INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Parallelism through Multithreaded Programming in a Constraint-Based System

by

Lawrence G. Muller

This dissertation is submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1997

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

4/30/97
Date

Professor Kenneth McAloon
Chairman of Examining Committee

5/1/97
Date

Professor Stanley Habib
Executive Officer

Professor Carol Tretkoff
Professor Gerald Meyer
Dr. Coskun Atay

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

Parallelism through Multithreaded Programming in a Constraint-Based System

by

Lawrence G. Muller

Advisor: Professor Kenneth McAloon

This research explores the applicability of multithreaded programming to constraint based programming. To that end, we used 2LP, a constraint based linear programming language whose framework has an imperative and declarative nature, and our current research extends the system to include language level control over parallelism. We also demonstrate the inherent scalability of multithreaded programming. In addition, insights into the issues of reorganizing a single-threaded system into a multithreaded system will be discussed. To benchmark success and failure, a number of applications are used; these include netlib problems such as set covering. This research is the first time a multithreaded constraint system has been implemented.

2LP has previously benefited from transforming a local area network into a parallel machine. This work explored distributed programming models using an embedded itinerary mechanism on the one hand and explicitly coded parallelism on the other. Further, this research demonstrated that significant load-balancing can be

obtained using cooperative threads. The hardware architectures that this research employes as a test-bed are two currently available symmetric multiprocessor (SMP) systems. The operating systems utilized are Microsoft's NT system, and Sun's Solaris system (including Sun's multithreaded library).

Acknowledgments

A special thanks to my advisor Professor Kenneth McAloon, and to Professor Carol Tretkoff, for without their encouragement, patience, and guidance this thesis may not have reached completion. Thanks also belongs to the other members of my examination committee Professor Gerald Meyer and Dr. Coskun Atay, and to Professor Stanley Habib, executive officer of the Ph.D. program in Computer Science.

I thank my wife Cheryl for her prayers, encouragement, and love, and my sons Jonathan and Michael for giving up daddy for a while. Lastly, but not least, I am thankful to God that it done.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

This thesis explores the multithreading of the constraint based programming language 2LP [1]. Employing multithreading techniques on a shared-memory system, we the developed a 2LP system that is parallel "portable" onto systems that support multiple threads. We then benchmarked both the threaded version and the "standard" sequential version on selected applications and, profiled performances on two distinct hardware architectures.

The use of threads is very strategic, in that, we believe that it can insure the "catching of the technology curve." Our system can capitalize on additional multiprocessor boards, and advancements in operating systems and libraries that support multithreaded programming.

This research is closely tied to applications, but is made in order to yield generic as well as application specific research results. Operating under the assumption that applications are a good starting point, in our research, we applied 2LP to problems in Artificial Intelligence (AI) and Operations Research (OR).

Linear Programming + Logic Programming = 2LP

The starting point of this research is the constraint-based programming language 2LP, which was developed at the Logic-Based Systems Laboratory at Brooklyn College, CUNY. It has characteristics of both imperative and declarative programming

languages. It is a constraint-based language with a C-like syntax and a top-down procedural aspect (i.e., $step_1$,..., $step_j$,..., $step_n$). However, unlike its fully imperative relative, C, 2LP has built-in chronological backtracking. A program's step will either fail or succeed. If a step succeeds then the next step is executed; however, if a step fails, the 2LP system will *backtrack* and attempt to re-evaluate the previous *choice point*. A choice point is a marked position in the 2LP's source code that execution can later back-up-to start anew. This is similar to application of `setjmp` and `longjmp` in traditional Unix C programming. It is within 2LP that we examine methods to create multiple threads.

2LP's strength is that through declarative methods (i.e., logic programming) a model builder gains the explicit ability to guide a search strategy. This is in contrast to systems such as AMPL, Lindo, and GAMS, where the modeler translates the mathematical formulation of a problem into the syntax of the modeling system. Such systems behave as *black-boxes* that are in control of the solution process, making it very difficult to embed "knowledge" about the problem's domain into the process. For a strictly linear programming (LP) problem, crunching the numbers through a *solver* (e.g., simplex algorithm) will often produce a quick solution, but this is not true for Mixed Integer programming (MIP) problems and disjunctive problems where a search is needed.

Earlier Work on Parallel 2LP

There has been earlier work on the parallelization of 2LP [2]. In this work an itinerary method was developed that provided a mechanism that was successfully adapted to a wide variety of high-performance computing platforms. Also, experi-

ments have been done using systems such as the distributed programming library (DP) on a local area network to tackle integer goal programs (PIG) [3]. These topics will be covered in chapter 7.

Why Threads

Threading a program essentially allows a process to execute multiple copies of itself within a single address space. Thus threads are a form of parallel programming, and have a number of characteristics that produce, what might be called, cleaner code. This is because the program can spin-off multiple threads of execution of itself, each with a private data stack, access to the process' context, unguarded access to global data and static variables and access to the processes' heap space. However, the programmer must deal with access control of shared variables, and synchronization among threads. An aspect of a multithreaded program is its inherent scalability, in that it can run as a single thread on one processor, or it can run as multiple threads in parallel---as processor boards are added to the system. Furthermore, it is suited for the symmetric multiprocessor (SMP) machine, although, a multithreaded program can run on asymmetric systems (ASMP) as well.

Chapter 2: Computer Hardware

In this chapter we present an overview of how a computer's architecture can be classified, with respect to parallelism, to support later discussions on how its underlying mechanics can affect a program's behavior.

Classification of Multiple Processor Machines

In the case of multiple processing systems, there are a several common terms that are used for gross classification. These terms can be used to classify a system based on its processing symmetry or the how main memory is attached to processors [6, 35]. A system is said to be a *symmetric multiprocessor* (SMP) if each of its CPUs are functionally equal in all respects, this usually means that the processors have uniform access to main memory (UMA), Interrupts, and Input/Output functions. The system is otherwise said to be an *asymmetric multiprocessor* (ASMP); for example, if in a four processor system, only one processor has direct access to I/O, the system is said to be an ASMP. An SMP system provides "better" performance than its counterpart and its implementation is much more complex [53]. The degree to which processors are able to communicate with one another, through memory, is often referred to as *coupling*. There are two extreme degrees of coupling. When each of the system's processors has equal access to main memory, the system is said to be tightly-coupled. Conversely, when processors do not have equal access to main memory, then the system is said to be *loosely-coupled.*

Flynn's Taxonomy

Flynn's scheme [67] for the classification of parallel computers is probably the most widely known [38]. It characterizes machine behavior based on the amount of overall parallelism in instruction and data streams. This is a high-level perspective of an architecture, and does not take into account low-level concurrency, or processors that are not used for computation. This high-level perspective of multiplicity in instruction and data streams yields four types of architectures.

Single Instruction Single Data (SISD) stream: In this architecture there is one sequence flow of instructions that acts on one data stream. The diagram below illustrates the instruction stream's "flow" into the processor, that controls a processing unit[1].

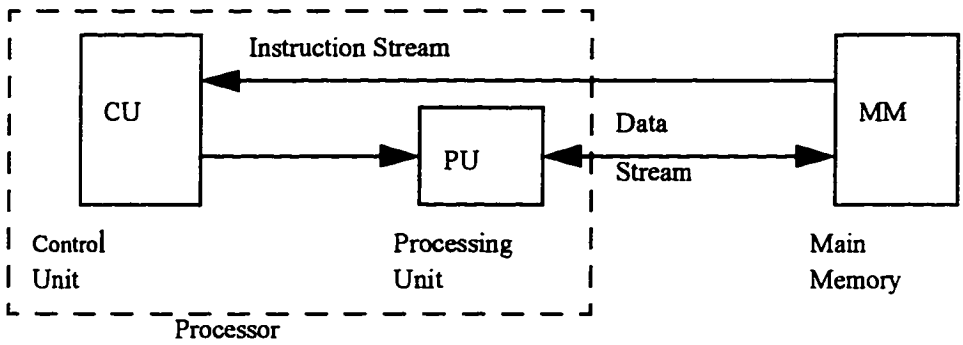


Figure 2.1: SISD

Single Instruction Multiple Data (SIMD) stream: An example of this type of machine is a vector, or array, processor where a single instruction controls multiple

1.Initially, Flynn classified a pipelined processor as a special case of the SISD machine; however, this position has changed, in that, he now states that given the application of pipelines in modern CPUs, "it is the pipelined processor--in which one operation is executed per state transition---that defines the SISD processor" [66].

processing units, that in turn can act on multiple data elements. In the diagram below, the control unit broadcasts an instruction to the processing units (PU), and each PU acts on different data elements from separate data streams.

