría de los sistemas con organización *MIMD* se encuentran la mayoría de los sistemas multiprocesadores y de los sistemas con computadoras múltiples. Su organización se muestra en la figura 2.4.

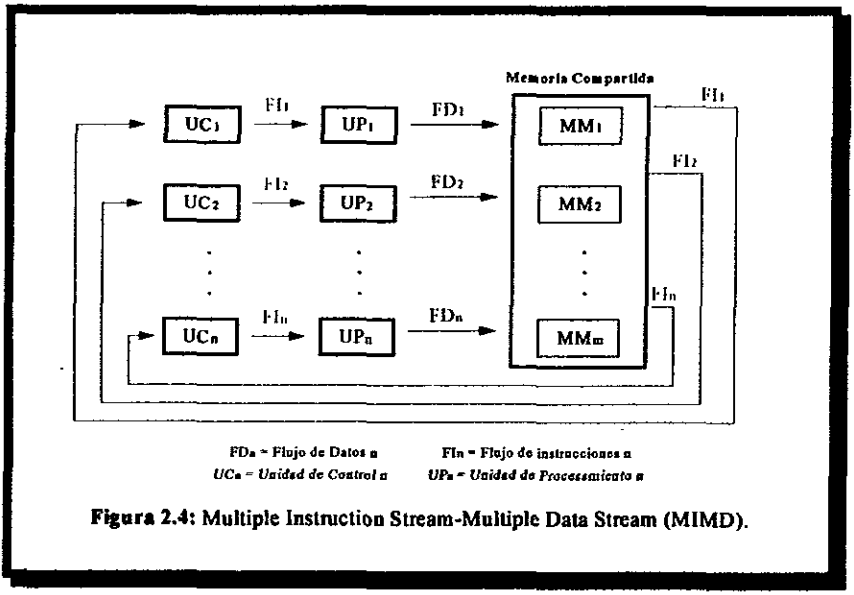

Figura 2.4: Multiple Instruction Stream-Multiple Data Stream (MIMD).

Un sistema con múltiples flujos de datos y de instrucciones implica interacciones entre las *n* unidades procesadoras, debido principalmente a que todos los flujos de memoria se derivan del mismo espacio de datos, que es compartido por todos los procesadores.

Si los *n* flujos de datos provinieran de diferentes espacios dentro de la memoria compartida, entonces se obtendría lo que se conoce como una organización *SISD*

múltiple, que significa que se tiene un conjunto de *n* sistemas monoprocesador SISD independientes.

2.3 Organización y arquitectura de los sistemas paralelos.

La *arquitectura paralela* es una técnica para el desarrollo de arquitecturas de sistemas en la que descompone un proceso secuencial en suboperaciones, y cada subproceso se ejecuta en un segmento dedicado especial que opera en forma concurrente con los otros segmentos. Estos sistemas poseen ciertas características básicas que permiten dividirlos en tres configuraciones arquitectónicas:

- Sistemas de segmentación encauzada.
- Procesadores matriciales.
- Sistemas multiprocesadores.

Estas tres categorías no son mutuamente excluyentes, de hecho, la mayoría de los sistemas son de segmentación encauzada y algunos de ellos adoptan también una estructura matricial o multiprocesadora.

2.3.1 Sistemas de segmentación encauzada.

Normalmente la ejecución de una instrucción implica cuatro pasos principales:

I. Búsqueda de la instrucción desde una posición en memoria,
II. Decodificación de la instrucción para identificar la operación a efectuar,
III. Búsqueda del operando apropiado y
IV. Ejecución de la operación decodificada.

En un sistema *no encauzado*, estos pasos deben finalizar antes de que se procese la siguiente instrucción. En cambio, en sistemas de *segmentación encauzada* las instrucciones sucesivas se ejecutan de modo solapado.

Debido al solapamiento de instrucciones y ejecuciones aritméticas, estos sistemas se ajustan de forma muy adecuada para efectuar repetidamente las mismas operaciones a través del flujo de datos, por lo que los sistemas encauzados son muy útiles en procesamientos vectoriales, donde las operaciones sobre los componentes deben ser repetidas varias veces.

2.3.2 Procesadores matriciales.

A una colección síncrona de procesadores en paralelo se le denomina *procesador matricial*, correspondiente a una organización SIMD. Los procesadores matriciales están especialmente diseñados para realizar operaciones vectoriales sobre matrices o colecciones de datos.

En los procesadores matriciales, las unidades aritmética-lógicas son conocidas como *unidades de proceso* (UP), que pueden operar en paralelo. Entre las *unidades de proceso* debe establecerse un mecanismo adecuado de encadenamiento de datos.

Como se muestra en la figura 2.5, las instrucciones se ejecutan directamente en la *unidad de control*. Cada *unidad de proceso* esta formada por una unidad aritmética-lógica con registros (P) y una memoria (M) local. Las *unidades de proceso* se interconectan a través de una red de encaminamiento de datos.

La estructura de interconexión establecida para un cálculo específico está controlada desde la *unidad de control*, formada por una unidad aritmética-lógica de control (PC) y una memoria de control (MC).

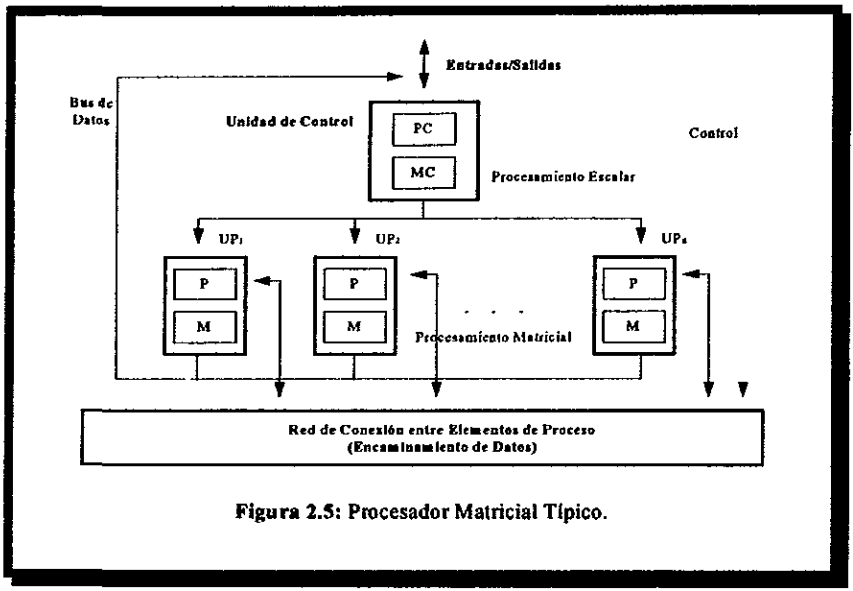

Figura 2.5: Procesador Matricial Típico.

Las unidades de proceso son dispositivos pasivos sin capacidad de decodificación de instrucciones. Las instrucciones son transmitidas a las unidades de proceso para su ejecución distribuida sobre diferentes operandos, tomados directamente de las memorias locales. La unidad de control es la encargada de la búsqueda de la instrucción (ya sea en las memorias locales o en la memoria de control) y de su decodificación.

2.3.3 Sistemas multiprocesador.

Los sistemas multiprocesador contienen dos o más procesadores de capacidades semejantes compartiendo acceso a grupos comunes de módulos de memoria, canales de E/S y dispositivos periféricos. Todo el sistema debe estar controlado por un sólo sistema operativo integrado que facilite las interacciones entre los procesadores y sus programas a diferentes niveles. Además de las memorias y los dispositivos de E/S compartidos, cada procesador dispone de una memoria local y de

dispositivos privados. Las comunicaciones entre procesadores pueden realizarse a través de las memorias compartidas o mediante una red de interrupción.

Una de las principales ventajas de este tipo de sistemas es la tolerancia a fallas, ya que si una unidad de procesado falla es posible continuar trabajando con los procesadores restantes. El sistema operativo debe percatarse de la falla de algún procesador en específico para reemplazarlo y evitar asignarle la realización de algún proceso. Los sistemas multiprocesador corresponden a una organización MIMD que es el tipo de organización que más favorece al procesamiento paralelo. La figura 2.6 muestra de forma conceptual la organización básica de un sistema multiprocesador.

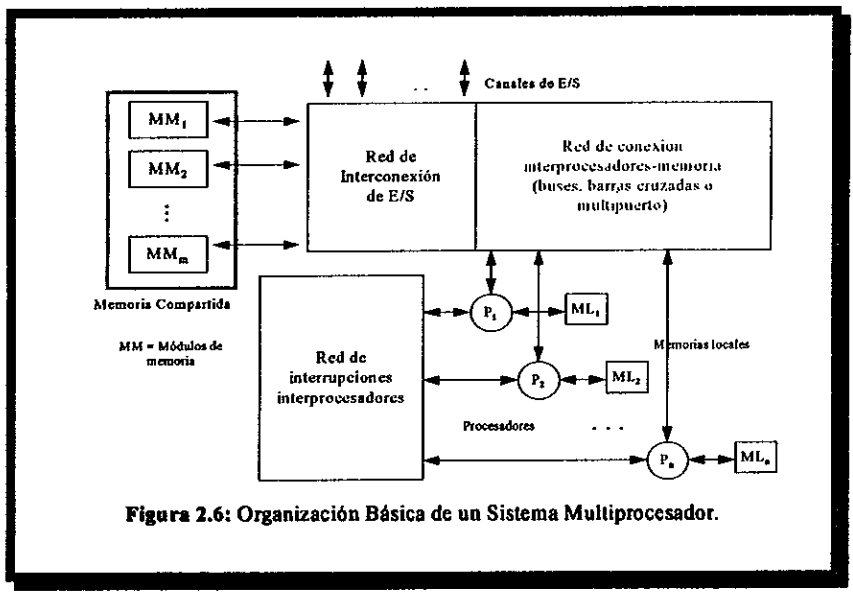

Figura 2.6: Organización Básica de un Sistema Multiprocesador.

La organización del hardware para el diseño de un sistema multiprocesador esta determinado principalmente por la *estructura de interconexión* entre las memorias y los procesadores y entre las memorias y los dispositivos de E/S.

2.4 Estructuras de interconexión para sistemas multiprocesador.

La interconexión entre los componentes de los sistemas multiprocesadores puede tener diferentes configuraciones físicas, dependiendo de la cantidad de trayectorias de transferencia disponibles entre los procesadores y la memoria, en el caso de sistemas con memoria compartida, o entre los elementos de procesamiento en el caso de sistemas con memoria distribuida.

Los tipos de interconexión más comunes son:

- Bus común de tiempo compartido.
- Red de computadora de barras cruzadas.
- Memorias multipuerto.
- Sistemas de hipercubo.

2.4.1 Bus de tiempo compartido.

Un bus de tiempo compartido es el sistema de interconexión más simple para un sistema multiprocesador, en el que un procesador comparte un conjunto de módulos de memoria principal y, posiblemente, dispositivos de E/S. Un sistema multiprocesador de bus común consta de varios procesadores conectados mediante una trayectoria común a una unidad de memoria. Las operaciones de transferencia las controla el procesador que controla el bus en ese momento. Cualquier otro procesador que desee iniciar una transferencia primeramente deberá determinar el estado de disponibilidad del bus y, sólo después de que el bus queda disponible, puede direccionar la unidad destino para iniciar la transferencia.

Esta organización es la menos compleja y la más fácil de configurar, como lo muestra la figura 2.7.

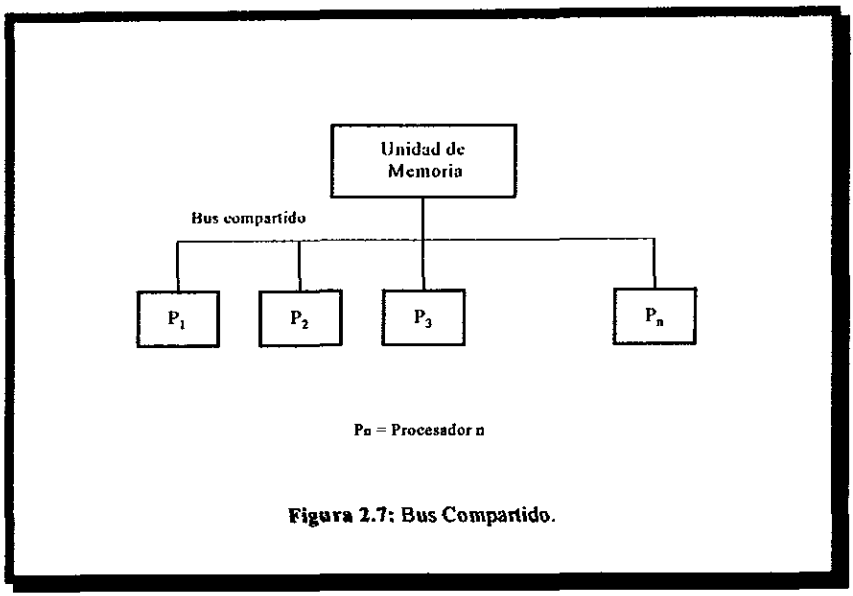

Figura 2.7: Bus Compartido.

Un sistema tan sencillo como este se encuentra limitado a realizar sólo una transferencia a la vez. Esto significa que cuando un procesador se está comunicando con la memoria, el resto de los procesadores están ocupados con operaciones internas o bien inactivos en espera del bus. La consecuencia de esta limitante radica en que la velocidad de transferencia total dentro del sistema está limitada a la velocidad del bus único.

2.4.2 Red de computadora de barras cruzadas.

Si se incrementa el número de *buses* en tiempo compartido de un sistema puede lograrse que siempre exista un camino disponible para cada unidad de memoria, es entonces cuando la red de interconexión se denomina *red de barras cruzadas*. Este tipo de organización consta de varios *puntos de cruz* que se colocan como intersecciones entre los buses del procesador y las trayectorias del módulo de memoria. La figura 2.8 muestra una red de computadora de barras cruzadas en donde

el pequeño cuadro en cada punto de cruz es un *conmutador* que se encarga de determinar la trayectoria de un procesador a un módulo de memoria.

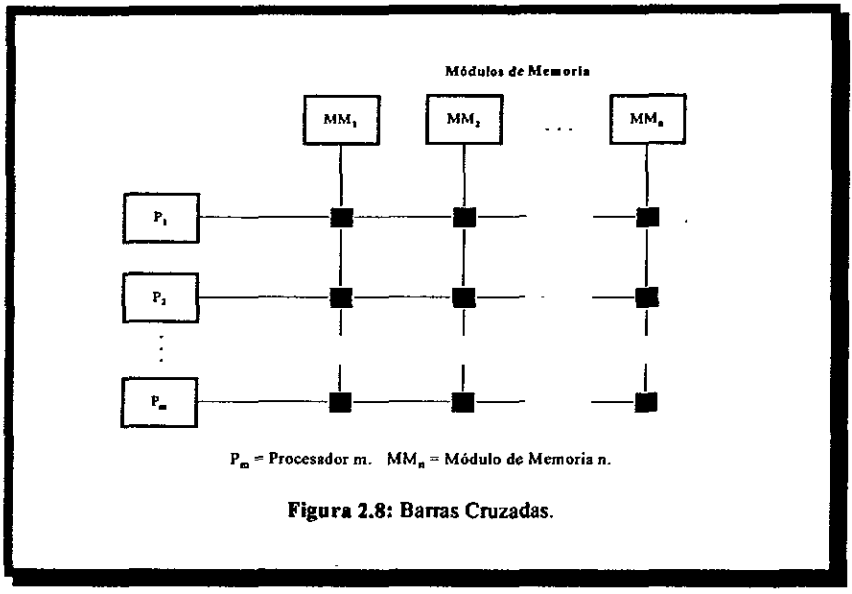

Figura 2.8: Barras Cruzadas.

Este tipo de organización soporta transferencias simultáneas de todos los módulos de memoria debido a que existe una trayectoria separada asociada con cada módulo. Sin embargo, los circuitos necesarios para implantarlo pueden resultar voluminosos y complejos.

2.4.3 Memorias multipuerto.

Un sistema de memorias multipuerto emplea buses separados entre cada módulo de memoria y cada procesador, como se muestra en la figura 2.9. Cada bus de procesador se encuentra conectado a cada módulo de memoria. Se dice que los módulos de memoria tienen *n* puertos, por lo que deben poseer una lógica de control

interna para determinar el puerto que tendrá acceso a memoria en determinado momento.

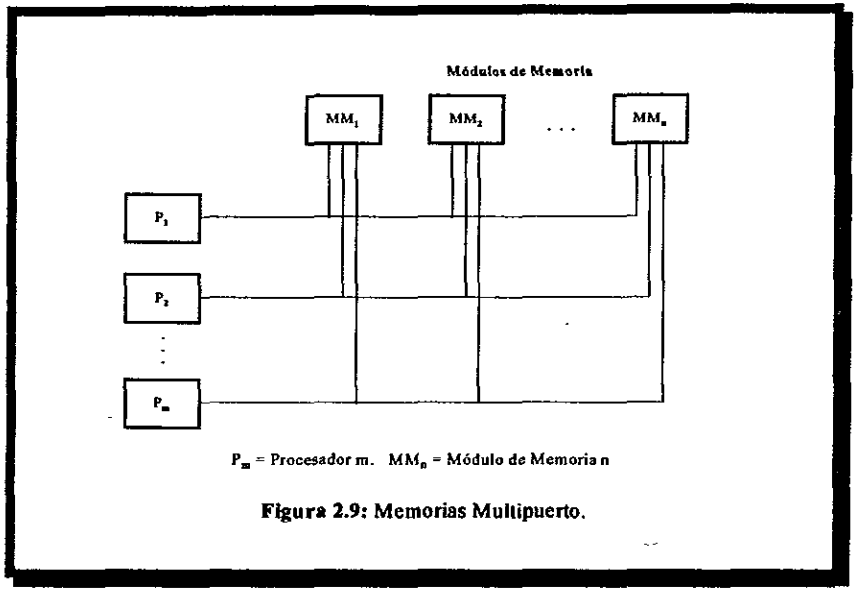

Figura 2.9: Memorias Multipuerto.

La ventaja de la organización de memorias multipuertos es la alta velocidad de transferencia que puede conseguirse debido a las trayectorias múltiples entre los procesadores y la memoria. No obstante, la principal desventaja es que requiere de una lógica de control de memoria y una gran cantidad de cableado, por lo que es más aplicable a sistemas multiprocesador con un número reducido de procesadores.

2.4.4 Interconexión de hipercubo.

Esta estructura de interconexión, también conocida como cubo *n* binario, es un sistema de memoria distribuida compuesto de 2^n procesadores interconectados en un cubo binario de *n* dimensiones. Cada procesador forma un nodo del cubo, a su vez, cada nodo del cubo no sólo contiene un procesador, sino que también contiene

memoria local e interfases de E/S. De acuerdo al cubo, cada procesador tiene trayectorias de comunicación directa a *n* procesadores vecinos. Estas trayectorias de comunicación corresponden a los lados del cubo de interconexión.