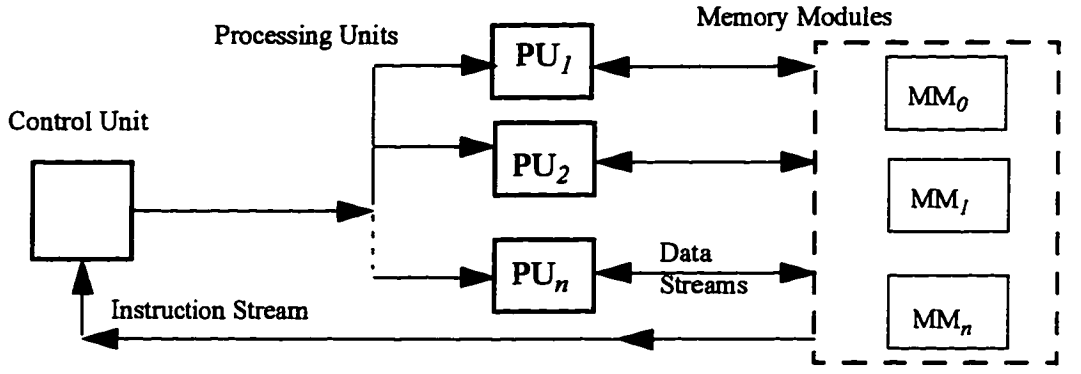


Figure 2.2: SIMD

Multiple Instruction Single Data (MISD) streams: In this scheme, see figure 2.3, multiple streams of instructions act on a single stream of data (see the diagram below). Operationally, it works as follows: program stream 0, causes control unit 0 to fetch data and performs some operations on it (possible the null operation), and sent its output is to control unit 1. As program stream 0 is executing so are the other streams, each accepting its data from the previous control unit and passing the result on, until the last control unit returns the final computation to the memory system.

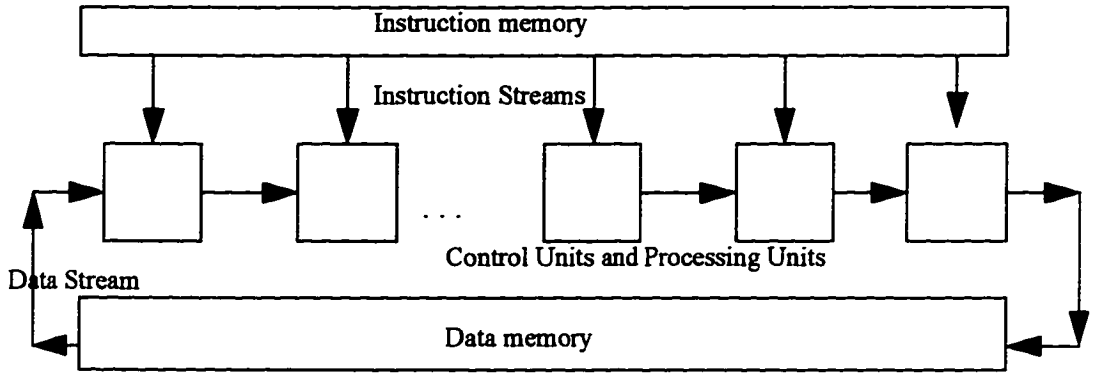


Figure 2.3: MISD

Multiple Instruction Multiple Data (MIMD) streams: This layout characterizes most multiprocessor systems, where multiple programs act on multiple data sets (cf. below). If the interconnection among the processors is loosely coupled or the data streams are disjoint data sets, then the derived MIMD machine is called a MSISD, that is, multiple SISD machines. However, intrinsically the MIMD machine implies an interaction among its processors and the sharing of data sets.[38].

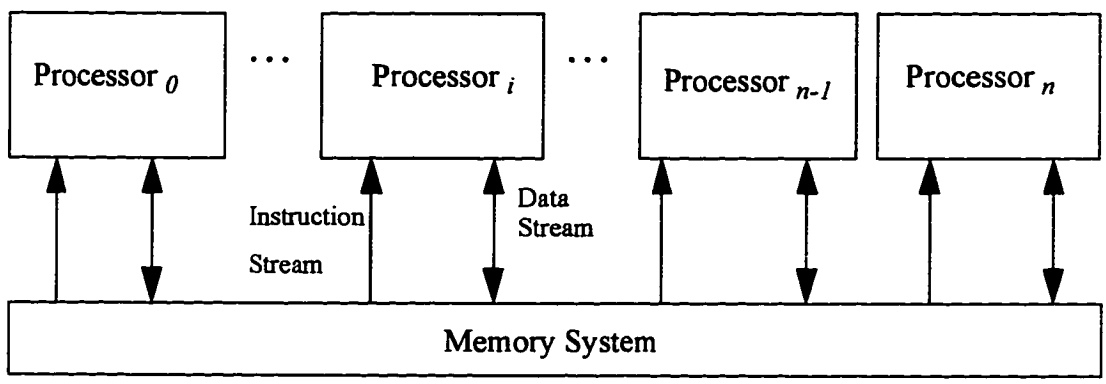


Figure 2.4: MIMD

Hardware Structure

The experiments in this thesis use two different 4 processor SMP machines: a

dual-boot 4 Pentium box that can run under either NT or Solaris, and a 4 CPU Ultra Sparc box under Solaris. While both machines are SMP, their underlying hardware is different. The Pentium based system is based on a shared *bus* technology, and Solaris system employes a *crossbar*. These two distinct hardware structures are used to measure the effectiveness of multithreading 2LP's run-time engine. This following sections are meant to provide a synopsis of generic system features. The first to be discussed is a multiple CPU system employing a time-switched bus to access shared memory. To characterize a multiprocessor bus-based system, we will begin by exploring a uniprocessor system, then extend the system to include multiple processors. The second discussion will focus multiprocessing systems that use a crossbar switch to access shared memory.

Time-Switched Bus System

Essentially a timed-switched bus is a form of *time-division multiplexing* (TDM). This technology employs a shared bus to interconnect shared system components. At the hardware level a bus commonly refers to a group of related wires; for example, a group of data wires would be called a data bus. A bus can be either dedicated or multiplexed. As an example of a multiplexed bus, Intel's 8086 microprocessor time-sliced a portion of its data bus onto the address bus, this was done in order to reduce the pins required for the 8086 microprocessor integrated circuit [47]. Commonly, however, bus-based computer systems have three types of buses: data, address, and control. It is through these three bus types that a processor can access memory or I/O devices. To ensure that the bus is used effectively, there are rules that all devices must follow. By using a bus arbiter (i.e., moderator) devices signal their need of access

and await acknowledgment for access. There can be any number of buses in a system, to interconnect various subsystems. Many desktop systems, for example, typically have a proprietary bus, that is essentially inaccessible to the user and is used by the CPU and other system components. This bus is often referred to as the system's internal (or private) bus. [41,44,45,48].

Interface Buses

In addition to the internal system bus, there are public buses that provide industry standard interfaces to the system; for example, ISA, PCI, and Sbus are industry standard buses that provide a means to add new hardware to a computer.

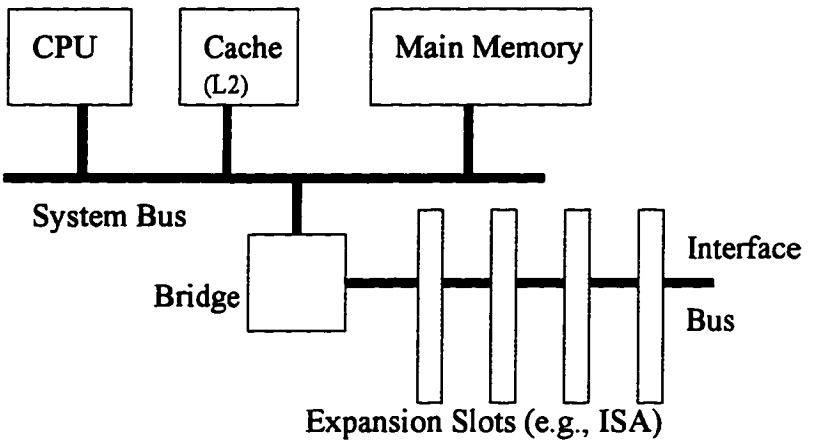


Figure 2.5: Bus-based system lay-

The diagram above shows the basic layout found in the popular *bus-based* computer systems. It contains a central processing unit (CPU), an external cache (L2 cache), main memory, and a bridge that connects the system's bus to a *public* bus that provides an interfaces to expansion slots.