2.4.5 Estructuras funcionales.

Finalmente, existen dos modelos arquitectónicos diferentes para un sistema multiprocesador basados en la organización de la memoria: multiprocesador *estrechamente acoplado* y multiprocesador *ligeramente acoplado.*

- Sistema estrechamente acoplado.

En este modelo, el espacio de memoria principal es *simétrico,* es decir, todos los procesos se comunican a través de una memoria principal compartida, como se muestra en la figura 2.11.

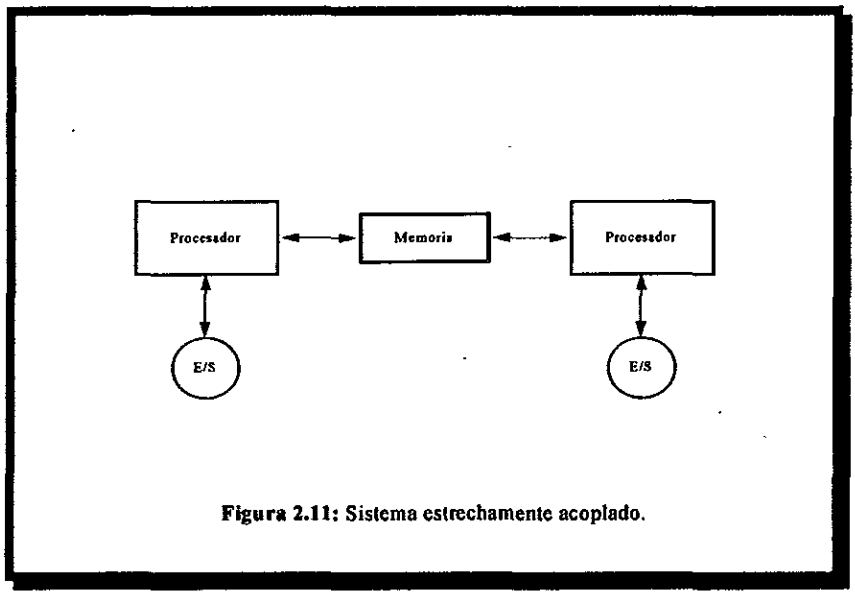

Figura 2.11: Sistema estrechamente acoplado.

De esta forma, la velocidad a la que se pueden comunicar datos de un procesador a otro esta en función del ancho de banda de la memoria. Los

procesadores normalmente ejecutan cada uno una copia del sistema operativo y se encuentran conectados mediante un bus común. Uno de los factores que limitan la expansión de un sistema estrechamente acoplado es la degradación del rendimiento del sistema debida a los conflictos que pueden presentarse cuando más de un procesador intenta acceder a la misma unidad de memoria concurrentemente.

- Sistema ligeramente acoplado.

En un diseño *ligeramente acoplado* el espacio de memoria es *asimétrico*, es decir, cada procesador posee su propio espacio de memora local e independiente lo suficientemente grande a donde acceden a la mayor parte de las instrucciones y de los datos. Cada procesador es autosuficiente, por lo que una falla en un procesador no afecta al resto. La figura 2.10 ilustra este diseño.

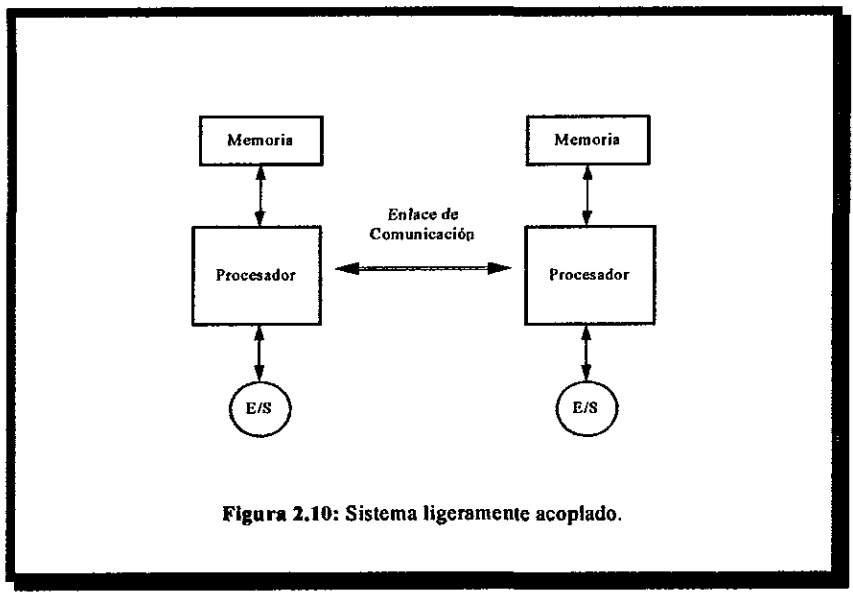

Figura 2.10: Sistema ligeramente acoplado.

Los sistemas ligeramente acoplados resultan eficientes cuando las interacciones entre las tareas son mínimas, mientras que los sistemas estrechamente acoplados pueden tolerar un alto grado de interacciones entre las tareas sin un deterioro significativo en el rendimiento.

2.5 Organizaciones de los Sistemas Operativos para multiprocesamiento.

La capacidad de aprovechar el paralelismo en el hardware y en los programas es fundamental en los sistemas multiprocesamiento en paralelo. Esta labor recae principalmente en el sistema operativo, que debe estar diseñado para aprovechar al máximo los recursos con que cuenta el sistema.

Una de las principales diferencias entre los sistemas operativos para los sistemas de procesamiento paralelo y los sistemas secuenciales es su organización y estructura con respecto a los múltiples procesadores. Las organizaciones básicas de los sistemas operativos para el multiprocesamiento son:

- Maestro/esclavo.
- Planificación coordinada de tareas.
- Organizaciones simétricas.

2.5.1 Maestro/esclavo.

En la organización *maestro/esclavo*, también conocida como *master/slave*, un procesador es designado como *maestro* y el resto de los procesadores son los *esclavos*. El maestro siempre se ejecuta sobre el mismo procesador y se encarga de mantener el estado de todos los procesadores del sistema, distribuyendo el trabajo entre los esclavos, es el encargado de ejecutar y dar mantenimiento al sistema operativo y de proveer de los diferentes servicios del sistema operativo, tales como operaciones de E/S. Por otro lado, los esclavos son tratados como recursos

planificables que ejecutan los trabajos asignados por el maestro, limitados por la cantidad de unidades de proceso disponibles.

Desde el punto de vista de la confiabilidad, si un procesador esclavo falla se pierde parte de la capacidad de cómputo, pero el sistema puede seguir funcionando. Mas sin embargo, si el procesador maestro falla, el sistema será incapaz de realizar operaciones de E/S. El principal problema que se tiene que enfrentar en esta organización es la *asimetría* del hardware, ya que sólo el procesador *maestro* puede ejecutar el sistema operativo, mientras que los procesadores *esclavos* sólo pueden ejecutar programas de usuario.

2.5.2 Planificación coordinada de tareas.

En esta organización, cuya arquitectura suele ser *ligeramente acoplada*, cada procesador ejecuta su propio sistema operativo y cada procesador es capaz de responder a las llamadas al sistema de los usuarios que estén ejecutando sus procesos. Una vez que un proceso inicia su ejecución en un procesador específico no desocupa dicho procesador hasta haber finalizado su ejecución. Esta organización es más confiable que la organización maestro/esclavo, debido a que es poco probable que la falla de un sólo procesador cause una falla total del sistema, ya que cada procesador controla sus propios recursos y las solicitudes de E/S regresan directamente a los procesadores que las iniciaron. Existe una contención mínima de los datos por el sistema operativo, ya que estos se encuentran distribuidos entre los sistemas operativos individuales de cada procesador para su propio uso.

Su principal desventaja es que dificulta el procesamiento paralelo, debido a que los procesadores no cooperan en la ejecución de un proceso individual y algunos

procesadores pueden permanecer inactivos mientras otros ejecutan procesos prolongados.

2.5.3 *Organización simétrica.*

Esta organización implica que todos los procesadores son idénticos. El sistema operativo administra un conjunto de procesadores idénticos y cualquiera de ellos se puede utilizar para controlar cualquier dispositivo de E/S o para hacer referencia a cualquier unidad de almacenamiento. Como varios procesadores pueden ejecutar el sistema operativo a la vez, se requiere de un código de exclusión mutua para reducir las regiones críticas. Sin embargo, y debido a la simetría del sistema, es posible equilibrar la carga de trabajo con mayor precisión que en otras organizaciones.

Los sistemas con organización simétrica resultan ser los más confiables y aptos para el procesamiento paralelo. Una falla en un procesador hace que el sistema operativo lo expulse del conjunto de procesadores disponibles y el sistema puede continuar funcionando mientras se reemplaza al procesador que falló.

Un proceso que se ejecute en un multiprocesador simétrico puede ser ejecutado en diferentes ocasiones por cualquiera de los procesadores equivalentes. En un momento dado, el procesador encargado de las funciones del sistema es denominado *procesador supervisor*, y sólo puede existir un supervisor a la vez, con lo que se evitan los conflictos sobre la información global del sistema. Sin embargo, se requiere de una cuidadosa estructura de datos para evitar la exclusión mutua por bloqueo provocada por la necesidad de varios procesadores a la vez de ser *supervisor*.

CAPITULO 3
SIMULACIÓN DE MULTITAREA A MULTIPROCESO. CASO UNIX SYSTEM V.

Desde sus inicios, el sistema operativo Unix se hizo muy popular cuando se ejecutaba en máquinas con diversos rangos de poder de procesamiento, desde *microprocesadores* hasta *mainframes*, brindando a todas ellas un ambiente común de ejecución de aplicaciones. Sin duda, actualmente Unix se ha convertido en el sistema operativo que representa el enlace entre casi todas las macrocomputadoras, minis, estaciones de trabajo, computadoras personales, supercomputadoras y redes. Uno de sus principales atributos, mediante el cual ha obtenido el éxito del que goza actualmente, es la posibilidad de lograr la combinación de ejecución de unos programas con otros, con lo que se fomenta el enfoque modular, de piezas de construcción y orientado a las herramientas, para el diseño de programas y aplicaciones.

Unix se encuentra escrito en lenguaje de programación C, y en sus inicios el sistema se distribuía con el código fuente incluido, por lo que muchos desarrolladores de sistemas hacían sus propias versiones con mejoras y modificaciones al sistema original, lo que en la actualidad resulta en la existencia de muchas variantes del sistema básico. A pesar de esto, se pueden enunciar varias características del sistema operativo que lo hacen tan popular:

- El sistema está escrito en lenguaje de programación C, que debido a su popularidad lo hace fácil de leer, entender, cambiar e instalar en otros equipos.

- Posee una interfaz con el usuario muy simple que brinda los programas y servicios que han hecho al sistema tan popular: *shell*, correo electrónico, procesadores de texto, etc.
- Utiliza un formato para archivos consistente, haciendo fáciles de escribir los programas de aplicación.
- Utiliza un sistema de archivos jerárquico que permite su fácil mantenimiento y eficiente implementación.
- Brinda una interfaz con los dispositivos periféricos simple y consistente.
- Es un sistema multiusuario y multitarea, por lo que varios usuarios pueden ejecutar varios procesos simultáneamente.
- Oculta la arquitectura del hardware del equipo al usuario, haciendo fácil escribir programas que se ejecutan en diferentes implementaciones de hardware.
- Brinda capacidad de interconexión entre procesos y manejo dinámico de memoria.

3.1 *Multiprocesamiento en ambiente* Unix.

El sistema operativo Unix está constituido básicamente por 3 capas o niveles:

i) *Núcleo del sistema.*

También llamado *kernel*, es la capa más interna y es la que interactúa directamente con el hardware del sistema. Principalmente se encarga del manejo de memoria, de la asignación de tiempos a cada proceso, del control de los recursos de E/S, de la administración del espacio en el sistema de archivos y de la supervisión de la transmisión de datos entre la memoria principal y los dispositivos periféricos.

ii) *Shell.*

Es la capa intermedia encargada de interactuar entre el kernel y las aplicaciones. Es un intérprete de comandos que pasa los comandos al kernel para su ejecución y a su vez transmite los resultados al usuario.

iii) *Aplicaciones.*

Es la capa externa del sistema operativo que incluye comandos, utilerías, procesadores de texto, programas, comunicaciones, bases de datos, etc.

El diseño original de Unix lo ubica como un sistema multitarea de tiempo compartido que asume el uso de una arquitectura de sistema con una unidad de proceso única. Sin embargo, en una arquitectura multiprocesador con dos o más unidades de proceso compartiendo memoria y dispositivos periféricos, se obtiene de forma potencial un mayor rendimiento, ya que los procesos pueden ser ejecutados concurrentemente en diferentes procesadores. El incremento en el rendimiento de un sistema multiprocesador estará en función de la capacidad de administración que el sistema operativo sea capaz de llevar a cabo.

La administración de un sistema multiprocesador ha llevado a una gran variedad de estudios e implementaciones, ya que los componentes replicados, particularmente aquellos que son heterogéneos o asimétricos, incrementan la cantidad de administración que debe proporcionar el sistema operativo. En su forma más general, el problema implica la planificación de un conjunto de procesos sobre un conjunto de procesadores, con características arbitrarias, a fin de optimizar alguna función objetiva, consecuentemente implica la selección de un proceso para su ejecución dentro de un conjunto de procesos.

A grandes rasgos, para la administración de procesos en ambiente Unix existen dos decisiones de asignación de recursos: una es en qué lugar situar el código y los

datos dentro de la memoria física y la otra es sobre qué procesador ejecutar cada proceso. En el caso de un sistema Unix administrando una arquitectura uniprocesador estas decisiones son triviales, debido a que las asignaciones están ya determinadas hacia una sola unidad de proceso, además, el espacio de direcciones de memoria física es accesible al único procesador del sistema, por lo que los conflictos de accesibilidad son relativamente sencillos de detectar y resolver. En cambio, para el caso de un sistema operativo Unix administrando una arquitectura multiprocesador, la administración de procesadores como un recurso compartido entre usuarios externos y procesos internos constará de dos clases de planificaciones: la planificación de carga externa a largo plazo y la planificación interna de los procesos.

La ejecución de procesos en sistemas Unix se divide en dos modos: modo *kernel* y modo usuario. Todos los procesos generados por aplicaciones y programas de usuario se ejecutan en *modo usuario*, pero cuando uno de estos procesos realiza una *llamada al sistema*, el modo de ejecución del proceso cambia de modo usuario a *modo kernel*, es entonces cuando el sistema operativo Unix ejecuta e intenta dar servicio a la solicitud del proceso, regresando un código de error si existe alguna falla.

Las diferencias entre ambos modos de ejecución de procesos son:

- Los procesos en modo usuario pueden accesar sus propias instrucciones y datos, pero no a las instrucciones y datos del *kernel* (o a los de otros procesos). Los procesos en modo *kernel*, sin embargo, pueden accesar a instrucciones y datos tanto del *kernel* como de los usuarios.
- Algunas instrucciones (por lo general instrucciones de bajo nivel) poseen privilegios y provocan errores si son ejecutadas en modo usuario. Por ejemplo, una instrucción que manipule el registro de estado del procesador deberá ser ejecutada

únicamente en modo kernel, por lo que los procesos ejecutándose en modo usuario no deben tener la facultad de ejecución de tal instrucción.

Aunque el sistema se ejecuta en alguno de los dos modos, el *kernel* se encarga de ejecutar los procesos de los usuarios. En realidad, el *kernel* no es un conjunto separado de procesos que se ejecuta paralelamente a los procesos de los usuarios, sino que es parte de cada uno de los procesos de los usuarios.

3.2 Problemas de la arquitectura multiprocesador.

El diseño original del sistema operativo Unix operando en sistemas uniprocesador logra su capacidad multitarea colocando juntos a los diferentes procesos a ser ejecutados en memoria compartida, y mediante un *algoritmo planificador* conmuta la atención del procesador de un proceso a otro. Cuando la atención del procesador cambia de un proceso a otro se dice que sucede un *cambio de contexto*. Estos cambios de contexto suceden en fracciones de segundo, por lo que, tanto para los usuarios como para sus aplicaciones, se vuelve aparente la existencia de un procesador dedicado para cada uno de ellos. En este sentido, Unix resuelve los conflictos de asignación y protege la integridad de las estructuras de datos del kernel de varias formas, una ellas es evitando que el kernel pueda ejecutar un proceso y cambiarse a otro mientras el primero siga en ejecución en modo kernel. En una arquitectura multiprocesador, sin embargo, si dos o más procesos tratan de ejecutarse en modo *kernel* en distintos procesadores, la estructura de datos del *kernel* podría alterarse o dañarse a pesar de las medidas de protección que se manejan en los sistemas uniprocesador.

Por ejemplo, la figura 3.1 muestra un fragmento del código fuente en lenguaje C del kernel.

```
structure cola {

} *bp, *bp1;
bp1 -> forp = bp -> forp;
bp1 -> backp = bp;
bp -> forp = bp1;

/* Aquí debe considerarse un posible cambio de contexto */

bp1 -> forp -> backp = bp1;
```

Figura 3.1: Colocando un *buffer* en una lista doblemente ligada

Este fragmento de código del *kernel* coloca una estructura de datos (apuntador *bp1*) después de una estructura existente (apuntador *bp*). Supóngase que dos procesos ejecutan este código simultáneamente en dos diferentes procesadores *A* y *B*, tal que el procesador *A* debe colocar la estructura de datos *bpA* después de *bp* y el procesador *B* debe colocar la estructura *bpB* después de *bp*. Sin considerar aún la relativa velocidad de ejecución de los procesadores, puede darse el peor de los casos en donde el procesador *B* pueda ejecutar las 4 instrucciones del código de la figura 3.1 antes que el procesador *A* pueda ejecutar otra instrucción, alterando la estructura de datos del kernel.

En este caso se puede manejar una interrupción con la que es posible retrasar la ejecución del código de la figura 3.1 en el procesador *A*, sin embargo, la alteración a la estructura de datos del *kernel* puede ocurrir aunque las interrupciones fueran bloqueadas, por lo que el *kernel* debe asegurarse que tales alteraciones a su estructura

nunca ocurrirán. Si dejara un proceso abierto en el cual esta situación de alteración pudiera ocurrir, el *kernel* estaría inseguro y su comportamiento sería impredecible.

En un entorno multiprocesador, existen tres formas para prevenir tales alteraciones al *kernel*:

1. Ejecutar toda actividad crítica en un sólo procesador, confiando en los métodos estándares utilizados en sistemas uniprocesador para la prevención de alteraciones al *kernel*.
2. Prevenir el acceso a regiones críticas de código ejecutable con la implementación de mecanismos de seguridad.
3. Rediseñar los algoritmos para evitar la contención para estructuras de datos del *kernel*.

3.3 Solución con procesadores maestro/esclavo.

Con una configuración mínima para un sistema multiprocesador con dos procesadores, es posible designar a un procesador como *maestro*, que puede ejecutarse en modo kernel y otro procesador denominado *esclavo* que solo puede ser ejecutado en modo usuario. Aunque esta implementación emplea únicamente dos unidades de proceso, la técnica puede extenderse hacia sistemas con un *maestro* y varios *esclavos*. Como ya se ha visto, el procesador *maestro* es responsable del manejo de todas las llamadas del sistema e interrupciones, mientras que los procesadores esclavos ejecutan procesos en modo usuario informando al procesador *maestro* cuando un proceso realiza una llamada al sistema.

El *algoritmo de planificación* de Unix, como el que se muestra en la figura 3.2, decide cuál procesador debe ejecutar un proceso. Existe un campo nuevo en la tabla de

procesos que designa al identificador de procesador (ID) sobre el que un proceso debe ejecutarse, que pueden ser, ya sea *maestro* o bien *esclavo*.

```
Algoritmo planifica_proceso(modificado)

entrada: ninguna
salida: ninguna

{
        while (no exista proceso seleccionado para ejecución)
        {
                if (proceso en el procesador maestro)
                    for (cada proceso en cola de ejecutables)
                        seleccionar el proceso de mayor prioridad
                        cargado en memoria;
                else
                    for (cada proceso en cola de ejecutables que no necesite
                        ejecutarse en procesador maestro)
                        seleccionar el proceso de mayor prioridad
                        cargado en memoria;
                if (no existe proceso elegible para ejecución)
                  desocupar el equipo;
                  /* ejecutar interrupción para colocar al equipo en espera*/
        }
        eliminar el proceso seleccionado de la cola;
        cambiar contexto hacia el proceso seleccionado, reanudar ejecución;
}
```

Figura 3.2: Algoritmo de Planificación de Procesos.

Cuando un proceso en un procesador *esclavo* realiza una llamada al sistema, el *kernel esclavo* coloca el campo del identificador del procesador ID en la tabla de procesos, indicando que el proceso deberá ejecutarse únicamente en el procesador *maestro*, haciéndolo de forma que pueda cambiar de contexto hacia la ejecución

planificada de otro proceso, como lo muestra el algoritmo llamada_sistema de la figura 3.3.

```
Algoritmo llamada_sistema
/* Algoritmo revisado para la invocación de una llamada a sistema */

entrada: número de llamada al sistema
salida: resultado de la llamada al sistema

{
        if (ejecutando en un procesador esclavo)
        {
                posicionar campo del ID del procesador en
                la tabla de procesos;
                realizar el cambio de contexto;
        }
        realizar el algoritmo regular para llamada a sistema;
        posicionar el campo ID del procesador a "cualquier" (esclavo)
        if (otros procesos deben ejecutarse en procesador maestro)
                realizar el cambio de contexto;
}
```

Figura 3.3: Algoritmo para un manejador de llamadas al sistema.

El kernel ejecutándose en el procesador maestro planifica el proceso con la más alta prioridad que debe ser ejecutado en el procesador maestro. Cuando la llamada a sistema finaliza, posiciona el ID del campo del procesador hacia un procesador esclavo, permitiendo su ejecución en modo *esclavo* una vez más.

Si el proceso debe ejecutarse en el procesador maestro, es preferible que el procesador maestro lo ejecute de inmediato y no lo mantenga en estado de espera. Esto se asemeja a un sistema uniprocesador cuando un proceso que realiza una llamada

"urgente" al sistema es ejecutado tan pronto como es posible, asignándole la más alta prioridad.

Si el procesador maestro estuviera ejecutando un proceso en modo usuario cuando un procesador esclavo solicita el servicio de una llamada a sistema, el procesador maestro continua su ejecución hasta que el algoritmo de planificación defina el siguiente cambio de contexto en el esquema de ejecución de procesos. El procesador maestro podrá responder de forma más rápida y eficiente si el procesador esclavo define una bandera global que el procesador maestro pueda verificar en el manejador del reloj de interrupciones; así entonces, el procesador maestro realizará el cambio de contexto en, a lo sumo, un tiempo de reloj. Alternativamente, el procesador esclavo podría interrumpir al procesador maestro y forzarlo a llevar a cabo el cambio de contexto inmediatamente. Esto último asume una capacidad especial de hardware para su realización.

El manejador del reloj de interrupciones en el procesador esclavo se asegura que el proceso sea replanificado periódicamente, de forma tal que ningún proceso *monopolice el procesador. A su vez, el reloj de interrupciones "despierta" al* procesador esclavo del estado de espera una vez por segundo. El procesador esclavo planifica la ejecución del proceso de más alta prioridad que necesite ejecutarse en un procesador maestro.

La única forma de corrupción de la estructura de datos del kernel puede darse en el algoritmo de planificación, debido a que el algoritmo no sería capaz de proteger la estructura de datos en el caso de que un proceso seleccionado para su ejecución se asigne a dos procesadores. En cambio, si la configuración del sistema contiene a un *maestro* y a dos *esclavos*, es posible que éstos dos procesadores *esclavos* encuentren

un proceso en modo usuario y *listo para ejecución*. Si ambos procesadores fueran a planificar el proceso simultáneamente, deberán leer, escribir y por lo tanto dañar o alterar su espacio de direcciones en memoria.

Este problema puede evitarse de dos formas posibles. Primero, el procesador maestro puede especificar el procesador esclavo en el cual el proceso deberá ejecutarse, permitiendo que más de un proceso sea asignado a un procesador. La consecuencia directa es el controlar el *balanceo de cargas*, ya que un procesador puede llegar a tener gran número de procesos asignados a él, mientras que otros pueden estar en estado de espera. El *kernel* maestro tendrá entonces que distribuir la carga de procesos entre los procesadores. En la segunda forma, el *kernel* puede permitir que *únicamente un procesador* ejecute el algoritmo de planificación, usando mecanismos tales como *semáforos*.

3.4 Solución con *Semáforos*.

Otro método para el uso de sistemas Unix en *configuraciones multiprocesador* es particionar el *kernel* en regiones críticas de forma que, a lo sumo, un procesador pueda ejecutar código en una región crítica a la vez mediante el uso de *semáforos*.

Existen 2 consideraciones importantes: cómo implementar los *semáforos* y dónde definir las regiones críticas.

Varios algoritmos en el diseño del sistema operativo Unix uniprocesador emplean una función, denominada *sleep-look*, que funciona como un candado que mantiene a los procesos fuera de una región crítica en el caso de que un proceso permanezca o se "duerma" dentro dicha región crítica.

El mecanismo para establecer este candado se ilustra en el algoritmo de la figura 3.4.

```
while (candado activo)
        sleep(condición hasta que el candado es liberado);
activar candado;
```

Figura 3.4: Mecanismo de Activación de Candado.

y el mecanismo para desactivar el candado es:

```
liberar candado;
despertar todos los procesos dormidos en la condición de candado activo;
```

Figura 3.5: Mecanismo de Desactivación de Candado.

Los candados *sleep-lock* delinean algunas regiones críticas, pero no funcionan en sistemas multiprocesador, como se muestra en la figura 3.6.

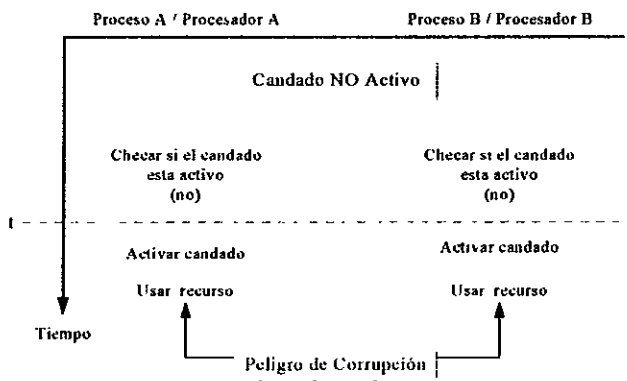

Figura 3.6: Condiciones de candados *sleep-lock* en Multiprocesadores.

Supóngase un candado libre y dos procesos en dos procesadores que, simultáneamente intentan activarlo. Tanto los procesadores como los procesos de este ejemplo encuentran que el candado está libre en el tiempo *t*, lo activan, entran en región crítica y pueden corromper la estructura de datos del *kernel*. Existe una constante en este requerimiento simultáneo: el candado *sleep-lock* falla si ninguno de los procesos ejecuta la operación de candado antes de que otro proceso ejecute la operación de prueba. Por ejemplo, si el procesador *A* maneja una interrupción después de encontrar que el candado está libre y, mientras se encuentra manejando la interrupción, el procesador *B* verifica el candado y lo activa, el procesador *A* regresará de la interrupción y también activará el candado. Para prevenir estas situaciones, se dice que los mecanismos para la activación del candado deben ser *atómicos*, es decir, las operaciones de prueba del status del candado y su activación deben ser hechas por una operación *única e indivisible*, de forma tal que sólo un proceso pueda manipular el candado a la vez.

3.4.1 Definición de Semáforo.

La figura 3.7 muestra el código en lenguaje C para la implementación de un *semáforo*. Un semáforo es un objeto valuado entero manipulado por el *kernel* que tiene definidas para ello las siguientes operaciones *atómicas*:

- Inicialización del semáforo en un valor no negativo.
- Una operación *P* que decrementa el valor del *semáforo*. Si el valor del semáforo es menor a 0 después de decrementar su valor, el proceso que hizo *P* se *duerme*.
- Una operación *V* que incrementa el valor del semáforo. Si el valor del semáforo se convierte mayor o igual a 0 como resultado, un proceso que se encuentre en estado *dormido* por medio del resultado de una operación *P despierta*.

```
struct semaforo
{
   int val[NUMPROCS];    /*candado: una entrada para cada procesador*/
   int lastid;    /*Identificador (ID) del último procesador que usó semáforo*/
};
int procid;                    /* ID de procesador, único para cada procesador */
int lastid;                    /* ID del último procesador que usó semáforo */
INIT(semaforo)
     struct semaforo semaforo;
{    int i;
     for (i=0; i<NUMPROCS; i++)
          semaforo.val[i]=0;
}
Pprim(semaforo)
     struct semaforo semaforo;
{    int i, first;
        loop:
                first=lastid;
                semaforo.val[procid]=1;
        forloop:
                for (i=first; i<NUMPROCS; i++)
                {       if (i == procid)
                        {       semaforo.val[i]=2;
                                for (i=1; i<NUMPROCS; i++)
                                   if (i!=procid && semaforo.val[i]==2)
                                        goto loop;
                                lastid=procid;
                                return;  /* operación exitosa, usar recurso */       }
                        else if (semaforo.val[i])
                                goto loop;                      }
                first=1;
                goto forloop;
}
Vprim(semaforo)
   struct semaforo semaforo;
   {    lastid = (procid + 1) % NUMPROCS;  /*elimina el siguiente procesador*/
        semaforo.val[procid]=0;          }
```

Figura 3.7: Implementación de un semáforo.

- Una operación condicional *P*, abreviada como *CP*, que decrementa el valor del semáforo y regresa un valor booleano de verdad o *true* si es mayor que 0. Si el valor del semáforo es menor o igual a 0, el valor del semáforo no es modificado y el valor regresado el falso o *false*.

3.4.2 Implementación de semáforos.

Es posible implementar semáforos sin necesidad de instrucciones especiales de máquina. La figura 3.7 presenta las funciones en C necesarias para implementar semáforos. La función *Pprim* coloca un candado al semáforo verificando los valores del arreglo *val*; cada procesador en el sistema controla una localidad en el arreglo. Cuando un proceso coloca un candado al semáforo, averigua si otros procesadores ya han hecho lo mismo (su localidad en *val* deberá ser 2), o bien, si algún procesador con un identificador ID menor esta actualmente tratando de colocar el candado al semáforo (su localidad en el arreglo *val* deberá ser 1). Si alguna condición es verdadera, el procesador asigna a su localidad en *val* un 1 e intenta nuevamente. *Pprim* inicia el primer ciclo con una variable igual al identificador ID del procesador mayor que aquel que utilizó un recurso, asegurándose que ningún procesador pueda monopolizar los recursos. La función *Vprim* libera al semáforo del candado y permite que otros procesadores obtengan acceso exclusivo a los recursos eliminando y limpiando la localidad correspondiente en *val* del procesador en ejecución y actualizando *lastid* El código de la figura 3.8 se usa para la protección de recursos.

```
Pprim(semáforo);
        Uso de recurso;
Vprim(semáforo);
```

Figura 3.8: Protección de Recursos.

La mayoría de los sistemas poseen un conjunto de instrucciones indivisibles que realizan la operación de bloqueo equivalente a la del candado, pero a menos costo, esto debido a que los ciclos *loop* en la función *Pprim* son lentos y pueden reducir el desempeño. Cuando este tipo de instrucciones son ejecutadas (como la de lectura/escritura), por ejemplo, el sistema lee el valor de una localidad de memoria, lo *limpia*, es decir lo actualiza a 0, y aplica el código de condicionamiento de acuerdo a si el valor original fue o no igual a 0. Si otro procesador utiliza esta instrucción de lectura/escritura simultáneamente sobre la misma localidad de memoria, uno de los procesadores tiene la garantía de estar leyendo el valor original y los otros procesadores leerán un valor de 0, por lo que el hardware asegura la *atomicidad*. Por lo tanto, la función *Pprim* puede ser implementada de forma más simple con una instrucción de *lectura y borrado*, como se muestra en la figura 3.9.

Un proceso con un *loop* que utilice una instrucción de *lectura/borrado*, realizará el ciclo hasta encontrarse un valor distinto a cero. El candado del semáforo debe ser inicializado con un valor de 1. Este es un semáforo primitivo que no puede ser utilizado en el *kernel*, debido a que un proceso que lo ejecuta se mantiene realizando *loops* sucesivos hasta que se cumpla alguna condición: si el semáforo esta siendo usado para colocar candado a una estructura de datos, un proceso debería *dormir* si encuentra el semáforo cerrado, de tal forma que el *kernel* puede cambiar de contexto hacia otro proceso y realizar trabajo útil

Dados *Vprim* y *Pprim*, es posible construir un conjunto de operaciones para el semáforo más sofisticadas.

```
Struct semaforo
{       int lock;
}
Init (semaforo)
        struct semaforo semaforo;
{       semaforo.lock = 1;
}
Pprim(semaforo)
        struct semaforo semaforo;
{       while (read_and_clear(semaforo.lock));
}
Vprim (semaforo)
        struct semaforo semaforo
{       semaforo.lock = 1;
}
```

Figura 3.9: Operaciones de semáforo usando instrucciones de *leer/limpiar.*

Primeramente, se define un semáforo como una estructura que contenga un *campo candado* que controle el acceso al semáforo mismo, el valor del semáforo y una cola de procesos *dormidos* en el semáforo.

El campo candado controlará el acceso al semáforo, permitiendo a sólo un proceso el manipular los otros campos de la estructura durante las operaciones *P* y *V* El semáforo se reinicializaría cuando las operaciones *P* y *V* se completen. El valor en el campo determina si un proceso debe tener acceso a una región crítica protegida por un semáforo.

En el inicio del algoritmo *P*, que se muestra en la figura 3.10, el *kernel* realiza una operación *Pprim* para asegurar el acceso exclusivo al semáforo y entonces decrementa el valor del semáforo. Si el valor del semáforo es no negativo, el proceso

en ejecución tiene acceso a la región crítica: se reinicializa el candado del semáforo con la operación *Vprim* de forma que los otros procesos pueden accesar al semáforo y regresar una valor boolenano de éxito (*true*). Si, como resultado de decrementar este valor, el valor es negativo, el *kernel* pone el proceso a *dormir*, siguiendo semánticas similares para aquellas que siguen el algoritmo regular que *duerme* procesos: se verifican las indicaciones de acuerdo al valor de prioridad, coloca el proceso ejecutable en la cola de procesos *dormidos* de forma PEPS (primero en entrar-primero en salir) y realiza el cambio de contexto.

La figura 3.11 muestra como la función *V* obtiene acceso exclusivo al semáforo por medio de la función *Pprim* e incrementa el valor del semáforo. Si algún proceso estuviera en la cola de procesos *dormidos*, el *kernel* elimina el primero y cambia su estado a "listo para ejecución".

Las funciones *P* y *V* son similares a las funciones para *dormir* y *despertar* procesos, la mayor diferencia en su implementación es que un semáforo es una estructura de datos, mientras que la dirección que se usa para *dormir* y *despertar* procesos es un simple valor numérico conveniente.

Un proceso siempre se *dormirá* cuando, al realizar una operación *P* en un semáforo, el valor inicial del semáforo es 0, por lo que *P* se puede reemplazar a la función *dormir*.

Sin embargo, la operación *V despierta* a un proceso únicamente, mientras que la función *despierta* de un sistema uniprocesador *despierta* a todos los procesos *dormidos* en un evento determinado por una dirección.

```
Algoritmo P          /* Operación de semáforo P */

entrada: (1) semáforo
         (2) prioridad
salida:  0 como salida normal
         -1 para una salida anormal debido a señales localizadas en el kernel
         saltos largos para señales no localizadas en el kernel
{
        Pprim(semaforo.lock);
        decrementar(semaforo.valor);
        if (semaforo.valor >= 0)
        {       Vprim(semaforo.lock);
                return(0)
        }
        if (verificar señales)
        {       if (existe una señal que interrumpa a un proceso dormido)
                {       incrementa(semaforo.valor);
                        if (señal localizada en kernel)
                        {       Vprim(semaforo.lock);
                                return(-1);
                        }
                        else
                        {       Vprim(semaforo.lock);
                                salto_largo;
                        }
                }
        }

        colocar en cola el proceso al final de la lista de procesos dormidos;
        Vprim(semaforo.lock);
        realizar el cambio de contexto;
        checar señales;
        return(0);
}
```

Figura 3.10: Algoritmo para la implementación de *P*.

```
Algoritmo V          /* Operación de semáforo V */
entrada: dirección del semáforo
salida: ninguna
{
        Pprim(semaforo.lock);
        incrementar (semaforo.valor);
        if (semaforo.valor <= 0)
        {
                eliminar primer proceso de la lista de procesos dormidos;
                hacer el proceso listo para ejecución (despertarlo);
        }
        Vprim(semaforo.lock);
}
```

Figura 3.11: Algoritmo para la implementación de *V*.

Semánticamente, el uso de funciones para *despertar* procesos indica que con una condición de sistema dada que no sea verdadera por un breve espacio de tiempo, se obtiene por resultado que todos los procesos que estuvieron *dormidos* por esa condición deberán despertar. Por ejemplo, cuando un *buffer* exista por un corto tiempo, es incorrecto que los procesos se *duerman* cuando el *buffer* está ocupado, por lo que el kernel *despierta* a todos aquellos procesos que estuvieran *dormidos*. Un segundo ejemplo, si múltiples procesos escriben datos en una terminal, el controlador o *driver* de dicha terminal puede *dormir* los procesos debido a que no le es posible manejar un alto volumen de datos. Después, cuando el controlador de la terminal decide que puede aceptar más datos, *despierta* todos los procesos que *durmió*, esperando datos de salida. El uso de las operaciones *P* y *V* es más aplicable para operaciones de cierre con candados donde los procesos obtienen acceso a un recurso

uno por uno y otros procesos ceden el acceso a determinados recursos de acuerdo a la forma en que estos son requeridos. Esto es usualmente más eficiente en sistemas uniprocesador, debido a que si todos los procesos *despiertan* concurrentemente a un solo evento determinado, la mayoría de ellos puede encontrar un condición de cierre por candado e inmediatamente *dormir* nuevamente. Por otro lado, es más complejo usar *P* y *V* para los casos donde todos los procesos deben *despertar* a la vez.