Uniprocessor system

As the name implies, a uniprocessor system has a single central processor. In this case the CPU also acts as a bus arbiter controlling access to subsystem components on the bus. The following diagram illustrates a bus-based system. It consist of a central processing unit (CPU), memory, N devices (or device interfaces), and a direct memory access controller (DMAC) that specializes in the transferring of data during input-output (I/O) operations. Notice that all devices need the bus to communicate. In this system the bus arbiter (sometimes called a bus master), governs access to the

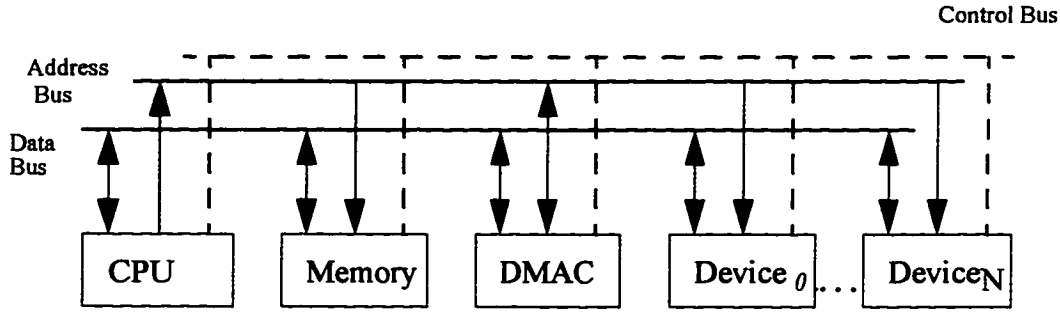


Figure 2.6: Bus-sharing

bus by initiating the required control signals. Other devices interface onto the bus as bus slaves.The control signals from the CPU provide each of the other devices such as main memory, with commands and timing information. For example, to read data from memory the CPU places the data's address on the address bus. It then signals memory, through the control bus, that there is a read operation being requested and that the address bus contains a valid address. Devices respond to the control signals on the buses, and will do nothing if it is not their address on the address bus. It is through the CPU's mastering of the control bus that devices cooperatively share the data and address buses.

Although, in our example, all control signals were negotiated though the CPU, some devices can initiate their own control signals. One class of signals that request the CPU's attention are called *interrupts*, and another signal type may request that the CPU relinguish control of the buses. Interrupts will be discussed in some detail in chapter 6. Later in this chapter we will discuss the DMAC to illustrate a request for bus control. This is an inherently dangerous operation, in that, a DMA controller is given direct access to main memory. This has the potential to corrupt the consistency of main memory, since the CPU is unaware.

Multiprocessor Bus System

Extending the uniprocessing bus-based system to a tightly-coupled (or loosely coupled) multiprocessor system appears straightforward. To accomplish this, however, a number of other things need be taken into account. These include the bus arbiter, data and instruction caching among processors, and memory access control.

Bus Arbiter

Recall that in the single processor system, the CPU could act as the arbiter. However, now that the system has grown, the generic CPU is not equipped to efficiently support an arbitrary number of devices requesting bus access; thus additional control is needed [47]. Bus arbitration can be centralized or distributed. In the diagram that follows, figure 2.7, a centralized control of the system bus is maintained by a single bus arbiter, also called a bus master.

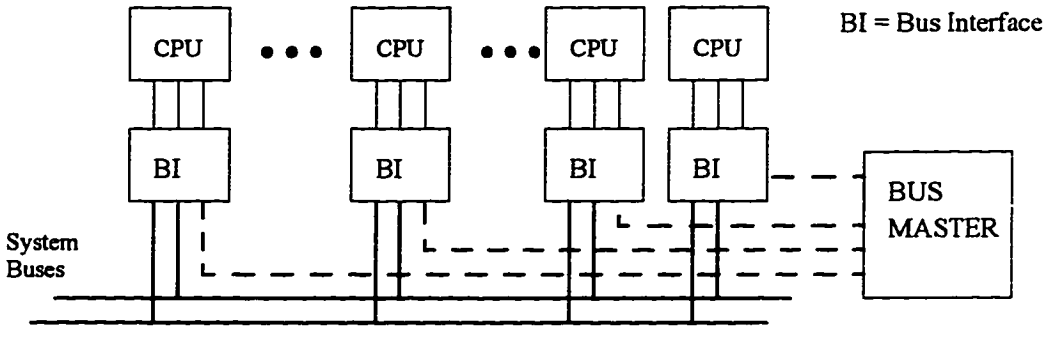


Figure 2.7: Bus-master

Dual Pentiums

As an example of distributed bus control consider that some models of Intel's Pentium processor (figure 2.8) series incorporates a private CPU-CPU bus arbitration protocol that enables dual processors to contend for bus control, without the use of an external bus controller [49]. This bus arbitration scheme takes into account a number of criteria that include efficient use bus bandwidth, fair access to the bus, support of inter-CPU pipe-lining, and the avoidance of the introduce of dead-lock.

During system initialization (i.e., system reset), the CPU in socket 7 checks for a processor in socket 5, and then performs a boot sequence. This initially places the CPU in socket 7 as the *primary processor* and the CPU in socket 5 as the *dual processor*, or *secondary processor*. In spite of their names, the CPUs have symmetric processing power, this includes access to memory, cache, interrupts, I/O memory, DMA, and system buses.

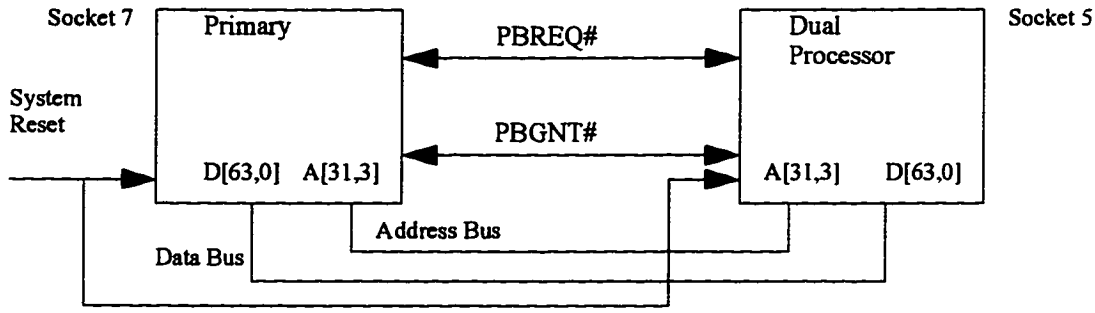


Figure 2.8: Dual Pentium bus negotiation

Access to the system is negotiated through the *bidirectional* control lines PBREQ# and PBGNT#, *private bus request*, and *private bus grant*, respectively. The CPUs cooperatively "toggle" between being in *primary* and *secondary* modes, with the primary being in "control" of the system. Operationally, the current primary has exclusive access to the system's buses, and the secondary must request ownership via the assertion of the its PBREQ# pin. When the primary completes its current cycle, it gives ownership of the buses to the LRM by asserting the BPGNT# pin. This then reverses each processor's role, that is the primary becomes the secondary, and the converse, the secondary becomes the primary. Note that the bidirectional semantics of the PBREQ# and PBGNT# pins always have the primary's BGNT# pin as an output, and the secondary's BPREQ# pin as an output [52]. Also, there are other aspects that are beyond the scope of this discussion such as how the processors take into account the current state of the system bus and the cache.

Increasing Throughput

To increase a system's ability to process information, a number of techniques

exist. We will take a brief look at three common methods, one that clearly illustrates bus sharing (i.e., DMAC), but is sufficiently generic to apply to a crossbar system. The other two improve memory access (i.e., cache storage and memory modules).

Direct Memory Access

Consider the following programming scenario. A process opens a handle to a file for input, and later requests (e.g., `fscanf`) to read data from the file. When the file system has gained access to the file, the file's data needs to transferred to main memory. In general, an application program cannot directly manipulate data unless it is in main (or cache) memory. While data is being transferred to main memory, the application program must typically wait. Clearly accelerating the transfer process will speed program execution. In the pervious example of a CPU reading data from memory, the CPU did not relinquish control of the bus. To increase I/O throughput, bus control is turned-over to a dedicated processor. This processor, commonly called a direct memory access controller (DMAC, or DMA channel), has one function, and that is to transfer "raw" data to and from an I/O device and main memory. This is more efficient than such transfers under program control. To see this improvement in efficiency, consider what the processor must do to run an I/O program from main memory. There would be activity on the buses as the CPU fetches I/O program instructions from main memory. There would also be activity on the buses transferring data from an I/O device to main memory. In addition most systems would need to use the CPU's internal registers as an intermediary in data transfers, see figure 2.9.