A este punto, se hace necesario considerar otro fenómeno que ocurre con el uso de semáforos en sistemas uniprocesador. Supóngase que dos procesos, *A* y *B*, compiten por un semáforo: el proceso *A* encuentra libre el semáforo y el proceso *B* se *duerme*; entonces el valor del semáforo es -1. Cuando el proceso *A* libera el semáforo mediante la función *V*, esta misma *despierta* el proceso *B* e incrementa el valor del semáforo a 0.

Ahora supóngase un proceso *A*, aún ejecutándose en modo *kernel*, tratando de cerrar con un candado el semáforo, una vez más *dormirá* mediante la función *P*, debido a que el valor de el semáforo es 0, aún y cuando el recurso este libre. El sistema incurrirá en un gasto extra al cambiar de contexto.

Por otro lado, si el candado fuera implementado de forma *sleep-lock*, el proceso *A* obtendrá acceso inmediato del recurso otra vez, ya que ningún otro proceso puede hacerlo. En este caso, el cierre *sleep-lock* puede resultar más eficiente que un semáforo. Cuando se cierran con candados varios semáforos, el orden de cierre debe ser consistente para evitar una falla fatal. Por ejemplo, considerence 2 semáforos, *A* y *B*, y considerence 2 algoritmos para *kernel* que deben tener a ambos semáforos con candados. Si los dos algoritmos colocaran los candados en orden inverso, podría surgir una falla como la que se muestra en la figura 3.12.

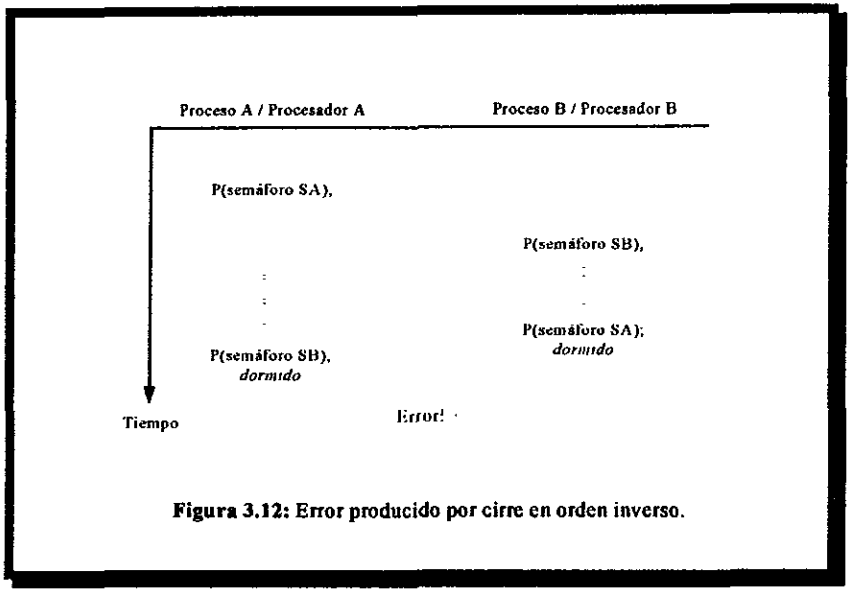

Figura 3.12: Error producido por cirre en orden inverso.

El proceso *A* en el procesador *A* cierra el semáforo *SA* mientras que el proceso *B* en el procesador *B* cierra el semáforo *SB*. El proceso *A* intenta cerrar el semáforo *SB*, pero la operación *P* causa que el proceso *A* se *duerma*, debido a que el valor de *SB* es cuando mucho igual a 0. De forma similar, el proceso *B* intenta cerrar el semáforo *SA*, pero su operación *P* coloca el proceso *B dormido*. Por lo tanto, ningún proceso puede proceder a ejecutarse.

Este tipo de errores, conocidos como *interbloqueos* o *deadlocks*, pueden evitarse implementando un algoritmo de detección de estos errores que determine su ocurrencia, si sucede, el algoritmo invalida la condición que produce el error. Sin embargo, la implementación de este tipo de algoritmos puede complicar el código del *kernel*. Dado que existe sólo un número finito de lugares en el *kernel* donde un proceso debe cerrar a varios semáforos de manera simultánea, se facilita la implementación de algoritmos de *kernel* para evitar condiciones de interbloqueo antes de que ocurran. Por

ejemplo, si un conjunto particular de semáforos estuvieran siempre cerrados en el mismo orden, la condición de interbloqueo nunca ocurriría. Pero si es imposible evitar que los semáforos se cierren en orden inverso, la operación *CP* previene la condición de error, como se muestra en la figura 3.13.

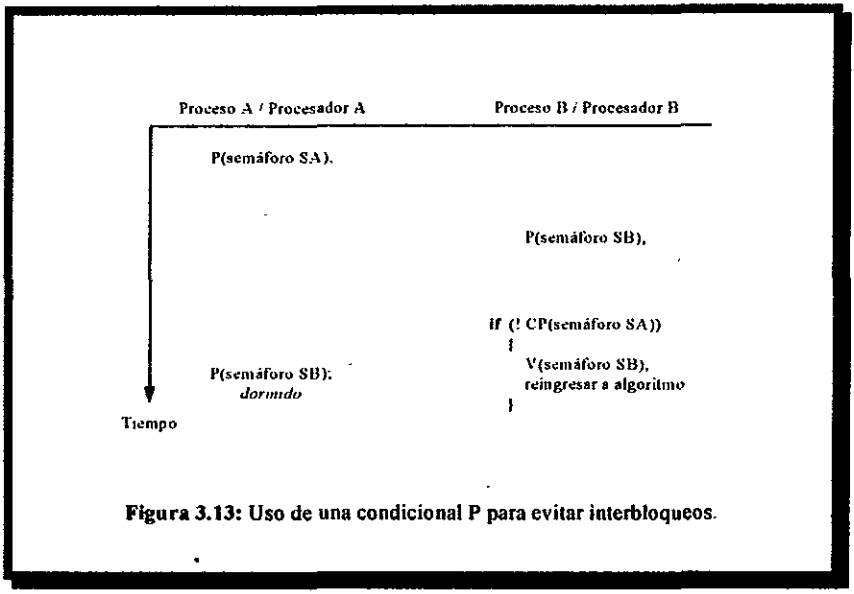

Figura 3.13: Uso de una condicional P para evitar interbloqueos.

Si la operación *CP* falla, el proceso *B* libera sus recursos para evitar el interbloqueo y reingresa en el algoritmo tiempo después, presumiblemente cuando el proceso *A* completa su uso de recursos. Un manejador de interrupción podría cerrar un semáforo para prevenir que varios procesos utilicen los mismos recursos de forma simultánea, pero no los puede *dormir*, por lo tanto no puede hacer uso de la operación *P*. En cambio, puede ejecutar un candado especial, llamado generalmente *spin*, para evitar que cambiar a *dormido*:

```
while(!CP(semáforo));
```

La operación realiza ciclos hasta que el valor del semáforo es menor o igual a 0; el manejador de interrupción no *duerme*, y el ciclo termina únicamente cuando el valor del semáforo se convierte positivo, que es cuando *CP* decrementa el valor del semáforo. Para evitar la condición de interbloqueo, el kernel deberá esbozar interrupciones que ejecuten un candado *spin*, de otro modo, un proceso puede cerrar un semáforo y ser interrumpido antes de abrir el mismo semáforo; si el manejador de interrupciones intenta cerrar el mismo semáforo usando un candado *spin*, el kernel sufre un interbloqueo. En la figura 3 14, por ejemplo, el valor del semáforo es, cuando mucho, igual a 0 cuando la interrupción ocurre, por lo que *CP* en el manejador de interrupción siempre será falso. La situación se evita esbozando interrupciones mientras el proceso tiene al semáforo cerrado.

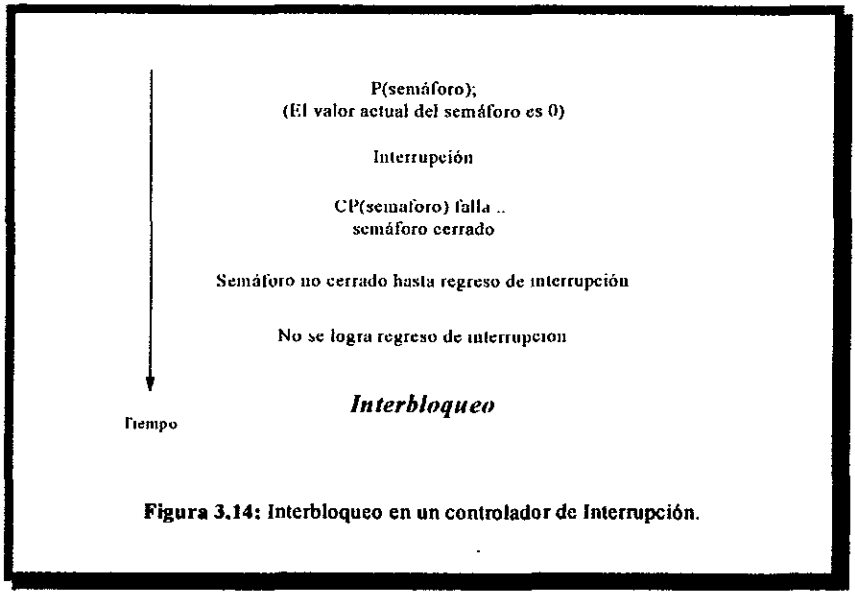

Figura 3.14: Interbloqueo en un controlador de Interrupción.

El uso de semáforos en la implementación de sistemas multiprocesador, como se ha descrito, permite que todos los procesadores del sistema se ejecuten en modo *kernel* de forma simultánea, a diferencia de una implementación con el uso de una configuración *maestro/esclavo* en la que sólo se permite a un procesador ejecutarse en modo *kernel*. Sin embargo, aunque estas implementaciones pueden generalizar a cualquier número de procesadores, es posible que el sistema no incremente su productividad en forma *lineal* de acuerdo al número de procesadores que posea.

Existen dos aspectos que se deben considerar: en primer lugar, existe *degradación* debido a que los accesos a memoria se vuelven más lentos y, en segundo lugar, al implementar semáforos, los procesos frecuentemente los encontrarán cerrados, lo que provoca que existan más procesos en la cola de espera de semáforos libres, y por lo tanto más procesos se mantienen en espera para accesar a un semáforo. De forma similar, en la configuración *maestro/esclavo* se puede generar un *cuello de botella* si el número de procesadores crece demasiado, ya que sólo un procesador puede ejecutar el código del modo kernel.

Al realizar un balance de estos conflictos, surge el siguiente caso de solución al problema de concurrencia de acceso a recursos del sistema, que propone el uso del modelo simétrico para evitarlos.

3.5 Solución con Organización simétrica.

Para reducir las limitaciones de desempeño del sistema que se obtienen con la implementación de *semáforos*, es importante determinar la arquitectura para el sistema multiprocesador. Como se ha visto, dentro de la organización MIMD (*Multiple Instruction Stream-Multiple Data Stream)* se encuentran la mayoría de los sistemas multiprocesadores. Para coordinar el uso de múltiples procesadores, los sistemas con

organización MIMD se pueden manejar de dos formas: *multicomputadoras* con planificación coordinada y multiprocesadores *verdaderos*, que basa su funcionamiento en el uso de algoritmos de planificación.

Los sistemas que utilizan planificación coordinada manipulan a cada procesador del sistema como una computadora individual, con sus propio sistemas de interrupciones y almacenamiento. Cada procesador posee una copia del kernel y recibe las cargas de procesos de un procesador administrador. Esta arquitectura permite la existencia de varios procesadores ejecutando programas de aplicación, sin embargo, una vez que un proceso es asignado a un procesador, este lo retiene hasta terminar con su ejecución. Esta arquitectura brinda varios beneficios, la naturaleza multiprocesador del sistema es transparente al usuario, además suelen ser sistemas ligeramente acoplados, lo que los hace relativamente fáciles de implementar. La principal desventaja es la ineficiencia y el uso desbalanceado de los recursos, además de que los sistemas ligeramente acoplados provocan conflictos en la implementación de aplicaciones de programación en paralelo.

Los multiprocesadores *verdaderos* o estrechamente acoplados se caracterizan por permitir al conjunto de procesadores accesar a todos los recursos del sistema, como memoria compartida, dispositivos de E/S y subsistemas de interrupciones. El algoritmo planificador define la distribución de procesos haciendo al sistema *simétrico*, con lo que cualquier procesador puede ejecutar cualquier proceso, ya sea en modo kernel o en modo usuario, lo que permite que el número de procesadores en el sistema puede crecer sin sufrir la *degradación* que produciría una implementación *maestro/esclavo*.

La principal desventaja de esta organización es su implementación, ya que solo existe una copia compartida del kernel y los mecanismos para la exclusión mutua y la sincronización deben ser lo suficientemente rápidos para minimizar colisiones que podrían limitar el desempeño de la arquitectura. Sin embargo, ya se ha discutido cómo la implementación de *semáforos* minimizan las regiones críticas donde el acceso simultáneo debe ser controlado y se debe asegurar su sincronización. Mediante la implementación de semáforos en una organización simétrica se minimiza el acceso concurrente a regiones críticas, ya que sólo existe una copia compartida del código del modo kernel. Una vez más, el hardware permite a todos los procesadores el acceso a los recursos del sistema y al ejecutarse el código del modo kernel se logra la planificación de los recursos a los procesadores de forma dinámica y no hacia procesadores particulares. Cuando se dice que el sistema es totalmente simétrico, significa que cualquier procesador puede ejecutarse en modo kernel. De esta forma, el algoritmo planificador asigna los procesos que se ubican en un conjunto homogéneo y común.

La cola de procesos ejecutables en espera se reduce, ya que el algoritmo planificador puede elegir el siguiente proceso a ejecutarse cuando el proceso en ejecución actual agota su tiempo o bien espera por alguna interrupción de entrada/salida. Al existir sólo una cola de procesos *dormidos*, es posible eficientar el que varios procesadores atiendan a una misma cola compartida de procesos, ya sea en espera o *dormidos*. El algoritmo planificador puede ser capaz de ampliar el tiempo del proceso en cada procesador, dando la impresión de ser una ejecución continua, con lo que se reduce el número de cambios de contexto. La figura 3.15 muestra el esquema de la organización simétrica de esta arquitectura. Cada procesador puede ejecutar el

kernel para el desarrollo de las aplicaciones de los usuarios. Así, cualquier trabajo que necesita ser realizado por el sistema puede ejecutarse en cualquiera de los procesadores. Todos los procesadores comparten el kernel y pueden hacerlo de forma simultánea. En particular, las solicitudes de E/S y las interrupciones de controladores de hardware pueden ser procesadas en paralelo en varios procesadores a la vez.

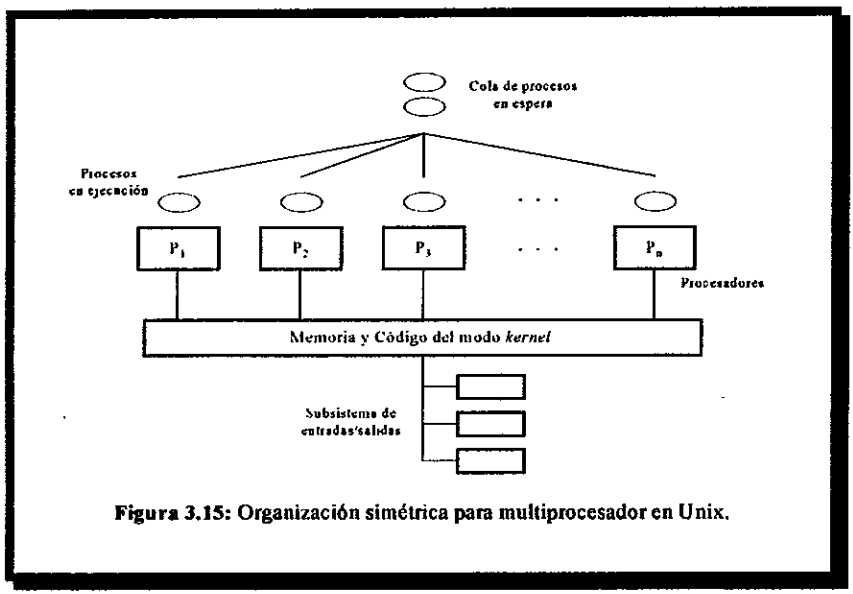

Figura 3.15: Organización simétrica para multiprocesador en Unix.

Un proceso que se ejecute en esta implementación simétrica puede encontrarse en diferentes ocasiones por cualquiera de los procesadores existentes, de acuerdo a la asignación del algoritmo planificador. La contención de procesos es resuelta por el diseño en estructuras de datos de los semáforos y las operaciones *P* y *V* que evitan la excesiva exclusión por bloqueo que suelen provocar los continuos accesos concurrentes a recursos.

Al tomar los beneficios del uso de semáforos en ambiente Unix en la implementación simétrica para el manejo de regiones críticas, se incrementan las ventajas de su uso: el sistema es totalmente simétrico, por lo que ningún procesador se apropia del código del modo kernel, sino que todos los procesadores, cooperando como un equipo, pueden compartirlo bajo cualquier condición de multiprogramación a aplicaciones multitarea. Como se mencionó, Unix normalmente soporta usuarios múltiples o tareas múltiples, dando la impresión de que cada proceso posee su propio procesador, pero en realidad existe solo un procesador que se mueve de un proceso a otro varias veces por segundo. Con la utilización de varios procesadores en este tipo de implementaciones simétricas, un sistema Unix permite a varios procesadores dar servicio a una cola de procesos única y compartida, con lo que puede incrementarse la cantidad de tiempo que cada proceso se ejecuta en un procesador, obteniendo como resultado una reducción en el número de cambios de contexto y una optimización en la utilización de recursos en una arquitectura multiprocesador.

CAPITULO 4
EJECUCIÓN DE APLICACIONES PARALELAS.

La continua expansión de los campos de aplicación de los sistemas de cómputo, unida al incremento de su potencia de cálculo y a la simplificación de la programación, obligan a una mejora constante de sus presentaciones. Para la construcción de sistemas capaces de soportar la explosiva demanda en la potencia de procesamiento, existen diversas alternativas, confluyendo todas ellas en la necesidad de desarrollar el *paralelismo* a todos los niveles.

Por otro lado, el uso de sistemas cada vez más potentes es requerido para el desarrollo y la investigación en diversas áreas del conocimiento humano y las ciencias, para que ayuden a la solución de problemas a gran escala. Sin el empleo de sistemas de cómputo suficientemente potentes, muchos de estos problemas difícilmente podrían ser solucionados o siquiera investigados. En la actualidad, ha tenido un gran auge el desarrollo de una teoría que es un claro ejemplo de tal afirmación: la teoría *fractal*

4.1 Ejemplo de aplicación en sistemas multiprocesador.

La palabra *geometría* etimológicamente significa *medir la Tierra*. En particular, la geometría Euclidiana *mide* la Tierra usando ángulos y longitudes describiéndola en términos de puntos, líneas rectas, círculos, rectángulos, triángulos, cubos y esferas. Sin embargo, esta descripción Euclidiana generalmente no suele concordar con la realidad. Supóngase que se emplea una regla de longitud t para hacer la medición del perímetro de un terreno, y que esta regla se superpone a lo largo de la curva que describe el

perímetro, entonces la longitud *L* del *perímetro sería* igual al producto de la cantidad de veces que se superpone la regla por *t*, como se indica en la siguiente expresión:

$$L(t) = nt$$

donde n es el número de veces que la regla es superpuesta.

Si se disminuye la longitud de la regla, entonces el valor de n se incrementa, lo que a su vez resulta en un incremento de $L(t)$, esto debido a que entre más pequeña sea la longitud de la regla que se utilice, entonces existe más distancia a medir.

Con el ejemplo anterior se puede deducir que las distancias entre dos puntos son relativas a la escala y al detalle de la observación. Por otro lado, existe una diferencia fundamental entre una curva como un círculo y una curva como el perímetro de un país. Esta diferencia separa los objetos de la geometría Euclidiana de los objetos de una geometría más compleja, conocida como *geometría fractal* y a cuyos elementos se les conoce como *fractales*.

4.1.1 Definición de *fractal*.

El enunciado siguiente es una definición de *fractal* intuitiva y acorde al ejemplo antes mencionado:

Si la longitud estimada de una curva crece arbitrariamente cuando la unidad de medida es cada vez más pequeña, entonces la curva se llama curva fractal o simplemente fractal.

Esta definición se puede generalizar para comprender otro tipo de figuras diferentes a las curvas, por ejemplo, superficies o volúmenes. En cualquier caso, la idea básica es la misma: la dificultad de obtener una medición es debida a la irregularidad del objeto a ser medido, y esta es una irregularidad que continua hasta niveles microscópicos.

En cierta forma la palabra irregular y la palabra fractal están estrechamente relacionadas. El matemático francés Benoit Mandelbrot, considerado como el "Padre de los Fractales", explicó que inventó la palabra *fractal* tomando como base el adjetivo en latín *fractus*, y su correspondiente verbo en latín *frangere* que significa *romper*, es decir, crear fragmentos irregulares, por lo tanto *fractus* debe también significar *irregular*.

4.1.2 Dimensiones Fractales.

La noción de espacios con dimensiones $D = 1, 2, ..., n$, es consecuencia directa de la geometría Euclidiana, por lo que a estos espacios se les conoce como *espacios Euclidianos*. Si se considera un objeto de un espacio Euclidiano de dimensión D, y este objeto se subdivide en N copias iguales siendo la razón de subdivisión r, entonces existe una ecuación que relaciona a D, N y r, la cual está dada por $Nr^D=1$, o equivalentemente

$$[1]\ N = 1 / r^D ;\ D = 1, 2, ..., n.$$

De esta forma, para una dimensión D dada, el número N de copias iguales depende de la razón r, es decir, N es una función de r: $N=N(r)$.

Considérese el proceso iterativo de agregar un triángulo a la mitad de una línea recta, como lo muestra la figura 4.1.

Teóricamente, las iteraciones sobre cada línea recta se pueden llevar a cabo indefinidamente. Es claro observar en este proceso que cada línea se transforma en cuatro líneas cada una de las cuales tiene una longitud de 1/3 de la original.

La curva que se obtiene, conocida como "Isla de Koch", es resultado de un proceso recursivo o iterativo que puede ser implementado mediante un lenguaje de programación.

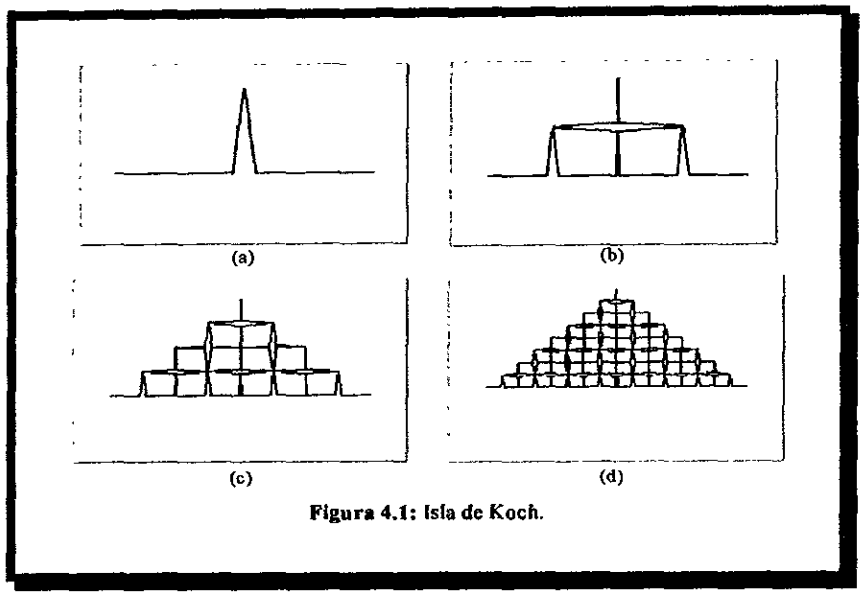

Figura 4.1: Isla de Koch.

Esta curva, de acuerdo a la definición, es un *fractal*, ya que no queda definida claramente la dimensión a la que pertenece la curva. De la ecuación **[1]** se obtiene que, aplicando logaritmos naturales, la dimensión D esta relacionada con N y r a través de la expresión

$$[2]\quad D = \ln(N) / \ln(1/r)$$

donde $\ln(x)$ denota la función logaritmo natural (de base e=2.718281..) de x. En el caso de la Isla de Koch se sabe que N=4 y r=1/3, entonces sustituyendo estos valores en la ecuación **[2]** se obtiene

$$D = \ln(4) / \ln(3) = 1.261859...$$

Este es un resultado incongruente con la geometría Euclidiana En primera instancia se podría pensar que la Isla de Koch es una curva que se encuentra aproximadamente a un cuarto de distancia entre una línea y un plano. El número D que

se obtiene de la ecuación [2] es conocido como *dimensión fractal*. La dimensión fractal de un objeto se puede interpretar como una medida del grado de irregularidad de un fractal a cualquier nivel de escala y puede ser una cantidad fraccionaria mayor que la dimensión Euclidiana de un objeto. Entre más grande sea la dimensión fractal de un objeto, este objeto es más irregular y la medida estimada para el objeto crece más rápidamente. Para objetos inmersos en espacios Euclidianos, la dimensión del objeto y su dimensión fractal son exactamente iguales. De esta forma, un fractal es un objeto que tiene una dimensión fractal que es mayor a su dimensión Euclidiana.

4.1.3 Los *fractales* y el Mundo real.

Aunque parezca incierto, el mundo está compuesto por fractales y la humanidad ha estado en contacto con ellos de alguna forma. Por ejemplo, una montaña es un fractal, ya que su rugosidad es la misma a diferentes escalas. Esta rugosidad es tal, que la dimensión fractal de una montaña es la misma a cualquier escala, incluyendo la escala humana. Por otro lado, la nubes son fractales, ya que su rugosidad es totalmente irregular y cambian de forma continuamente. La apariencia de su suavidad es producto de la mezcla de los colores que reflejan.

La superficie del agua en el océano o en un lago es también un fractal. En esta superficie se pueden observar infinidad de pequeñas olas y entre ellas existen zonas completamente suaves. La superficie del agua es demasiado compleja y por lo tanto es difícil de describir. Esta superficie contiene zonas suaves y rugosas que están presentes a cualquier nivel de escalamiento. Debido a esta complejidad es difícil determinar su dimensión fractal. Otros ejemplos pueden visualizarse entre el reino vegetal. En general, los árboles, arbustos y flores se desarrollan con un cierto patrón de crecimiento que los hacen pertenecer a la geometría fractal. Por ejemplo, cada una de

las ramas de un árbol es una miniaturización del árbol mismo, aunque su grado de autosimilaridad no sea el mismo. En el organismo humano también es posible reconocer fractales. Las arterias y las venas están conectadas por una red de vasos diminutos que se dividen en ramas pequeñas y que a su vez se subdividen en ramas más pequeñas hasta llegar a convertirse finalmente en capilares microscópicos. Esta geometría del sistema circulatorio es típica de los fractales. En efecto, esta separación en ramas y subramas tiene la misma forma cuando se examinan en detalle a escalas más pequeñas.

Como estos ejemplos, existe una infinidad que se pueden mencionar. En la actualidad, esto se ha convertido en tema de investigación de muchas ramas de la ciencia, sin embargo, muchos investigadores están convencidos de que es posible modelar patrones autosimilares de objetos de la naturaleza, tales como las montañas, con el uso de expresiones y fórmulas matemáticas relativamente sencillas. Esto se debe a que los patrones a escalas pequeñas se repiten a grandes escalas como consecuencia del uso de fórmulas iterativas.

4.1.4 Fractales por computadora.

El avance tecnológico de los actuales sistemas de cómputo y de la técnicas de recursividad en programación, han permitido interactuar con expresiones de las dimensiones fraccionarias, con lo que se han facilitado el describir puntos en una dimensión fractal. Normalmente en matemáticas, cuando se manejan ecuaciones muy complejas, se dice que es necesario realizar un análisis, tanto con ecuaciones más simples como con las leyes fundamentales de las matemáticas básicas. Lo mismo sucede con la dimensión fractal, cuando se contempla un punto a simple vista en una dimensión fractal, es muy probable que no muestre algo interesante, pero si se

amplifica ese punto cada vez más, entonces se puede notar que no sólo se trata de un simple punto, sino que se trata de un dibujo que aparenta ser sólo un punto insignificante.

Esto se puede lograr gracias al avance tecnológico que se ha desarrollado en los últimos años en materia velocidad de procesamiento y capacidad de almacenamiento de los sistemas de computo. Hoy en día se sabe que un fractal, además de ser un dibujo con cierta armonía, es también la evaluación de una ecuación que se encuentra vinculada con la naturaleza. Se puede decir que un fractal posee las siguientes características:

- Se rige por una ecuación.
- Utiliza las técnicas de recursividad.
- Un dibujo fractal es la evaluación repetitiva de una ecuación y la combinación de procesos matemáticos
- Es el resultado de obtener un acercamiento con los procesos que ocurren en la naturaleza
- Su objetivo principal es ayudar al hombre a encontrar la ecuación que rige a la naturaleza (hoy en día es lo que muchos matemáticos esperan).

En las siguientes secciones se estudiarán a dos de los más conocidos fractales: el Conjunto Julia y el Conjunto de Mandelbrot. Se analizará su representación matemática y se desarrollarán los algoritmos y programas para poder generarlos. Finalmente, se realizarán pruebas y evaluaciones de ejecución de los programas en diferentes organizaciones de sistemas de cómputo, en donde se resaltará la forma en que la tecnología de los sistemas multiprocesadores ha servido como una herramienta muy poderosa para el desarrollo de las investigaciones sobre fractales.

4.2 El Conjunto Julia.

Desde que la multiplicación de números complejos tuvo sentido, ha sido posible considerar a la ecuación cuadrática como una función compleja. Considérese la función $T(z) = z^2$, donde $z = x + iy$, es decir, un *número complejo*. En términos de las partes real e imaginaria de z, la función T está dada por

$$T(x + iy) = x^2 - y^2 + i\,(2xy)$$

Así, la parte real de $T(x + iy)$ es $x^2 - y^2$, y la parte imaginaria es $2xy$. Cuando se aplica la función T a un número complejo, se obtiene un nuevo número complejo, denominado z^2. La *órbita* de cualquier número complejo bajo esta función es una colección de puntos en el plano complejo más que en la línea real. Es posible graficar tales puntos, recordando que existen dos coordenadas a graficar y no únicamente una.

El *Conjunto Julia* de una función compleja es llamado así en honor al matemático francés Gaston Julia, quien descubrió muchas de las propiedades básicas del conjunto que lleva su nombre. Una definición precisa del conjunto Julia de un polinomio es que se trata de *el límite del conjunto de puntos que escapan hacia el infinito*. Esto significa que un punto en el conjunto Julia tiene una órbita que no escapa hacia el infinito, pero arbitrariamente cerca, existen puntos cuya órbita si escapa.

4.2.1 Órbitas de Escape.

Para estudiar la geometría del Conjunto Julia, se tomarán como base funciones cuadráticas de la forma

$$Q_c(z) = z^2 + c$$

donde c es un parámetro complejo, es decir, $c = c_1 + ic_2$ donde c_1 y c_2 son números reales. Aquí se presentará un algoritmo para calcular el conjunto Julia de la función Q_c que funciona correctamente cuando Q_c tiene una órbita de atracción periódica

También se mostrará un algoritmo que funciona mejor cuando no existe un ciclo de atracción presente. Cuando se dice que la función Q_c tiene una órbita de atracción periódica, significa que existe una colección de puntos que serán atraídos hacia un ciclo. Estos son puntos que se encuentran en lo que se conoce como *el conjunto de atracción del ciclo.* Como ya se mencionó, para la función $T(z) = z^2$ existen puntos cuyas órbitas tienden al infinito. Los puntos que se encuentran en el límite entre el conjunto de atracción y los puntos de escape, forman el conjunto llamado *Conjunto Julia.* El primer método de graficación del Conjunto Julia será coloreando de azul oscuro aquellos puntos cuyas órbitas escapan y coloreando de azul aquellos puntos cuyas órbitas no lo hacen. El límite entre esas dos regiones será entonces el Conjunto Julia

Primeramente, se debe determinar si una órbita escapa o tiende a infinito. Para ecuaciones cuadráticas de la forma $z^2 + c$ con $|c| \leq 2$ es muy sencillo: si cualquier punto en la órbita de z_0 se encuentra fuera del circulo de radio 2, entonces la órbita entera escapa hacia el infinito. Así, para comprobar si la órbita de z_0 escapa, sólo se debe verificar si algún punto en la órbita tiene *módulo* -la distancia del punto z al origen- mayor que 2; si es así, entonces z_0 se encuentra en una órbita de escape. Para comprobarlo, se debe recordar que sólo se consideran valores para c donde $|c| \leq 2$, si cualquier punto en la órbita tiene módulo mayor a 2, entonces tal órbita a la larga tenderá al infinito.

Este hecho puede aprovecharse para generar un algoritmo. El programa en lenguaje de programación C que se muestra en la figura 4.2 acepta como entrada un parámetro complejo $c = c_1 + ic_2$, después despliega en color azul el conjunto de puntos que se encuentran incluidos dentro de la región $|x|, |y| \leq 2$ cuya órbita no ha

escapado más allá del circulo de radio 2 centrado en el origen antes de la iteración número 20 de Q_c.

Es importante recordar que esta prueba sólo funciona para $Q_c(z) = z^2 + c$ cuando $|c| \leq 2$.

4.2.2 Algoritmo Julia-1.

El algoritmo usado para producir el conjunto Julia de Q_c está dado por los siguientes pasos:

1. Leer los valores para c_1 y c_2.
2. Seleccionar un área de *400x400* en el plano.
3. Para cada punto z_0 en esta área, calcular los primeros 20 puntos en la órbita de z_0. Verificar en cada iteración si el punto correspondiente se encuentra fuera del círculo de radio 2.
4. Si algún punto en la órbita se encuentra fuera del círculo de radio 2, entonces dejar de iterar y desplegar el punto original z_0 en negro.
5. Si los 20 puntos en la órbita de z_0 se encuentran dentro del circulo de radio 2, entonces desplegar el punto original de z_0 en azul.

La figura 4.2 muestra el código en lenguaje C necesario para ejecutar este algoritmo. Este programa no produce una salida inmediata, por el contrario, puede consumir una cantidad considerable de tiempo de ejecución, dependiendo de las características del equipo que se use.

Son calculadas más de 20 iteraciones de Q_c sobre cada punto en un área de 400x400 Esto significa que se deben calcular más de 1 millón y medio de iteraciones de Q_c para poder dibujar el Conjunto Julia.

```
scanf("%f",&c1);
scanf("%f",&c2);
for(m=0;m<=400;m++)
        {
         x0=-1+m/100;
         for(n=0;n<=200;n++)
                {
                 y0=1-n/100;
                 x=x0;
                 y=y0;
                 for(i=1;i<=20;i++)
                        {
                         x1=x*x-y*y+c1;
                         y1=2*x*y+c2;
                         x=x1;
                         y=y1;
                         z=x*x+y*y;
                         if (z>4) break;
                        }
                 if (z>4) continue;
                 putpixel(200+m,100+n,11);
                 putpixel(400-m,300-n,11);
                }
        }
```

Figura 4.2: Programa Julia-1.

Para obtener la gráfica se deben transformar las coordenadas del plano complejo *xy* a coordenadas de la pantalla de un monitor. La transformación esta dada por las sentencias

$$x = -1 + m / 100$$

$$y = 1 - n / 100$$

Aquí, las coordenadas de la pantalla están dadas por (m,n). Es importante tomar en cuenta que $1 \leq m, n \leq 400$, mientras que $-2 \leq x, y \leq 2$.

Para cada punto (m,n) en la pantalla, la transformación anterior elige un número complejo de la forma $x + iy$ cuya órbita será evaluada. La evaluación la efectúa la sentencia *if* donde se verifica si $|z|^2 > 4$. Resulta más rápido evaluar si $|z|^2 > 4$ que invocar a la función raíz cuadrada y después evaluar si $|z| > 2$.

Hay que notar que existen 3 ciclos anidados en el programa, que iteran a lo largo de las coordenadas de la pantalla. El primer ciclo itera sobre las coordenadas *m* seleccionando en cada ciclo la coordenada *n*.

En el programa, la coordenada *n* es evaluada sólo desde 0 a 200, debido a que se explota la simetría en éste sistema dinámico: después de una iteración, las órbitas de *z* y *-z son exactamente* iguales, ya que $Q_c(z) = Q_c(-z)$. Así, si se ha determinado que la órbita de *z* escapa, entonces también lo hará la órbita de *-z*. Por lo tanto, si se realiza la evaluación en el punto (200+m,100+n), también se obtendrá la evaluación para el punto (400-m, 300-n) que corresponde a *-z*.

Si se utiliza el programa Julia-1 para un valor de $c = 0$, la salida deberá ser un disco centrado en las coordinadas de la pantalla (200,200), con un radio de 100 pixeles. Sin embargo, utilizando el programa para generar el conjunto de puntos cuya órbita no escapa bajo una iteración de Q_c para diversos valores de *c*, *se obtienen* resultados como los que muestra la figura 4.3.