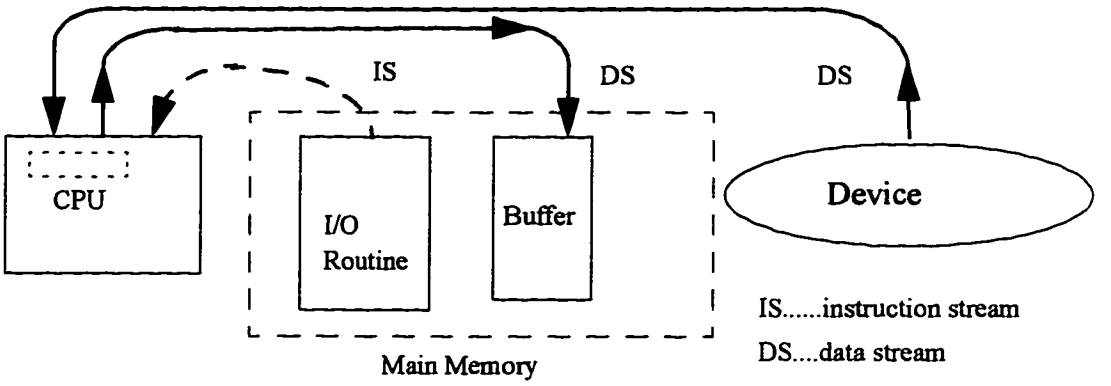


Figure 2.9: Programmed I/O

The following five steps illustrate the essential operation of a direct memory access controller used to transfer data from a device to main memory [13].

1. An I/O routine initializes a DMAC by giving it the addresses of a main memory block that it can use to store data from the device the DMA will access;

2. When an input device attached to the DMAC (cf. above) has data to be transferred to main memory it signals its DMA controller.

3. The DMAC sends a Bus Hold request signal to the bus arbiter (in this case the CPU);

4. The CPU completes its current bus cycles, disables its hold on the bus, and then acknowledges the DMAC's hold request on the bus. The CPU will not attempt to use the system bus again, until the DMA releases it by dropping its Bus Hold request;

5. The DMAC now has the bus and makes the transfer(s).

The transfer is faster in that the DMA controller has its instructions hardwired, and does not need to access an I/O routine in main memory, thus removing one set of bus cycles. Further, a DMA that is integrated on the I/O device can directly copy data to and from the device and main memory, and memory to memory. This is possible because the I/O device's "buffers" are the DMAC's registers, so it has direct access, thus removing the CPU's need for a device-to-register-to-memory bus cycles. Thus the CPU is unaware of changes being made to memory.

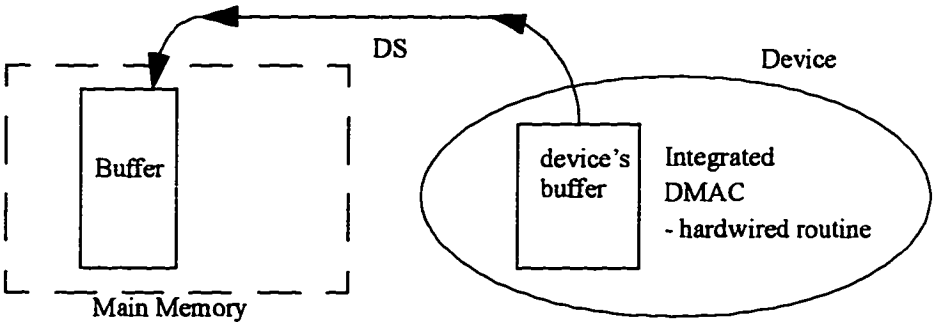


Figure 2.10: DMA I/O

Memory Cache

The common CPU is many times faster than inexpensive main memory (e.g., dynamic random access memory); in fact, many systems have required the insertion of one or more CPU wait states so that the slower random access memory (RAM) can respond. To reduce this bottleneck, faster, more expensive RAM is placed between slower RAM and the CPU. This memory is called *cache memory*. Operationally, the CPU queries the cache for a memory location (either data or instruction). If the item is present, it is called a *hit*, and the cache quickly sends over a copy of the item to the CPU. If the item is not found within the cache, it is called a *miss*, and the item must be fetched from slower main memory. Once an item is fetched from main memory, it is

placed in cache and taken by the CPU. To make the process more efficient, blocks (or lines) are transferred to the cache. This scheme can speed-up overall program execu-

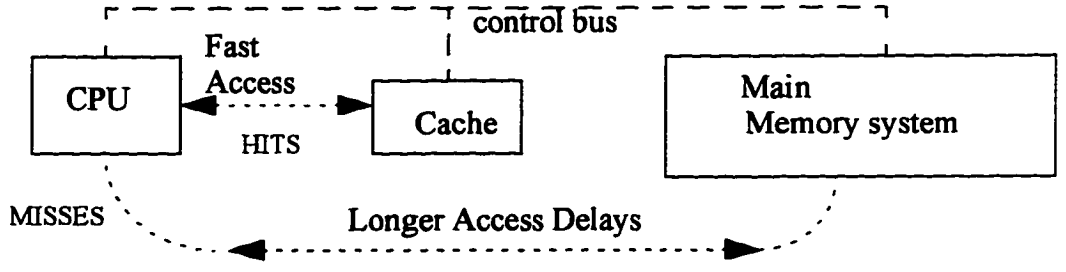


Figure 2.11: Cache hit-miss diagram

tion, because "well-behaved" programs typically posses a property know as *locality-of-reference*.

Locality-of-reference

Accumulated profiles of computer programs have shown that they tend to frequently reference small regions of themselves, and this phenomenon is known as locality-of-reference [42]. An explanation of this phenomenon is that programs do not generate random memory requests; they are often part of a loop, a block-formatted control structure, or a computation (e.g., `Y=2*Y*Y+3*Y`). There are three components that can be used to characterize a program's cache demands: *temporal*, *spatial*, and *sequential*. Each component influences the characterization of a memory system's design.

- Temporal: references near elements from the recent past.
- Spatial: references the address space in the neighborhood of the last reference.
- Sequential: if the last reference was to location M_i, then there is a likelihood that

the next reference will be M_{i+1}.

Current PCs, workstations, and servers have caches that range from 512Kbytes to 2M bytes; for example, DEC's Alpha Server 300 supports 2M bytes of L2 cache. Some of this increased size is likely due to the decline in hardware cost, but another important factor is the types of application systems that are being used. For example, modern programming has become heavily burdened with graphic user interfaces (GUI) and demand much more memory than the simple console interfaces of the past generation of applications. Work has been done on the scheduling of threads and the locality of reference [72], the fusion of loops for code locality [73], and code optimization for threads [10].

Memory Modules

The partitioning of a computer system's memory impacts its performance. In the typical (von Neumann) computer model, main memory is a bottleneck. Both program instructions and data reside in main memory, and both must transverse the same bus to and from the processor [36,37,40]. To accomplish this the bus is time-sliced between fetching instructions and accessing data. Given a memory technology, there is a performance gain by partitioning memory it into independent modules. While there are a number of design techniques we will highlight, in a general way, the efficacy of partitioning memory.

There are significant delays when writing to and reading from memory. A processor's demand on memory can far exceed the memory's ability to respond to the demands for data or instructions. To improve performance there are interleaving techniques such as S-Access [42]. This method divides the memory into *memory banks* in

such a way as to "anticipate" a processor's next request. It works because of the same locality-of-reference principle used by caches; for example, if a processor request address 0x00f0 it is likely to need 0x00f1 next. This is accomplished by storing contiguous strings of data and instructions across memory modules.

To understand what S-access is, let us take, as an example, a memory system that has been partitioned into 4 memory banks. Using an S-access memory design, the system does not directly use all N bits in the address bus to access the memory bank directly. Instead, it uses $log_2(4)=2$ least significant bits (LSB) of the address bus to distinguish among memory modules. The address bus' remaining $N-2$ bits are then fully decoded by memory to concurrently select an actual address simultaneously on 4 memory banks.

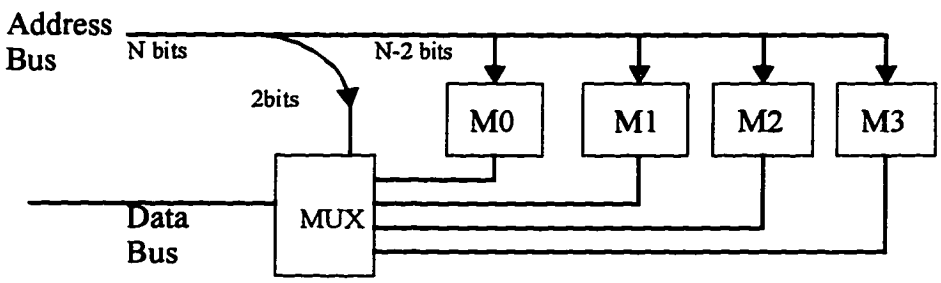


Figure 2.12: Memory banking

This mapping provides 2^N addresses, the same as if there were a unified memory bank. Operationally, when a processor ask for location 0x00f0, the memory modules respond to the $N-2$ bits presented on their address bus. The 2 least significant bits of the address bus are directed towards a multiplexer (MUX). To present the processor with the correct location, the multiplexer uses the 2 bits that it was given to select the

memory module that corresponds to the requested address. Thus when a processor request location 0x00f0, all 4 modules will be stimulated, and the memory system as a whole will have the addresses 0x00f0 through 0x00f3 at the ready. Thus subsequent processor requests to these locations can be satisfied quickly.

The starting location of a datum influences a memory system's performance. This location is known as a datum's alignment. In the pervious example, a four byte integer is better stored in addresses 0x00f0 to 0x00f3.. To understand why, consider the work the above system must do if the integer were aligned at location 0x00f2. First, the system will need to access address 0x00f0. This entails access to addresses 0x00f0 through 0x00f3 to retrieve the first byte of the integer. Again, the memory sub-system must be accessed to get addresses 0x00f4 through 0x00f7. In summary, two memory accesses were required because of the misalignment. Furthermore, many older processors can only access a datum that is aligned to specifications. Compilers will often align a datum on a specific memory boundary to permit a processor its "natural" access to memory. This may include skipping some locations between data. As an illustration, consider a one byte character and a 4 byte integer. When placing the variables in memory, it is more efficient to align the character at 0x0000 then skip locations 0x0001, 0x0002, and 0x0003, so as to align the integer at location 0x0004. While three bytes of memory are "wasted," this alignment requires only one memory access for each variable.

Caching Among Processors

There are two basic approaches to multiprocessing system cache design, for a survey of schemes see [64]. One method is for processors to either share a common

cache (as with Intel's dual Pentium architecture), another is to dedicate a cache to each processor. In bus-based systems a shared cache increases bus traffic and thus limits the system's scalabilty. That is, as processors are added to the system, the additional bus traffic generated by the use of a single shared cache can degrade performance by saturating the bus. However, in dual processor systems, a shared cache is beneficial. A dedicated cache while increasing circuit complexity and cost, does, however, provide for system scalabilty. This is accomplished by keeping transfers private between a CPU and its cache.

In addition to an external bus cache, the typical CPU has one to two internal caches, that is, a small region of chip real-estate that is used to cache data and instructions. While caching will statistically improve processor throughput, problems arise in multiprogramming. Consider what might occur between two programs each sharing a variable.

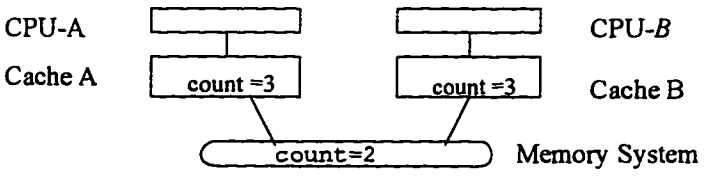


Figure 2.13: Variable sharing

If program A is running in CPU-A and program B is running in CPU-B, then it is possible that each of the CPU caches will have a copy of the shared variable at the same time. Thus, each program can be independently changing that variable, but the purpose in sharing implies some communication of the act of modifying the shared variable. Thus any change to variable in one cache must be reflected in the other cache(s) in

order to be consistent. This is known as the *cache consistency problem*. In addition to the consistency of data among caches, there is also the need to reflect that consistency in main memory

Controlling the Cache

The cache of many systems is controllable. For example, a modern system can be programmed to allow any area of its memory to be cached, and can also be programmed exclude specific areas from being cached. Excluding or including memory from the cache system can be accomplished through hardware or software. In the Pentium system, for example, the processor has a dedicated cache enable input pin (KEN#) that is designed to work in conjunction with a cache controller. If during a read-cycle, the controller deactivates its KEN# pin, the processor will bypass the cache and perform a direct read from main memory, thus excluding the cache. In software, there is a bit that can be set for a page table that will disallow caching of that page's region of memory: this bit is called the *page cacheability disable* (PCD) bit in the page table entry. In fact, for a line of memory to be cached by the processor, both hardware and software must be in agreement [46,47,48].

Cache Consistency

It is important that a multiprocessing system maintain consistency of data among its caches, CPUs, and main memory. This is a difficult problem, and there are a number of solutions, each with its own costs and benefits. One fundamental requirement of a cache consistency solution is that it be transparent to the application software [46].

Memory Access Control

A memory subsystem need not follow the accesses of a process' programmed order. That is, for efficiency sake, the memory subsystem may reorder program reads across CPUs. Also, systems that have write-buffers and use a cache can sometimes lose the ordering of memory across processors. A system that enforce access ordering is referred to as a *strongly ordered* system, and as a consequence imposes restrictions on optimization. *Weakly ordered* systems, however, remove some access restrictions, allowing designers greater flexibility for optimization. Weakly ordered systems, unfortunately, create software compatibility and portability problems [46]. The problem of strongly ordered and weakly ordered systems deals with what is called a system's *memory consistency* model.

Memory Consistency Model

A memory consistency model is the specification of how memory appears to a programmer. The specification of a memory model affects all of a system's interfaces, from memory, cache, and processor design to programming applications. The model that is used also affects a program's portability across platforms that use different models [63]. Thus the rules that govern a system's memory model are important and have been studied in [61,62].

In a uniprocessor system, a program's last write to some location X should be the value returned when the program next reads from that same location X. Adherence to this rule is known as *program order*. Symbolically at time t, let $W(x,y)_t$ mean a write to location x the value y, and $R(x)_t$=y means that at time t, the value y was found

in location x. Thus program order can be expressed as:

if $W(x,y)_t$, then $R(x)_{t+1} = y$.

Program order is not inherent in computing systems. Consider a system that uses a multi-level switched network to connect its processors to memory. In such as system, it is possible that a $W(x,y)_t$ will be buffered at some level because of congestion, and the read request, $R(x)_{t+1}$, will route through an free path, producing $R(x)_{t+1}=z$, and z is not a result of $W(x,y)_t$. Thus the desired program order was violated. A multiprocessing system can enforce program order in each processor, but this does not imply that memory access is consistent across processes sharing a variable [42].

To illustrate the point consider the following diagram that employs a multi-level switch to connect CPUs to memory. Further, assume that the system enforces program order for a process. Now, consider two asynchronous processes, one on CPU-A and another on CPU-B, that share the variables x and y. In the diagram, CPU-A produces two ordered writes to memory, with CPU-B ordering two reads from the same memory locations. In detail, first, CPU-A perform a write operation to memory location x, W(x,1), at *time+0* followed by another write to memory location y, W(y, 1), at *time+1*. As the two *memory-writes* travel through the switch, the write to location x is temporarily buffered, and the second write makes its way unobstructed to location y. At *time+1* CPU-B initiates a read from location x, R(x), followed by a read from location y, R(y), at *time+2*. Thus if the program in CPU-B was waiting for variables x and y to change in a particular order, it runs the risk of not correctly "seeing" the data change in the intended order. This will be discussed as viewed by a programmer in chapter 5.

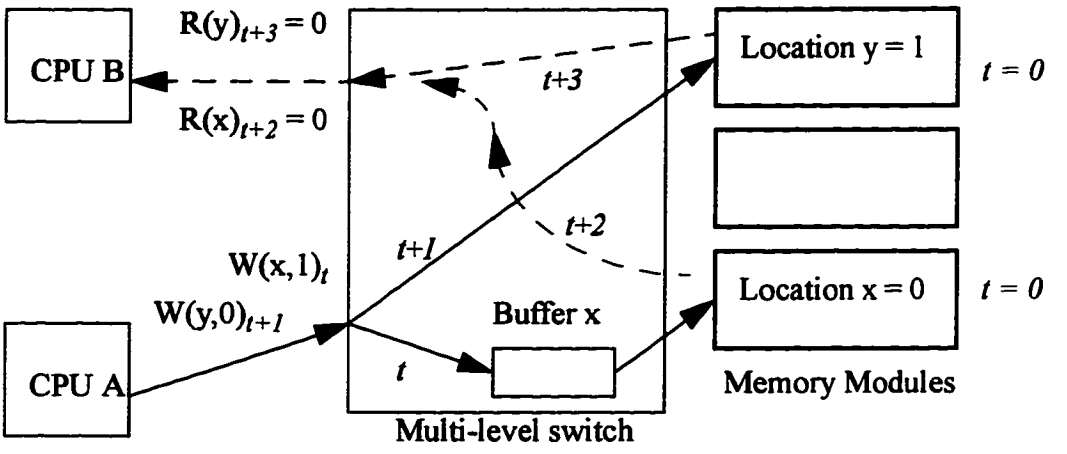


Figure 2.14: multi-leveled switch delay

Many high-level languages such as C allow a programmer to assume that all memory operations are in *program order*. For the purposes of optimization, some compilers and hardware systems re-order memory operations. In fact, Intel and Sun warn not to depend on future systems being strongly ordered, that is, that operations will maintain their ordering across processors.