Cabe mencionar que las gráficas mostradas en la figura 4.3 fueron vistas por vez primera a principios de los años ochenta Es cierto que los matemáticos de entonces conocían con certeza la forma del Conjunto Julia (como por ejemplo, el Conjunto Julia de una ecuación cuadrática simple), pero también es cierto que nadie realmente entendía cuan diferentes y complicadas podrían ser esas figuras dependiendo del valor del parámetro *c*.

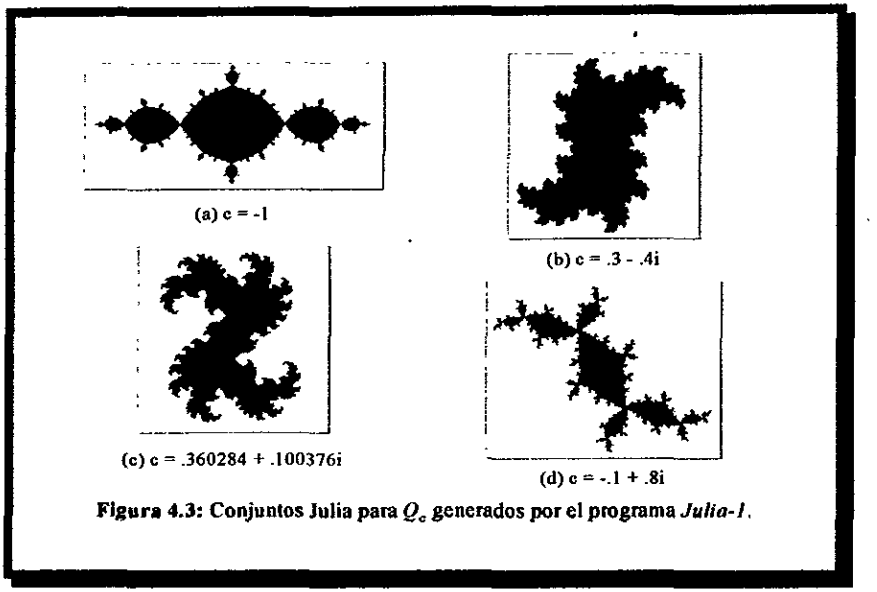

Figura 4.3: Conjuntos Julia para Q_c generados por el programa *Julia-1*.

Como ya se mencionó, el Conjunto Julia esta formado sólo por el límite (o borde) de las regiones azul oscuro que se muestran en la figura 4.3. Las regiones azul oscuro que se observan suelen denominarse como el *relleno* del Conjunto Julia. Al agrandar porciones del Conjunto Julia se revelan más detalles que los que se observan a simple vista. Estos detalles son muy semejantes a la figura original. Como ya se mencionó, estas características de autosimilaridad en el Conjunto Julia son propiedades de los *fractales*. En el Apéndice se muestra el código fuente completo del programa Julia-1 para compilarse en el Borland Turbo C y producir el archivo ejecutable correspondiente.

4.2.3 Polvo fractal.

Existen varios métodos alternativos para desplegar el Conjunto Julia de una función polinomial además del anterior. Uno de ellos es conocido con el nombre de

Método de iteración atrasada. Este método tiene la ventaja de ser más rápido que el método anterior, aunque la figura que despliega en ocasiones es menos precisa. Otras ventajas incluyen el hecho de que produce una imagen real del Conjunto Julia, conocida con el nombre de *polvo fractal*, a diferencia del método anterior que lo mostraba incluso con su *relleno*.

Para el algoritmo anterior, se calculaba la órbita de un número complejo z mediante la iteración de la función cuadrática $Q_c(z) = z^2 + c$. Para hacer esto, se requería saber cómo se eleva al cuadrado un número complejo. Ahora, para calcular una *órbita atrasada*, se requiere sabe cómo anular esta operación, es decir, calcular la raíz cuadrada de un número complejo.

Como en el caso de los números reales, cada número complejo posee dos raíces cuadradas. Supóngase el número complejo $z = r \cos \theta + ir \operatorname{sen} \theta$, entonces una de las raíces cuadradas de z tiene módulo $\sqrt{r}$ y ángulo polar $\theta / 2$, así que

$$\sqrt{z} = \sqrt{r} \cos(\theta / 2) + i \sqrt{r} \operatorname{sen}(\theta / 2)$$

Al igual que en el caso de los reales, la otra raíz cuadrada de z es simplemente el negativo de este número complejo. Así, dado un número complejo z, se pueden encontrar sus dos raíces cuadradas. Primero se necesita calcular su representación polar para obtener el módulo r y el ángulo polar θ de z. Las dos raíces cuadradas de z estarán dadas por

$$w_1 = \sqrt{r} \cos(\theta / 2) + i \sqrt{r} \operatorname{sen}(\theta / 2)$$

$$w_2 = -\sqrt{r} \cos(\theta / 2) - i \sqrt{r} \operatorname{sin}(\theta / 2)$$

Ahora se analizará cómo la operación de obtener las raíces cuadradas de un número complejo permiten calcular el Conjunto Julia. En el método anterior se manejó una función cuadrática $T(z) = z^2$ como una función compleja donde $z = x + iy$. Se vio

que todas las órbitas de puntos que satisfacían que $|z| \neq 1$ tendían hacia uno de dos lugares: Si $|z| < 1$, entonces la órbita tiende a 0, pero si $|z| > 1$, entonces la órbita tiende a infinito. Por lo que el círculo de radio 1 es el límite entre los puntos que son atraídos hacia 0 y los puntos cuyas órbitas escapan. Este era el Conjunto Julia de T.

Lo anterior sugiere que se puede encontrar el Conjunto Julia de T calculando órbitas atrasadas, utilizando la raíz cuadrada. Si z_0 satisface que $|z| = r > 1$, entonces las dos raíces cuadradas de z_0 tienen módulo $\sqrt{r}$, por lo que se tiene que $r < \sqrt{r} < 1$. En ambos casos, $\sqrt{r}$ está más cerca de 1 que r. Esto significa que no importa si $r > 1$ ó $r < 1$, las dos raíces cuadradas de z_0 se encuentran más cercanas al círculo de radio 1 que z_0. Ahora se seleccionará a una de las dos raíces cuadradas de z_0 y se denominará a este nuevo punto z_{-1}. Como $T(z_{-1}) = z_0$, se puede pensar en z_{-1} como si fuera el primer punto de una de las órbitas "atrasadas" de z_0. Se puede repetir este proceso eligiendo una de las dos posibles raíces cuadradas de z_{-1}, y denominándola z_{-2}. El proceso continua de forma similar: se selecciona una raíz cuadrada en particular y entonces se obtiene su raíz cuadrada, sucesivamente. Cada vez que se selecciona una de las dos raíces cuadradas, el punto elegido se encuentra cada vez más cerca al Conjunto Julia. La órbita atrasada resultante de la función cuadrática tiende hacia el círculo de radio 1, es decir, al Conjunto Julia de la función T.

Para cada iteración en este proceso, una de las dos raíces cuadradas debe ser elegida. Sin embargo, no se recomienda elegir siempre a la raíz "positiva" o a la raíz "negativa", es mejor elegir a una de las dos opciones en forma aleatoria.

Este procedimiento funciona para producir una figura áspera y poco estética del conjunto Julia de la función $Q_c(z) = z^2 + c$. Para calcular la órbita atrasada de Q_c es notable que si

$$z^2 + c = w$$

entonces

$$z = \pm \sqrt{w - c}$$

El número complejo w - c posee dos raíces cuadradas, esas son las dos preimágenes de w bajo Q_c

4.2.4 Algoritmo Julia-2.

Con esta observación, se puede elaborar el siguiente algoritmo para calcular el Conjunto Julia de Q_c.

1. Seleccionar cualquier punto w_0 en el plano complejo.
2. Calcular una de las dos raíces cuadradas de (w_0 - c), eligiendo la raíz cuadrada positiva o negativa en forma aleatoria. Sea z_0 la variable que denota el valor de esta raíz cuadrada.
3. Reemplazar w_0 por z_0.
4. Ejecutar los pasos 2 y 3 un total de 30 mil veces, graficando en cada iteración el punto z_0 en el plano. Sin embargo, no graficar los primeros 50 puntos.

El programa de la figura 4.4 ejecuta este algoritmo. Este programa acepta como entrada el parámetro c y una semilla inicial z_0 para después calcular las 30 mil preimágenes de z_0. En cada iteración el programa elige aleatoriamente una de las dos preimágenes. La selección aleatoria se realiza mediante el uso de la función *random*. Cada vez que el programa encuentra la sentencia random(2), genera un número, ya sea 0 ó 1 de forma aleatoria, por lo que sentencia

```
theta = theta / 2 + pi * ( random(2) )
```

genera uno de los dos ángulos polares posibles asociados con la función raíz cuadrada; el nuevo ángulo polar se selecciona de forma aleatoria para que sea $\theta / 2$ o bien $\theta / 2+\pi$, dependiendo de si el número que se obtuvo de forma aleatoria fue 0 ó 1.

```
scanf("%f",&c1);
scanf("%f",&c2);
scanf("%f",&x0);
scanf("%f",&y0);
pi=3.141592;
for(i=1;i<=30000;i++)
        {
         w0=x0-c1;
         w1=y0-c2;
         if (w0==0) theta=pi/2;
         if (w0>0) theta=atan(w1/w0);
         if (w0<0) theta=pi+atan(w1/w0);
         r=sqrt(w0*w0+w1*w1);
         theta=theta/2+pi*(random(2));
         r=sqrt(r);
         x0=r*cos(theta);
         y0=r*sin(theta);
         m=(x0+2)*450/4;
         n=(2-y0)*450/4;
         if (i>50) putpixel(m,n,11);
        }
```

Figura 4.4: Programa Julia-2.

Se debe notar que en este programa no se grafican las primeras 50 preimágenes, sólo se grafican puntos en la órbita atrasada que se encuentra muy cerca del conjunto Julia

Utilizando el programa Julia-2 para graficar los Conjuntos Julia de la función Q_c para diversos valores de c, se obtienen resultados como los que muestra la figura 4.5.

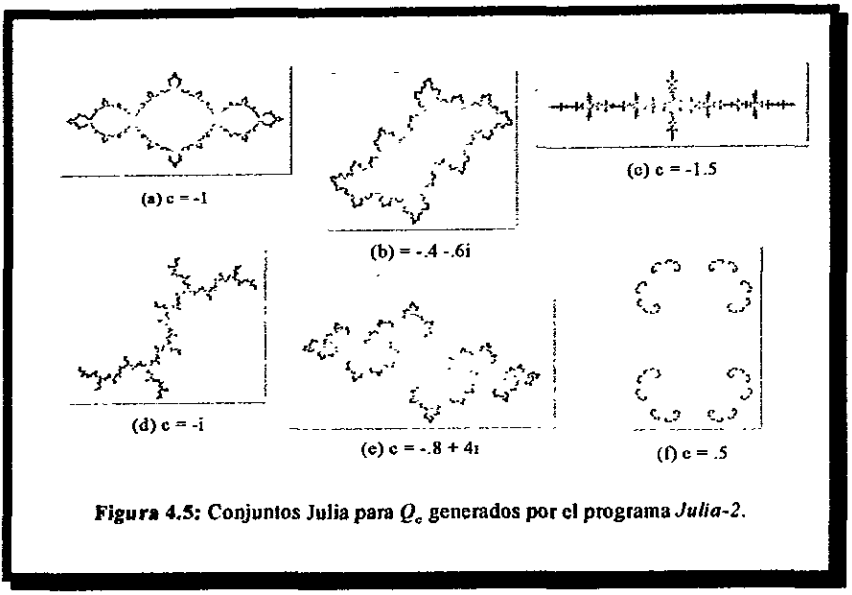

Figura 4.5: Conjuntos Julia para Q_c generados por el programa *Julia-2*.

Los Conjuntos Julia para estos valores de c poseen diferentes formas. Los Conjuntos Julia mostrados de los incisos (a) hasta (d) aparentemente consisten de una sola pieza. Para $c = -1$ y $c = -.4 - .6i$ los Conjuntos Julia se asemejan a las regiones de puntos que genera el programa Julia-1. En los incisos (c) y (d) el Conjunto Julia es una silueta que pareciera no delimitar ninguna región. Finalmente, el los incisos (e) y (f) el Conjunto Julia aparenta estar compuesto de varias "islas" de puntos.

El *Método de iteración atrasada* genera los conjuntos Julia más rápidamente que el método visto con anterioridad. Esto se debe principalmente a que sólo son calculados 30 mil puntos en una sola órbita, en lugar de las más de 1.5 millones de iteraciones que se realizan en el programa Julia-1. Sin embargo existen algunas desventajas. El Conjunto Julia para Q_{-1} producido por el programa Julia-2 no parece contener el borde completo que sí genera el programa Julia-1, aún y cuando se

incrementara el número de iteraciones en Julia-2. No obstante, este método proporciona una buena aproximación del Conjunto Julia.

Existe una diferencia muy obvia entre los Conjuntos Julia de la figura 4.5. En algunos casos, el Conjunto Julia está formado por una sola pieza, mientras que en otros casos está formado por varias piezas aisladas. Nuevamente se puede observar que los Conjuntos Julia poseen la propiedad de autosimilaridad. Si se llevan a cabo amplificaciones sucesivas, se observaría que los Conjuntos están formados por "nubes" de puntos que se encuentran esparcidas formando varias piezas separadas. A estos tipos de Conjuntos Julia suele llamárseles *Polvo Fractal*, aunque el término técnico más apropiado para este tipo de estructuras es *Conjunto de Cantor*.

4.3 Conjunto de Mandelbrot.

El Conjunto de Mandelbrot es considerado como uno de los objetos más intrincados y bellos en las matemáticas que, a pesar de su complejidad, es relativamente sencillo de calcular. Como se observó, la función cuadrática $Q_c(z) = z^2 + c$ exhibe diversos comportamientos dinámicos, y el Conjunto Julia de la función Q_c varía de acuerdo a los diferentes valores para c. En cierto sentido, se podría decir que el *Conjunto de Mandelbrot* es una compilación de todos esos diferentes fenómenos y que explica la forma en que se relacionan todas estas estructuras y figuras..

4.3.1 Puntos críticos y órbitas.

Para la construcción del Conjunto de Mandelbrot, simplemente se necesita entender a la órbita del punto 0 bajo la función cuadrática $Q_c(z) = z^2 + c$ para cada diferente valor de c. Debido a que la órbita de 0 bajo Q_c juega un rol especial (o crítico), es llamada *órbita crítica*, y al punto 0 se le conoce como *punto crítico*.

La razón por la que 0 es un punto crítico bajo la función $Q_c(z) = z^2 + c$ es por el hecho de que la derivada de la función Q_c se indetermina en el punto 0. Por otro lado, la razón por la que 0 es un punto especial proviene del hecho de que es el único punto que satisface que $Q_c(z) = c$ y que no existe algún otro punto z_0 en el plano complejo para el cual

$$z_0^2 + c = c$$

De hecho, si se toma cualquier otro punto w_0 en el plano, siempre existirán dos puntos z_0 en el plano que satisfacen que $Q_c(z_0) = w_0$. Es posible resolver la ecuación

$$z_0^2 + c = w_0$$

calculando las dos raíces cuadradas de $w_0 - c$, con lo que se obtienen dos valores diferentes para z_0, siempre que $w_0 - c \neq 0$. Sin embargo, $w_0 = c$ es el único valor en el plano complejo que posee sólo una preimágen bajo la función Q_c, y tal preimágen es el punto crítico 0. Otra de las razones por la que la órbita crítica resulta tan importante, es el siguiente hecho:

"Supóngase que Q_c posee una órbita de atracción periódica. Entonces la órbita crítica es atraída hacia esta órbita"

Este hecho tiene como consecuencia importante que la función Q_c puede tener, a lo sumo, un solo ciclo de atracción periódico, ya que la órbita crítica pude ser atraída, a lo sumo, hacia un solo ciclo de atracción.

Dada la importancia de la órbita del punto 0 tanto para determinar la estructura del Conjunto Julia como para encontrar ciclos de atracción, es natural cuestionarse por aquellos valores para c que posean órbitas críticas que escapan y por aquellos valores para c cuyas órbitas no escapan. Esta misma interrogante se formuló el matemático francés Benoit Mandelbrot, quien fue uno de los primeros en cuestionarse por el

conjunto de valores para c cuya órbita no escapa. Lo que encontró es el conjunto que lleva su nombre: el Conjunto de Mandelbrot.

4.3.2 Construcción del Conjunto de Mandelbrot.

Para ser precisos, el Conjunto de Mandelbrot, denotado como M, es el conjunto de valores de c para los cuales la órbita crítica de Q_c *no* tiende a infinito. Es importante enfatizar que el Conjunto de Mandelbrot es una figura en el plano c a diferencia del Conjunto Julia, que es una figura ubicada en el plano z.

Para conocer la forma del conjunto M se necesita conocer los valores de c que tienen órbitas críticas que escapan. Ciertos valores de c escapan de forma inmediata de la órbita crítica. Por ejemplo, es cierto que si $|c| > 2$ entonces la órbita de 0 escapa de forma inmediata. Para comprobar este hecho, se dirá que $|c| = 2 + \ell$, con $\ell > 0$. Cualquier punto z que cumpla con $|z| \geq |c|$ escapa bajo la iteración de Q_c (en particular, c escapa por sí mismo y $c = Q_c(0)$, por lo que la órbita de 0 escapa). Esto sucede debido a que si $|z| \geq |c| > 2$, entonces

por desigualdad del triángulo, $|Q_c(z)| = |z^2 + c| \geq |z|^2 - |c|$, entonces

$$|Q_c(z)| \geq |z|^2 - |z|$$

$$|Q_c(z)| = |z| (|z| - 1)$$

$$|Q_c(z)| \geq |z| (1 + \ell)$$

Esto significa que $|Q_c(z)| \geq |z| (1 + \ell)$ mientras que $|z| \geq |c|$ Pero se tiene que $1 + \ell > 1$. Por lo tanto, $|Q_c(z)| > |z|$. Este resultado puede ser interpretado diciendo que, mientras que z se encuentra fuera del círculo de radio $|c| > 2$ en el plano, su imagen $Q_c(z)$ se encuentra más alejado del origen que z Esto es, bajo una iteración, estos puntos se mueven cada vez más cerca del infinito. Por lo tanto, se

puede aplicar el mismo argumento para decir que bajo dos iteraciones, los puntos se mueven cada vez más alejados del origen.

Este razonamiento se puede aplicar a $Q_c^2(z)$ para encontrar

$$|Q_c^2(z)| = |Q_c(Q_c(z))| \geq |Q_c(z)|(1+\ell), \text{ ya que } |Q_c(z)| \text{ es mayor que 2}$$

$$|Q_c^2(z)| \geq |z|(1+\ell)^2$$

Aquí se utilizó el hecho de que $|Q_c(z)| > |z|(1+\ell)$, con lo que se puede decir que

$$|Q_c^n(z)| \geq |z|(1+\ell)^n$$

donde $1+\ell > 1$. El resultado de elevar a la potencia n a $1+\ell$ crecerá a medida que el valor de n sea incrementado, lo que lleva a decir que

$$(1+\ell)^n \to \infty$$

cuando $n \to \infty$. Por consiguiente

$$|Q_c^n(z)| \to \infty$$

también cuando $n \to \infty$.