In a multiprocessor system there is a need to extend the program order model so that it is made manifest across processors. The extension is called *sequential consistency* [63]. In a multiprocessing system, each processor's program observes its own memory accesses in program order, but another processor attempting those same memory accesses (as in shared memory) may see something different (weak ordering). When multiple processors see, in effect, a serialized access to shared memory, this is called the *property of observability* [42]. The assumption that memory operations have the property of observability in writing to memory is not necessarily valid, and many commercially available shared memory systems violate this property [63]. A well

known program that is dependent on system's memory model is Dijkstra's solution to process synchronization, since one of his underlying assumptions was the observability property. Historically, the debate at the time was whether it was at all possible to synchronize processes using a common programming language such as ALGOL 60. A salient point here is that many programmers are usually not aware when their code requires the observability property [42].

Multiprocessor Crossbar system

A crossbar is an alternative to a shared bus to connect multiprocessors to a shared main memory. It is a device that provides, in effect, dedicated point-point communications. It is accomplished through reconfiguring a matrix of switches. This is a costly technology that is used in medium to "high-end" computers.

Crossbar Switch

A crossbar is an interconnection scheme that makes it possible for all processors to connect to main memory without contention. In fact, unlike a bus that is time-shared, a crossbar can provide dedicated connections. The following diagram outlines the basics principle behind the crossbar's ability to connect multiple memory modules

to multiple processors.

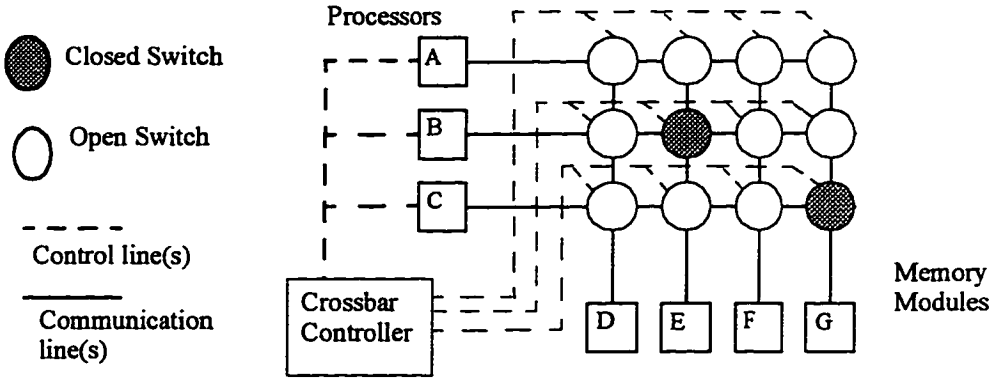


Figure 2.15: Crossbar diagram

As shown above, a crossbar switch can be viewed as a matrix of horizontal and vertical lines whose points of intersection can be "opened" and "closed" by a central controller (or arbiter). Each horizontal and vertical line represents a possible communications path. An opened intersection means that there can be no information flow between its corresponding horizontal and vertical line. For instance, in above diagram, only processors B and C can communication with memory modules E and G, respectively. While the name *crossbar* may evoke visions of an electromechanical switching matrix, as in the "old" telephony systems in [70], its current solid-state counterpart can be found on micro-chips [69]. The crossbar is a type of network connection scheme that allows a direct connection between any free *source* and *destination* pair. This does not mean that crossbars are not without contention, that is, when two or more sources request the same destination some form of arbitration is needed. For a small count of source and destinations that need to be interconnected, a *fully connected crossbar* is ideal; however, its cost grows $O(n^2)$ and thus is not feasible for a "large" N. To reduce the cost, but still provide connection capabilities, a number of multistage schemes

exist [42,35,74]. These schemes are not without performance costs; for example, some connection combinations will block others. This type of network is referred to as *blocking*. Some schemes may require that a data item be *recirculated* through the network an number of times until it reaches its destination. For example, IBM's Scalable Powerparallel System (SP2) uses a multistage interconnection scheme to interconnect its processor nodes [52].

An operating system can take advantage of its underlying hardware to selectively place a process' working set in a single memory module, this is known as a process' *home memory* [38]. Thus, when a process has multiple threads of control, each thread can run on a home processor accessing the same memory module, reducing memory access contention across processes. Another alternative is to attempt to uniformly distribute memory references across modules to reduce contention. Other studies have been wider in scope; for example the Digital's AN1 computer network, studied a LAN composed of crossbar switches [69].

Hybrid Systems

Cluster hybrids are now beginning to appear on the "popular" market. An example is Hyundai's adaptive memory crossbar (AMX) architecture [51]. It is based on the Intel Pentium-Pro processor. The AMX architecture interconnects two clusters

of bus-based processors. The crossbar separates the two bus subsystems, allowing

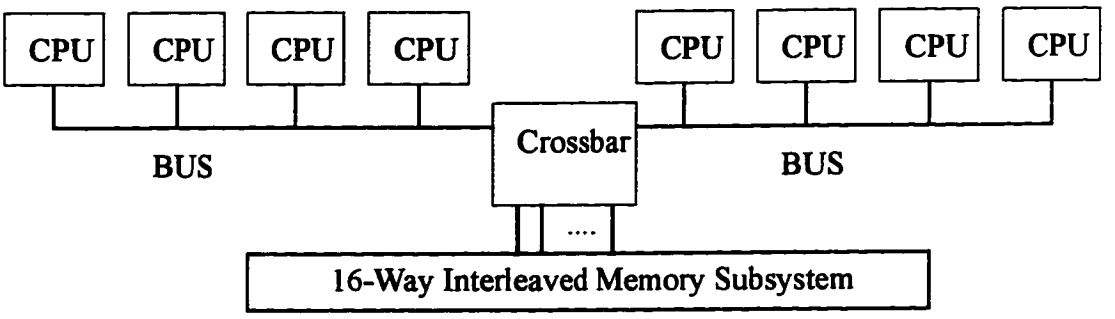


Figure 2.16: Hyundai's adaptive memory crossbar

each bus to operate at its rated speed (i.e., 533M bytes/sec for Intel's P6 bus). Thus the crossbar supports 2x533 Mbyte/sec (or 1.066Gbytes/sec) to memory. To help sustain this high data rate, the memory subsystem has been partitioned into16 interleaved memory banks.

On the "high-end," massively parallel processing (MPP) is arriving on the market that now uses a shared-memory paradigm, with built-in support of cache-coherency; for example, HP/Convex's Exemplar [65]. This system provides support for distributed and shared memory. It consist of up to 16 SMP clusters that are interconnected through four *ring* networks. Each cluster consist of 4 pairs of RISC processors with each pair sharing 64MB of RAM and a 1MB L2 cache. The pairs within a cluster can communications via a small crossbar that is private to the cluster, see figure 2.17 on the following page.

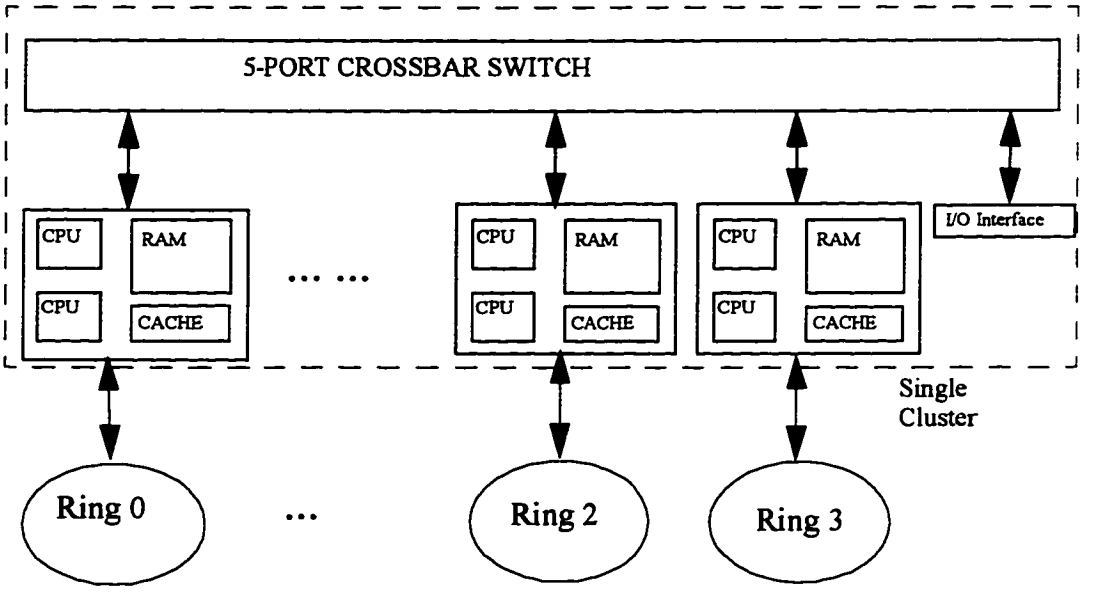


Figure 2.17 HP/Convex

Chapter 3: Threads

A processor reads a program by fetching its instructions and executing them. It is this path of execution through the running process that is referred to as a thread (or thread-of-control, or flow-of-control) and every process has at least one (i.e., the main function)[18,19,21,22,23,24,27,56,57]. Reading a process' code does not alter it[1]. Thus multiple processors can concurrently read through a program. In fact, the concurrence can be real or logical; that is, the number of actual processors need not equal the of number threads. Excluding the problems of deadlock and blocking the application of a time-sharing mechanism (e.g., round-robin) can guarantee all threads the opportunity to execute.

What is a Thread?

A program has the following basic components: a code section, data section, and a combined heap and stack section. Using Intel's x86 assembler language [43,44], we show the division between a program's code (also known as its text) and its data section. The division between code and data is clearly demarcated by the segment headings below.

1. Excluding self-modifying code

```
.data segment
   count word '0'/* count's offset is zero, wrt its data segment */
.end segment

.code segment
   mov bx, @data
   mov ds,bx              /* move data segment address into ds reg */
   ...
   mov ax, ds[count]      /* segment address=ds+offest(count) */
   inc ax                 /* add one to register ax */
   ...
.end segment
```

To execute a program, a processor ostensibly starts at the beginning of the code segment and incrementally steps through the code using a program counter to maintain its place in the program. When an instruction references a data item in memory, its addressing mechanism will either be absolute or relative.

Relative Addressing

A relative addressing scheme is diagrammed below. It employs a base address plus a relative offset to the actual data item in the memory. The mnemonic code to add var1 and var2 is: `ADD ds[0],ds[4]`. Notice that the actual memory addresses are not within the code. In fact, the actual locations in memory of code and data are not known apriori, and are constructed by the processor during instruction execution.

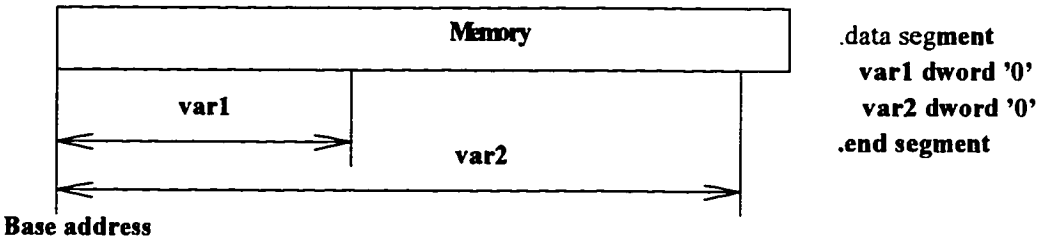


Figure 3.1 Relative Addressing

For example, assuming that a dword (double word) is 4 bytes, var1's offset from the base is zero (0), and var2's offset is four (4) bytes. Thus, if a base address in register ds is given as 0xFF00, then var1's physical address will be 0xFF00+0, and var2's physical address will be `0xFF00+4=0xFF04`. This scheme, and its variants, provide an easy means to reallocate data within memory without modifying actual code.

Absolute Addressing

Absolute addressing, also known as direct addressing, requires that complete addressing information be part of the instruction; for example, the mnemonic `ADD var1, var2` will be required to assemble into `ADD 0xFF00, 0xFF04`, in contrast to relative addressing. Thus when a program uses absolute addressing, its data must be loaded into specified memory locations only; for instance, if `var1` was actually placed in location `0x001`, the code would not operate as intended. To correct this situation, the program's instructions that reference memory need to be modified, or some other mapping scheme needs to be applied such as the address translation tables that are used in virtual memory systems.

A Process' Context

The full context of a process in a multiprogramming environment contains all the information required to situate it in the system. The form of the information is hardware and operating system dependent. In general, it consist of such items as a task state segment, state segment descriptor, task register, gate descriptor, local, global, and interrupt descriptor tables, process entry table, save areas, and a number of process specific items [12,13,31]. A process context is the operating system's view of a process,

and because of the amount of information required for a complete view, it is costly to switch among full process contexts. It is this cost of a complete context-switch that has driven the trend toward multithreaded programming.

A Thread's Context

The smallest amount of information about a program's execution state, or context, is contained within its CPU's register set that includes the program counter, general purpose registers, data segment register (ds), and the CPU's condition code flags. Using part of Intel's x86 register set as an example, the diagram below illustrates how the various registers keep track of a program's execution.

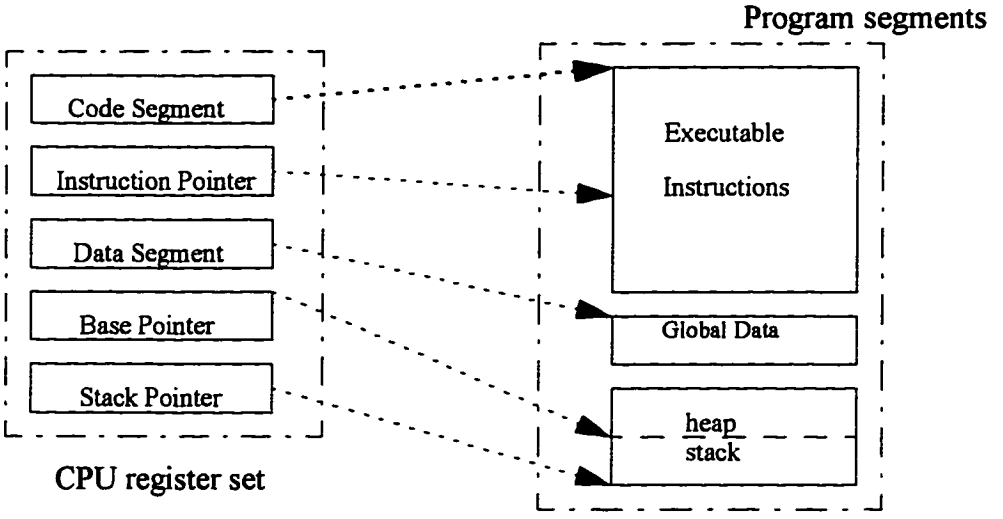


Figure 3.2 Thread Context

Tracing the execution of a program through this simple context is known as the program's *thread-of-control*, or *thread*. It is important to note that a program's context is essentially a snap-shot of the CPU's view of a running program. If a program is written in such a way as to allow multiple threads-of-control, the program is said to be *multithreaded*. For example, it is possible for a single CPU to switch among multiple

contexts of code to provide the illusion of concurrent independent running instances of the code. This is know as *logical concurrence*.

```
/* context 1's data segment */

.data seg1
    count    dw  '0'
    inputbuf dw 256
    ouputbuf  dw1024
.end seg1
```

```
/* context 2's data segment */

.data seg2
    count    dw  '0'
    inputbufdw  256
    outputbufdw1024
.end seg2
```

The only difference in the above two segments is that they are two separate memory segments. Thus the previously shown code fragment can essentially execute the different segments as though there were two complete and independent versions of the program in memory by switching data segments. If there are multiple CPUs simultaneously running various contexts of the same program, then each instance of the program would actually be running in physical (real) concurrence.