Como consecuencia de lo anterior, ahora se sabe que el Conjunto de Mandelbrot se encuentra dentro del círculo de radio 2 en el plano complejo Para encontrar el Conjunto de Mandelbrot, se necesita verificar los valores de c dentro del círculo de radio 2 para comprobar si pertenecen al conjunto M. Esto puede realizarse utilizando la siguiente observación: si la órbita de c en algún momento abandona el disco de radio 2, entonces necesariamente escapa hacia el infinito. Esta observación proporciona un algoritmo para calcular el Conjunto de Mandelbrot, en el que se calculará la órbita de 0 de acuerdo a la sucesión $Q_c^n(z)$:

$$0,\ c,\ c^2+c,\ (c^2+c)^2+c,\ [(c^2+c)^2+c]^2+c, \ldots$$

y se verificará si algún punto en esta órbita tiene módulo mayor a 2. Una vez que esto ocurra, se habrá garantizado que la órbita crítica escapa y que *c no* pertenece al Conjunto de Mandelbrot. El algoritmo es el siguiente:

1. Dividir la región comprendida por $-2 \leq |x|, |y| \leq 2$ en el plano, en un área de 400x400.
2. Manejar a cada punto en el área de 400x400 como un valor de *c*.
3. Para cada valor de *c*, comprobar si la órbita de 0 bajo Q_c escapa dentro de las primeras 30 iteraciones.
4. Si la órbita escapa, colorear de negro a *c*.
5. Si la órbita no escapa, colorear de azul a *c*.

La figura 4.6 muestra el código fuente en lenguaje C del programa que ejecuta este algoritmo.

Es muy notoria la similitud entre este programa y el programa Julia-1. La única diferencia es muy sutil: para cada punto en el área de 400x400 en el plano, se debe utilizar el correspondiente valor de *c* durante toda la iteración. Esta es la razón de utilizar las sentencias

```
x = c1
y = c2
```

desde el inicio del programa y después ejecutar todos los cálculos usando las variables *x* e *y*. Nuevamente se ha hecho uso de la simetría para acelerar la ejecución del programa. A diferencia de los Conjuntos Julia de Q_c, *M* no es simétrico con respecto al origen, es decir el conjunto *M* no es el mismo si se reemplazan los puntos c_1 y c_2 por sus negativos. No obstante, el Conjunto de Mandelbrot es simétrico con respecto el eje *x* en el siguiente sentido: Supóngase que se sabe que el punto $c = c_1 + ic_2$ se encuentra

en el Conjunto de Mandelbrot, con $c_2 > 0$, entonces $c_1 - ic_2$ también debe pertenecer al conjunto M.

```
for(i=1;i<=400;i++)
    {
     for(j=1;j<=200;j++)
          {
           c1=-1.5+3*i/600;
           c2=1-3*j/600;
           x=c1;
           y=c2;
           for(n=1;n<=30;n++)
                {
                 x1=x*x-y*y+c1;
                 y1=2*x*y+c2;
                 r=x1*x1+y1*y1;
                 if (r>4) break;
                 x=x1;
                 y=y1;
                }
           if (r>4) continue;
           putpixel(i,j,11);
           putpixel(i,400-j,11);
          }
    }
```

Figura 4.6: Programa Mandelbrot.

Esta simetría se debe al concepto de *el conjugado* de un número complejo Por lo que si se conoce la órbita de z bajo Q_c, se puede obtener la órbita de $\bar{z}$ bajo $Q_{\bar{c}}$ simplemente tomando el conjugado complejo en cada iteración. Este hecho es utilizado en el programa Mandelbrot para reducir a la mitad el número de cálculos que son necesarios para producir el conjunto M. De hecho, si el pixel correspondiente a c es

iluminado con azul, entonces se colorea con azul a $\bar{c}$ de forma inmediata. La salida producida por el programa Mandelbrot se muestra en la figura 4.7. Cabe recordar que la figura generada por el programa Mandelbrot se encuentra en el plano *c*.

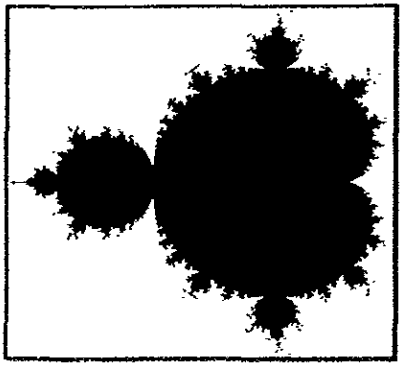

Figura 4.7: Conjunto de Mandelbrot.

En la figura 4.7, la región del conjunto *M* está coloreada de azul oscuro. A diferencia de los Conjuntos Julia, para los que se obtienen diferentes figuras para cada diferente valor de *c*, el conjunto *M* posee sólo una imagen. En el programa Mandelbrot, el ciclo anidado *for* más interno realiza las 30 iteraciones en las que, para cada valor de *c* se comprueba si la órbita de 0 bajo *c* escapa. Si se visualiza el conjunto *M* para diferentes valores de *n*, se obtienen imágenes como las de la figura 4.8.

La imagen del conjunto *M* en la iteración 30, como se muestra en la figura 4.7, consiste de una región con forma de cardioide. La cúspide del cardioide se encuentra exactamente en el punto $c = 1/4$. En la parte izquierda de se encuentra una amplia región circular, la cual se encuentra unida al cardioide en el punto $c = -3/4$. Finalmente,

en la extrema izquierda de la figura se aprecia una "cola" que emana del conjunto *M* y apunta a la izquierda. Esta "cola" o antena se encuentra a lo largo del eje real *y* termina exactamente en el punto $c = -2$.

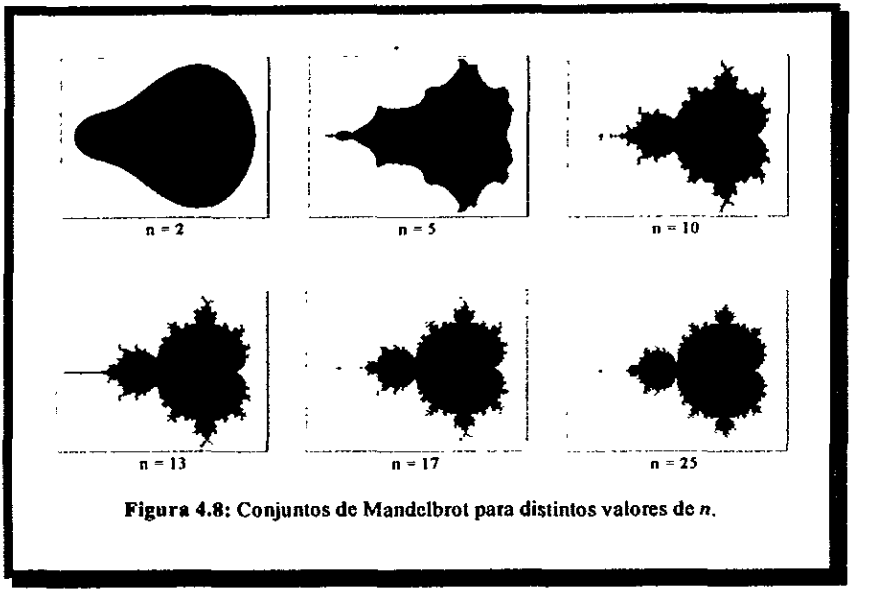

Figura 4.8: Conjuntos de Mandelbrot para distintos valores de *n*.

Si se realizan amplificaciones sucesivas al conjunto *M*, se obtendrían pequeñas copias del conjunto de Mandelbrot, cumpliendo con la característica de autosimilaridad en un fractal. Además, todos los comportamientos dinámicos que suceden en el cuerpo principal de *M*, también forman parte del comportamiento de cada una de esas pequeñas copias En el Apéndice se muestra el código fuente completo del programa Mandelbrot para compilarse y ejecutarse en el Borland Turbo C.

4.4 Pruebas de ejecución.

Como se ha mencionado, para generar las diferentes imágenes de los Conjuntos Julia y de Mandelbrot, los algoritmos y programas aquí presentados involucran la

realización de una gran cantidad de operaciones. En el caso del programa Julia-1, se realizan 1.6 millones de iteraciones, el programa Julia-2 realiza 30 mil iteraciones y el programa Mandelbrot realiza 2.4 millones de iteraciones. Es de esperarse que la ejecución de estos programas no sea inmediata, debido a que en cada iteración se realizan varias operaciones aritméticas y lógicas.

Sin embargo, el continuo desarrollo de tecnologías en cómputo ha impulsado el estudio de la geometría fractal y ha hecho posible la ejecución eficiente de este tipo de algoritmos, ya que la velocidad de procesamiento de operaciones que brinda un sistema de cómputo actual representa la principal herramienta para ser aprovechada. De hecho, es gracias a un lenguaje de programación y una computadora que se obtuvieron las primeras imágenes de los Conjunto Julia y Mandelbrot.

No obstante lo anterior, los tiempos de ejecución para ambos programas, además de ser dependientes de la cantidad y del tipo de operaciones a realizar, se encuentran sujetos a la capacidad de procesamiento del equipo en el que se ejecuten los programas. Es otras palabras, es posible obtener diferentes tiempos de ejecución dependiendo de la arquitectura del equipo de cómputo que se esté utilizando. Para comprobar esta afirmación, se realizaron pruebas de funcionalidad en varios equipos para comparar los tiempos de respuesta en cada caso, ejecutando una copia de los programas Julia-1, Julia-2 y Mandelbrot. Los sistemas utilizados fueron:

- Computadora personal de escritorio con procesador Intel Pentium a 133 Mhz ejecutando sistema operativo MS-DOS versión 6.22.
- Computadora personal de escritorio con procesador Intel Pentium a 133 Mhz ejecutando sistema operativo Windows 95.

- Servidor de red con procesador Intel Pentium a 133 Mhz ejecutando sistema operativo Unix Release V version 3.
- Sistema multiprocesador Sequent Balance 8000 con 4 procesadores RISC de 32 bits ejecutando sistema operativo Dynix

4.4.1 Sistemas uniprocesador convencionales.

Para evaluar la capacidad de procesamiento de un sistema uniprocesador convencional se utilizaron dos entornos de sistema operativo diferentes: sistema operativo MS-DOS y sistema operativo Windows 95.

MS-DOS es el sistema operativo de más amplio uso en el terreno de las computadoras personales. Una de sus características principales es el trabajo con un sistema de archivos basado en una *tabla de asignación de archivos* (File Allocation Table), que reside en memoria principal y no en disco, con lo que se reduce el acceso a discos para administrar los archivos. Sin embargo, MS-DOS tiene varias limitaciones con respecto a otros sistemas operativos, ya que sólo puede manejar a un usuario ejecutando un solo programa utilizando únicamente 640k de memoria, sin importar la si la cantidad de memoria disponible es mayor, no existe multiprogramación ni multitarea, además, los servicios que proporciona suelen ser deficientes y los programas con frecuencia ignoran al sistema operativo y realizan sus propias operaciones de E/S.

A pesar de estas limitaciones, MS-DOS a ganado gran popularidad y uso entre los usuarios de computadoras personales. Para la ejecución de los programas, se empleó una computadora personal con procesador Intel Pentium a 133 Mhz con 16 Mb de RAM y ejecutando la versión 6.22, observándose los siguientes resultados:

a) En el caso del programa Julia-1, el tiempo de ejecución varía de entre 4 y 8 segundos, dependiendo del valor asignado a la variable *c*. La ejecución más lenta se obtiene al generar el disco centrado en el origen, cuando $c = 0$, que toma 8 segundos de ejecución

b) El programa Julia-2 esta desarrollado con base en un algoritmo que emplea menos procesos iterativos y que resulta ser más rápido. Los tiempos de ejecución varían de 2.2 a 2.6 segundos.

c) El programa Mandelbrot tarda en generar el conjunto *M* un total de 8.8 segundos.

MS-DOS maneja estos programas como cualquier otro, siempre por debajo de los 640k de memoria principal. A pesar de esto y de ser un sistema *unitarea* y *monousuario*, el uso de un procesador Intel Pentium, el cual posee un coprocesador matemático integrado, ayuda a agilizar las operaciones aritméticas y con ello a reducir los tiempos de ejecución.

Por otra parte, el sistema operativo *Windows* nace como una extensión de MS-DOS como un ambiente gráfico que permite ejecutar más de un programa de aplicación a la vez y transferir información entre aplicaciones. En *Windows*, el usuario puede escoger varios programas a ejecutar, cada uno en una ventana individual, con la posibilidad de interactuar unas ventanas con otras. La versión denominada *Windows 95* es manejada como un sistema operativo autónomo con interfaz gráfica, independiente de MS-DOS y que se desempeña como un sistema operativo *multitarea prioritaria* de 32 bits. Sin embargo, no obstante a su gran popularidad y a sus constantes mejoras, *Windows 95 resulta ser un sistema operativo que consume gran cantidad de recursos y* que su capacidad multitarea aún se ve superada por otros sistemas operativos.

Al utilizar *Windows 95* como sistema operativo para ejecutar los programas Julia-1, Julia-2 y Mandelbrot, se advierten dos cuestiones importantes. Primero, los tiempos de ejecución crecen con respecto a MS-DOS versión 6.22:

a) En el caso del programa Julia-1 los tiempos varían desde 5.2 segundos hasta 52 segundos, dependiendo del valor asignado a c. El tiempo más prolongado de ejecución para Julia-1 se obtiene al con $c = 0$, que genera el disco centrado en el origen en 52 segundos.

b) Para el programa Julia-2 los tiempos van desde 3.2 a 3.4 segundos, es decir, aproximadamente 1.2 segundos más con respecto a MS-DOS.

c) El programa Mandelbrot genera el conjunto M en 9.8 segundos, es decir, un segundo más que en MS-DOS

La segunda observación es relativa a la capacidad multitarea de *Windows 95*. En efecto, mientras se generan las figuras correspondientes a los Conjuntos Julia o Mandelbrot por medio de alguno de los programas, o bien mientras se encuentre activa cualquier otro programa, un usuario puede permutarse de aplicación y llevar a cabo otras tareas. Sin embargo, al permutar de aplicación mientras se genera alguno de los conjuntos, la ejecución del programa se detiene y la atención del sistema operativo pasa hacia la otra aplicación. El programa reanuda su ejecución sólo hasta que el usuario cambia la atención del sistema operativo hacia el programa que estaba generando, ya sea el Conjunto Julia o el Conjunto de Mandelbrot, con lo que los tiempos de ejecución podrían incrementarse de forma indefinida.

Cabe mencionar que, para no afectar los tiempos de ejecución, los programas se ejecutaron separadamente y ninguna otra aplicación se activó desde el inicio hasta la terminación de la ejecución de los programas los programas Julia y Mandelbrot.

4.4.2 Sistema multitarea de tiempo compartido.

En los ambientes multitarea de tiempo compartido que ofrecen los sistemas Unix, varios usuarios tienen la posibilidad de ejecutar varios programas de aplicación de diversas formas. Usualmente, los *shells* de Unix no tienen orientación gráfica, aunque en la actualidad son utilizadas varias interfases de orientación gráfica como *X Windows* de MIT, *Open Look* de AT&T y Sun, y *DECwindows* de DEC.

En este caso, para la ejecución de los programas se utilizó un equipo Hewlett Packard LM con procesador Intel Pentium a 133 Mhz y 98 Mb en RAM, ejecutando Sistema Operativo SCO Unix System V release 5.2. Para estas evaluaciones se utilizó el compilador de lenguaje C, proporcionado por el sistema operativo, para generar los programas ejecutables apropiados para la interfaz gráfica del Unix de SCO. Los tiempos de ejecución que se obtienen son muy similares a los obtenidos con MS-DOS:

a) Para el programa *Julia-1*, la ejecución se realiza desde de 3 a 6 segundos, dependiendo del valor de c.
b) El programa Julia-2 tarda en ejecutarse de 2 a 2.4 segundos, igualmente los tiempos se encuentran en función del valor asignado a c.
c) El programa Mandelbrot genera el conjunto M en aproximadamente 8.6 segundos.

La principal diferencia entre esta evaluación y la anterior es el sistema operativo y la cantidad de RAM con que cuenta el equipo. Esta aparente similitud en los tiempos de ejecución puede ser engañosa. Es cierto que, en un ambiente con sistema operativo Unix, el tiempo de ejecución de los programas sólo se redujo aproximadamente un segundo con respecto a MS-DOS. Sin embargo, y como ya se mencionó, existen varias formas de ejecutar estos, y otros programas de aplicación. Por ejemplo, los sistemas Unix disponen de herramientas para modificar la *prioridad* de ejecución de un proceso,

con lo que, incrementando la prioridad se logra que un proceso se ejecute antes que otros con prioridades menores y por lo tanto su ejecución termine más rápidamente, a costa de otros procesos en el sistema que dispondrán de menos tiempo de la atención del procesador. Si por el contrario se decrementa la prioridad de un proceso, éste tardará más tiempo en ejecutarse y hará que demande menos tiempo del procesador.

El manejo de prioridades, aunado a la capacidad de Unix para ejecutar procesos en segundo plano o *background,* permite que los programas sean ejecutados sin necesidad de esperar su culminación para realizar otras tareas. Es posible asignar una prioridad determinada de ejecución al proceso, ejecutarlo en *background* y redireccionar su salida hacia un archivo, con lo que se obtiene una velocidad de ejecución mayor o menor, de acuerdo a la prioridad asignada al proceso. Estas características multitarea en los sistemas Unix no existen en MS-DOS y son más poderosas a las que ofrece *Windows 95.* Para el caso de los programas que generan los Conjuntos Julia y Mandelbrot no es la excepción, los programas pueden ejecutarse en segundo plano con diferentes prioridades de acuerdo a las necesidades de respuesta del usuario del sistema. Igualmente, es importante mencionar para la ejecución de los programas se utilizó clave de supervisor o *root* y que ningún otro usuario se encontraba activo en el sistema, además, los programas se ejecutaron separadamente y para no afectar los tiempos de ejecución, se desactivaron la mayor parte de los procesos y servicios activos en el sistema, para que ninguna aplicación ajena a los programas estuviera activa mientras los programas eran ejecutados.

4.4.3 Sistema multiprocesador.

Podría pensarse que los tiempos de ejecución hasta ahora obtenidos son muy aceptables. De hecho, a pesar de las limitantes que pudiera ofrecer un sistema

operativo, la arquitectura de los procesadores Intel Pentium provoca que sean capaces de realizar gran cantidad operaciones en cada vez menos tiempo.

Sin embargo, es muy importante enfatizar que estos programas son tan sólo algunos ejemplos de procesos iterativos con una gran cantidad de evaluaciones que suelen acaparar la atención del procesador, y que para el desarrollo de aplicaciones concretas con gran demanda de recursos, el desempeño de un sistema uniprocesador pudiera no ser muy útil. El objetivo principal de estas pruebas es evaluar y comparar el desempeño de varias organizaciones de sistemas de cómputo y obtener aquella que brinde los mejores resultados.

Para la última de las evaluaciones se utilizó un equipo de cómputo con organización *multiprocesador*, un sistema *Sequent Balance 8000*. Este sistema *Sequent* emplea una arquitectura de procesadores paralela para mejorar los tiempos de ejecución de las diversas aplicaciones de usuario. *Sequent* opera con 4 unidades de proceso RISC, pero este número puede incrementarse hasta 30, repartidos por pares en *subsistemas de procesamiento* con microprocesadores de 32 bits, unidades de manejo de memoria y unidades de punto flotante. Para aumentar el número de procesadores en el sistema, únicamente se requiere agregar los subsistemas extra deseados sin necesidad de eliminar el hardware existente.

El sistema operativo utilizado es *Dynix*, que soporta dos ambientes o *universos*, el primero de ellos soporta el ambiente Berkeley Unix 4.2 mientras que el segundo soporta programas y usuarios de sistemas AT&T System V.2. Los programas compilados en un *universo* pueden ejecutarse en el otro y un usuario puede cambiarse de un *universo* a otro. Esto permite a los usuarios de *Dynix* ejecutar software de aplicación en ambientes Unix.

El sistema operativo *Dynix* permite que una aplicación este formada por múltiples procesos cooperantes, todos ejecutándose simultáneamente en diferentes procesadores. Gran cantidad de aplicaciones secuenciales pueden transformarse en aplicaciones paralelas, produciendo mejoras lineales (o casi lineales) en cuanto a su desempeño. En su configuración máxima con 30 procesadores, *Sequent* acelera la ejecución de las aplicaciones de usuario hasta 27.5 veces. En el sistema utilizado para las pruebas, operando con 4 unidades de proceso, se espera que los tiempos de respuesta con 4 procesadores se reduzcan aproximadamente 3.7 veces

Gracias a que *Dynix* soporta aplicaciones de usuario para sistemas Unix como AT&T, los programas Julia-1, Julia-2 y Mandelbrot pueden ejecutarse sin problemas Igualmente se utilizó el compilador de C de *Dynix* para producir los archivos ejecutables correspondientes a cada programa.

Al ejecutar los programas, la reducción en los tiempos de respuesta es muy notoria.

a) En el caso del programa Julia-1, la ejecución más lenta se da con $c = 0$ en un tiempo aproximado de 1.7 segundos, mientras que la ejecución más rápida, con $c =$ -1 tarda 0.9 segundos.

b) Para el programa Julia-2, los tiempos van de 0.5 a 0.6 segundos.

c) El programa Mandelbrot genera el conjunto M en un total de 2 4 segundos.

Como en el caso anterior, para evitar afectar los resultados, la ejecución de los programas se realizó con clave *root* y ningún otro usuario se encontraba activo en el sistema, los programas se ejecutaron separadamente y se desactivaron la mayor parte de los procesos y servicios activos en el sistema para que ninguna aplicación estuviera activa mientras los programas eran ejecutados.

La figura 4.9 muestra una tabla comparativa de los tiempos de respuesta en las cuatro evaluaciones realizadas con los programas Julia-1, Julia-2 y Mandelbrot.

	MS-DOS	Windows 95	SCO Unix V	Dynix
Julia-1	de 4 a 8	de 5.2 a 52	de 3 a 6	de 0.9 a 1.7
Julia-2	de 2.2 a 2.6	de 3.2 a 3.4	de 2 a 2.4	de 0 5 a 0.6
Mandelbrot	8.8	9 8	8.6	2.4

Figura 4.9: Comparación de tiempos de ejecución (en segundos) de los programas Julia-1, Julia-2 y Mandelbrot.

Como puede apreciarse en la figura 4.9, los tiempos de respuesta que ofrece el equipo *Sequent* son aproximadamente 4 veces mejores con respecto al resto de los resultados. El sistema operativo *Dynix* igualmente permite controlar la prioridad de ejecución de los procesos, así como ejecutarlos en segundo plano para explotar su capacidad *multitarea* y *multiprocesamiento*. Por otro lado, el desempeño del sistema *Sequent* puede modificarse agregando o removiendo procesadores, sin necesidad de realizar ajustes al software; además, mientras que la naturaleza multiprocesadora del sistema es completamente transparente en situaciones multiusuario, la arquitectura multiprocesador permite acelerar las aplicaciones paralelas mediante la utilización de múltiples procesadores.

La figura 4.10 muestra una gráfica que ilustra los resultados obtenidos para cada evaluación. Para esta gráfica se tomaron en cuenta los valores máximos obtenidos en cada programa y es muy notoria la forma en que las curvas están próximas a llegar al eje *x* en el momento que aparece la ejecución en la arquitectura multiprocesador.

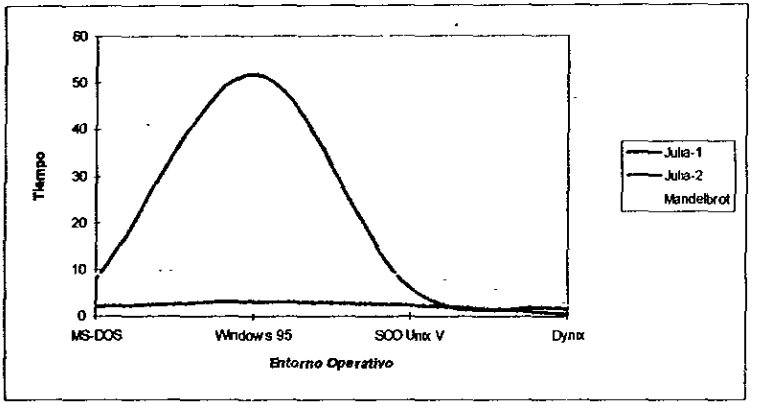

Figura 4.10: Gráfico comparativo de tiempos de ejecución (en segundos).

Así, el gran potencial ofrecido por sistemas multiprocesadores para descomponer programas secuenciales en varios procesos para su ejecución en paralelo, aporta herramientas de desarrollo muy útiles para ejecutar procesos recursivos con gran cantidad de iteraciones empleados en investigación científica y tecnológica, brindando un ambiente que optimiza la ejecución de este tipo de procesos

CONCLUSIONES

Por un lado, el creciente campo de aplicación del paralelismo en la ciencia y la tecnología, requiere de sistemas de cómputo capaces de responder eficazmente a sus demandas de velocidad de respuesta y de alto rendimiento, lo que ha obligado a los sistemas multiprocesador paralelos a ser capaces de ejecutar varios procesos en forma concurrente de forma óptima. La adecuada distribución de los procesos a los procesadores, así como la planificación de tareas y recursos recae sobre el sistema operativo que adopten dichos sistemas multiprocesador. En otras palabras, para el correcto funcionamiento de este tipo de sistemas de cómputo, además de una arquitectura adecuada, es necesario contar con un sistema operativo que ofrezca un desempeño eficiente en la ejecución de procesos concurrentes. Por otro lado, el sistema operativo multitarea de tiempo compartido Unix ofrece alta potencialidad ofreciendo herramientas de software que dan apoyo a ambientes de desarrollo de programas. Debido a esto, el Sistema Operativo Unix ha sido adoptado por un gran número de sistemas de cómputo de alta velocidad.

Como consecuencia de lo anterior, nace la necesidad de aprovechar el entorno que ofrece Unix para la ejecución de aplicaciones de programación en paralelo, mediante la reestructuración del diseño del sistema operativo para así simular un entorno multiprocesador en el ambiente multitarea de tiempo compartido proporcionado por los sistemas Unix en los que sea posible ejecutar aplicaciones de

programación paralela. El diseño original del sistema cuenta con mecanismos de protección al propio sistema para evitar conflictos al compartir una unidad de proceso. El principal problema para un sistema multiprocesador es de asignación, al planificar los procesos sobre los diferentes procesadores existentes. Se analizaron varios modelos para la planificación de los procesos, concluyendo que los problemas de asignación pueden ser resueltos involucrando dos factores, el sistema operativo y la arquitectura del equipo: primeramente se necesita una reestructuración de los algoritmos de planificación de procesos utilizados originalmente por Unix para llevarlos hacia un modelo multiprocesador, con mejoras y ampliaciones en sus mecanismos para la asignación y protección de los datos. En segundo lugar, la arquitectura que favorece mejor el paralelismo es una organización simétrica en la que se coloquen varios procesadores compartiendo un espacio único de memoria, en donde cada procesador pueda ejecutar cualquier proceso, de acuerdo a la planificación realizada por el sistema operativo.

El materializar este concepto llevaría a un sistema multiprocesador con un sistema operativo Unix capaz de llevar a cabo ejecución en forma paralela de los diferentes procesos del sistema, con el consecuente aumento en el rendimiento del sistema, muy útil hoy en día en diversos campos de la investigación científica y tecnológica.

APÉNDICE

Programa Julia-1

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main(void)
{
  float m,n,c1,c2,x0,y0,x,y,x1,y1,z;
  int i;

  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;

  textmode(C80);
  clrscr();
  gotoxy(25,4); printf("Conjunto Julia: Area iluminada");
  gotoxy(25,8); printf("Parte real c1 = ");
  gotoxy(50,8); scanf("%f",&c1);
  gotoxy(25,10); printf("Parte imaginaria c2 = ");
  gotoxy(50,10); scanf("%f",&c2);
  clrscr();

  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if (errorcode != grOk)
  {
    printf("Error Grafico: %s\n", grapherrormsg(errorcode));
    printf("Presione cualquier tecla para terminar:");
    getch();
    exit(1);
```

```
}

setcolor(getmaxcolor());

cleardevice();
for(m=0;m<=400;m++)
      {
       x0=-1+m/100;
       for(n=0;n<=400;n++)
             {
              y0=1-n/100;
              x=x0;
              y=y0;
              for(i=1;i<=20;i++)
                    {
                     x1=x*x-y*y+c1;
                     y1=2*x*y+c2;
                     x=x1;
                     y=y1;
                     z=x*x+y*y;
                     if (z>4) break;
                    }
              if (z>4) continue;
              putpixel(200+m,100+n,11);
              putpixel(400-m,300-n,11);
             }
      }

getch();
closegraph();
return 0;
}
```

Programa Julia-2

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main(void)
{
  float c1,c2,x0,y0,pi,w0,w1,theta,r;
  int i,m,n,mm,nn;

  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;

  textmode(C80);
  clrscr();
  gotoxy(25,4); printf("Conjunto Julia: Polvo Fractal");
  gotoxy(25,8); printf("Parte real c1 = ");
  gotoxy(50,8); scanf("%f",&c1);
  gotoxy(25,10); printf("Parte imaginaria c2 = ");
  gotoxy(50,10); scanf("%f",&c2);
  gotoxy(25,12); printf("Semilla inicial x0 = ");
  gotoxy(50,12); scanf("%f",&x0);
  gotoxy(25,14); printf("Semilla inicial y0 = ");
  gotoxy(50,14); scanf("%f",&y0);
  pi=3.141592;
  clrscr();

  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if (errorcode != grOk)
  {
    printf("Error Grafico: %s\n", grapherrormsg(errorcode));
    printf("Presione cualquier tecla para terminar:");
    getch();
```

```
    exit(1);
  }

  setcolor(getmaxcolor());

  cleardevice();
  for(i=1;i<=30000;i++)
       {
        w0=x0-c1;
        w1=y0-c2;
        if (w0==0) theta=pi/2;
        if (w0>0) theta=atan(w1/w0);
        if (w0<0) theta=pi+atan(w1/w0);
        r=sqrt(w0*w0+w1*w1);
        theta=theta/2+pi*(random(2));
        r=sqrt(r);
        x0=r*cos(theta);
        y0=r*sin(theta);
        m=(x0+2)*450/4;
        n=(2-y0)*450/4;
        if (i>50) putpixel(m,n,11);
       }

  getch();
  closegraph();
  return 0;
}
```

Programa Mandelbrot

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main(void)
{

  float i,j,n,c1,c2,x,y,x1,y1,r;

  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;

  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if (errorcode != grOk)
  {
    printf("Error Grafico: %s\n", grapherrormsg(errorcode));
    printf("Presione cualquier tecla para terminar:");
    getch();
    exit(1);
  }

  setcolor(getmaxcolor());

  cleardevice();

  for(i=1;i<=400;i++)
       {
        for(j=1;j<=200;j++)
              {
               c1=-1.5+3*i/600;
               c2=1-3*j/600;
               x=c1;
               y=c2;
```

```
                for(n=1;n<=30;n++)
                    {
                     x1=x*x-y*y+c1;
                     y1=2*x*y+c2;
                     r=x1*x1+y1*y1;
                     if (r>4) break;
                     x=x1;
                     y=y1;
                    }
                if (r>4) continue;
                putpixel(i,j,11);
                putpixel(i,400-j,11);
               }
        }

  getch();
  closegraph();
  return 0;
}
```

GLOSARIO

ALU Unidad Aritmética y Lógica, encargada de llevar a cabo las operaciones aritmética y lógicas en una unidad de proceso.

Aplicación Conjunto de programas necesarios para llevar a cabo una actividad determinada de cómputo.

Apuntador Variable que contiene una dirección de memoria.

Arreglo Colección de objetos del mismo tipo que se referencían por un nombre común.

Buffer Área de memoria principal para retener datos durante transferencias de entrada/salida.

Bus Conjunto de líneas de flujo utilizadas para transportar datos o instrucciones.

Cambio de contexto *Permutación de la atención* de un procesador de un proceso a otro colocados en memoria compartida.

Código fuente Conjunto de sentencias de un lenguaje de programación necesarias para obtener un programa ejecutable.

Cola Estructura de datos en la que el primer dato en entrar es el primero en salir.

Compilador Programa que transcribe el código fuente de un programa a

código objeto o programa ejecutable.

Concurrencia Técnica mediante la cual varios sucesos pueden darse simultáneamente en varios recursos diferentes.

Conmutador Dispositivo encargado de direccionar flujos de un bus de datos hacia varios destinos.

Contención Problema presentado en sistemas con memoria compartida cuando diferentes procesos o procesadores intentan tener acceso a la misma localidad de memoria

CPU Central Process Unit, Unidad Central de Proceso.

Econometría Rama de la economía que expresa las teorías económicas en términos matemáticos, en orden de verificarlas mediante métodos estadísticos.

Ensamblador Lenguaje de programación propio de cada procesador que maneja de forma directa los recursos de hardware del sistema.

Estación de trabajo Nodo terminal de un ambiente de red, conectado a su vez con otras estaciones de trabajo y con servidores de aplicaciones y de archivos.

Estructura de datos Agrupación de variables bajo un mismo nombre que proporciona un manejo más dinámico de datos y un medio eficaz de mantener junta información relacionada.

Evento Tipo de proceso especial que sirve para reanudar la ejecución de un proceso. Un evento puede ser la finalización de una

operación de E/S.

Exclusión mutua Herramienta de seguridad mediante la cual un proceso obtiene acceso a un recurso compartido impidiendo que otros procesos hagan simultáneamente lo mismo.

Firmware Se refiere al software almacenado en un dispositivo de hardware, como un microchip propio del fabricante del sistema.

Fractal Representación gráfica de la evaluación repetitiva de una ecuación y la combinación de diferentes procesos matemáticos.

Hardware Componentes físicos y tangibles de un sistema de cómputo.

Intérprete Programa que transcribe una a una las líneas de un programa fuente a un programa objeto o ejecutable.

Interrupción Mecanismo mediante el cual un proceso puede cambiar su estado y ser atendido por un procesador o pasar a un estado de espera.

Kernel Núcleo del sistema operativo Unix que se encarga del manejo de memoria, control de procesos y administración del sistema de archivos.

Lenguaje Conjunto de expresiones que son interpretadas o traducidas en programas ejecutables.

Lista Estructura de datos que permite manejar de forma bidireccional el almacenamiento de datos.

Llamada a sistema Solicitud hecha por un proceso de algún servicio proporcionado

por el sistema operativo

Matriz Arreglo de $n \times m$ elementos similares bajo un mismo nombre o identificador.

Microcódigo Instrucciones ubicadas por debajo del lenguaje ensamblador almacenadas en microchips o microprocesadores.

Modelo Representación matemática de algún hecho o situación de la vida real.

Pila Estructura de datos en la que el primer dato en entrar es el último en salir.

Planificador Algoritmo encargado de la asignación de recursos, principalmente procesos, procesadores y memoria, en un sistema de cómputo.

Proceso Programa en ejecución, que puede estar en cualquiera de tres estados: ejecución, listo para ejecución y bloqueado o *dormido*.

Punto a Punto Computación entre nodos similares o equivalentes.

Región crítica Momento en el que un proceso obtiene acceso a datos compartidos modificables. Cuando esto sucede, el proceso se encuentra en una *región crítica*.

Registro Combinación lógica de datos de diferentes tipos en un nuevo tipo de dato.

Secuencial Técnica de ejecución de procesos en la que, hasta alcanzar la finalización de un proceso, se puede iniciar la ejecución de otro.

Shell Capa intermedia del sistema operativo Unix, intermediaria entre el *kernel* y las aplicaciones.

Software Componentes intangibles de un sistema de cómputo. Agrupa *todos los programas y aplicaciones* empleadas en un sistema.

Tiempo compartido Modelo de sistema operativo en el que una sola unidad de proceso es compartida por varios procesos.

Tiempo real Mecanismos implementados en un sistema para asegurar la respuesta inmediata a las diversas solicitudes de ejecución de procesos que se le hagan.

Tolerancia a fallas Mecanismos implementados en un sistema para asegurar su funcionalidad a pesar de la presencia o aparición inesperada de fallas.

Tomografía Conjunto de procedimientos *mediante los cuales se obtienen* radiografías de secciones muy finas del cuerpo humano.

Variable Posición en memoria con nombre que se usa para mantener un valor que puede ser modificado por un programa.

VLSI Very Large Scale Integrated, Chips integrados de muy alta escala.

BIBLIOGRAFÍA

- MULTIPROCESSOR SYSTEM ARCHITECTURES.

  Cantanzaro, Ben. *Sun California*

- THE DESIGN OF THE UNIX OPERATING SYSTEM.

  Bach, Maurice J. Prentice Hall.

- SISTEMAS MULTIPROCESADORES.

  Gómez Pedraz, Salvador; Alvarez Sanz, Emilio. Paraninfo.

- OPERATING SYSTEMS.

  Harvey M. Deitel. Addison-Wesley *Publishing Company.*

- COMPUTER SYSTEM ARCHITECTURE

  M. *Morris* Mano. Prentice Hall.

- UNIX. THE COMPLETE REFERENCE.

  Coffin, Stephen. McGraw Hill.

- COMPUTER ARCHITECTURE AND PARALLEL PROCESSING

  Hwang, Kai. Briggs, Faye A. Briggs.

- DYNAMICAL SYSTEMS AND FRACTALS.

  Karl-Heinz Becker; Michael Dofler. Cambridge University Press.

- CHAOS AND FRACTALS: NEW FRONTIERS OF SCIENCE.

  Peitgen, Heinz-Otto. Springer.