Reentrant Code

Another issue in programming is whether a function is *reentrant*. We will say a function is active if it has not yet returned, and it is reentrant if records of multiple activations are maintained. For example, a thread-A enters function `foo`, and prior to its completing and exiting `foo`, another thread enters; then if this second thread can destroy the previous CPU register set or any other pertinent data, the function `foo` is said to be *non-reentrant*. A common solution is to pass function parameters on a stack and to allocate all local variables on the same stack [47]. This way, making code reentrant be-

comes a simple matter of using a separate stack segment for each thread.

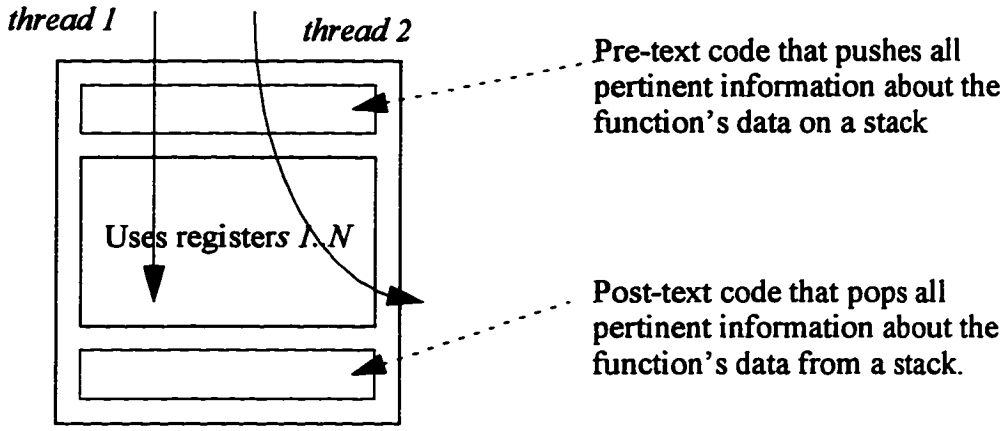


Figure 3.3 Reentrant code

Sharing Memory

The concurrency mechanism discussed thus far is not unlike that used in modern operating systems to run multiple instances of a program. Consider 100 users running the same application (APP). In terms of memory usage, it is more efficient to map 100 unique data segments of an APP into 1 copy of its code. Each instance of the APP is technically a thread. However, we distinguish this example from a multithreaded program in that each of the 100 aforementioned instances have their own full process context. In contrast, had the 100 instances been under a single process context, then we would say that the program is multithreaded. Here lies an important distinction. Threads that are parts of the same process context, share its address space. Thus each thread will naturally have access to its process' global data.

Light-Weight versus Heavy-Weight Processes

In a multitasking system, that is, a system that switches among processes, each

process requires a full context switch: these processes are called *heavy-weight* processes. In contrast, a *light-weight* process does not require a full-switch. A full context is a structure that encompasses a program's thread context and a structure(s) for the various protection schemes and resources that the system affords among its processes.

Historical Perspective on the Terminology

To achieve some form of parallelism, either physical or logical, and avoid the price of a full context switch, there is the light-weight process (LWP) or thread context. Historically [7], a LWP was a user-space library entity that captured a process' main thread, and created a new thread within the current process' context.These libraries were not kernel supported. Thus the only context switch was the one created by the library. For example, a non-preemptive library would most likely use C's `setjmp` to capture a context, and `longjmp` to restore it. In such a library, the operating system views a single process and thus allows it to execute as one process and there is no possibility of physical parallelism. This is because the Operating System will never give the process/library another thread.

Temporal parallelism allows a programmer to solve problems that are not easily solvable using conventional programming techniques. Traditionally, programming problems are seen from a functional perspective; that is, first step A, then step B,..., step Ω. On the other hand, multithreaded programming introduces a temporal perspective; for example, in no particular order, execute the following A, B, ..., Ω. To see how these two perspectives operate, consider the design of a simple word processor that permits editing text while text is being printing. Functional decomposition allows either text

printing or editing, but not both; and lengthy print-out can frustrate a user wishing to continue editing. A quick solution is to decompose the process into an edit-some-stuff and print-some-stuff loop. Such a fix can provide an acceptable user interface. However, in terms of program design, the exact definition of "some-stuff" is unclear. Threading, on the other hand, provides a cleaner solution: create two threads, one thread for text editing, and the other for text printing.

Kernel Support

Systems now provide kernel support for multithreaded programming. For example, a Solaris process has a full-context, and appended to it is a list of the process' LWPs, that is, the CPU contexts of its threads. This provides Solaris with the ability to create multiple threads-of-control within a process' context without the need for a full context switch for each thread. In the diagram below, a simplified process context is shown. In the traditional Unix environment, the "*Thread Contexts 0..n*" do not exist, except for a single context that would have been used in place of "*LWP Link.*"

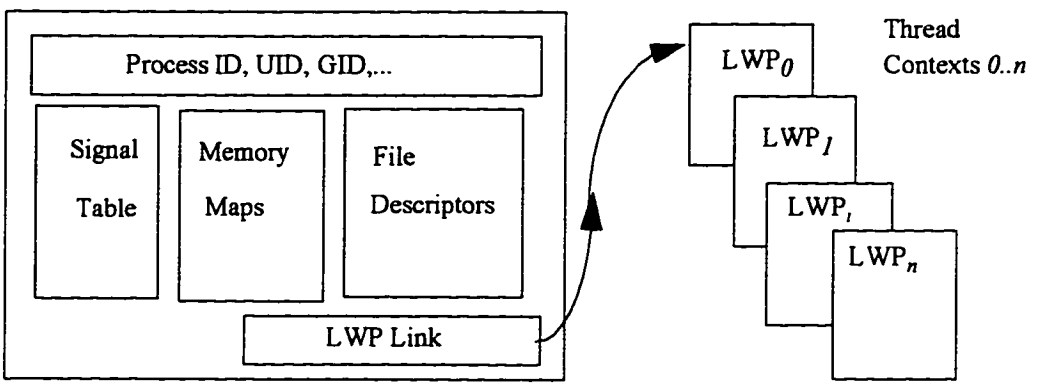


Figure 3.4 Solaris' LWPs

Solaris refers to these kernel supported multiple contexts of a process as light-weight-processes (or LWPs), and the entities that its user level library manages are simply called threads. In fact, there is a set of LWP system calls. However, Sun recommends not using these LWPs directly, but rather accessing them through the threads library. Thus Solaris' threading mechanism is two tiered.

NT also provides kernel support for multithreaded programming, where the pro-

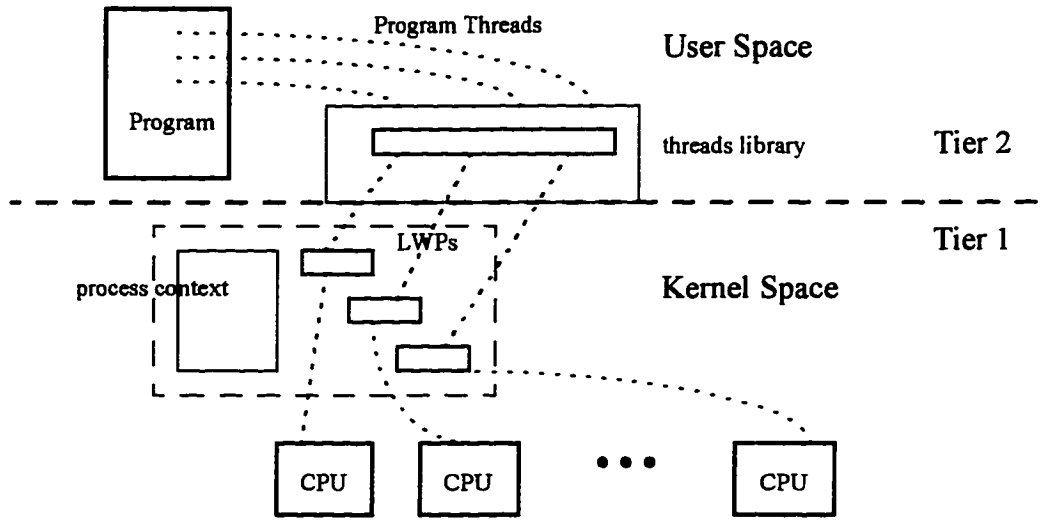


Figure 3.5 NT's Threads

grammer directly calls the kernel which creates and controls threads. NT's threads system is not tiered and each thread that is created becomes a kernel entity, and is treated as a process. A Windows NT program communicates to the system kernel through a protected sub-system known as the *Win32* subsystem [6]. Through a set of Win32 APIs a program can create and manipulate its own threads. It is illustrated in figure 3.6.

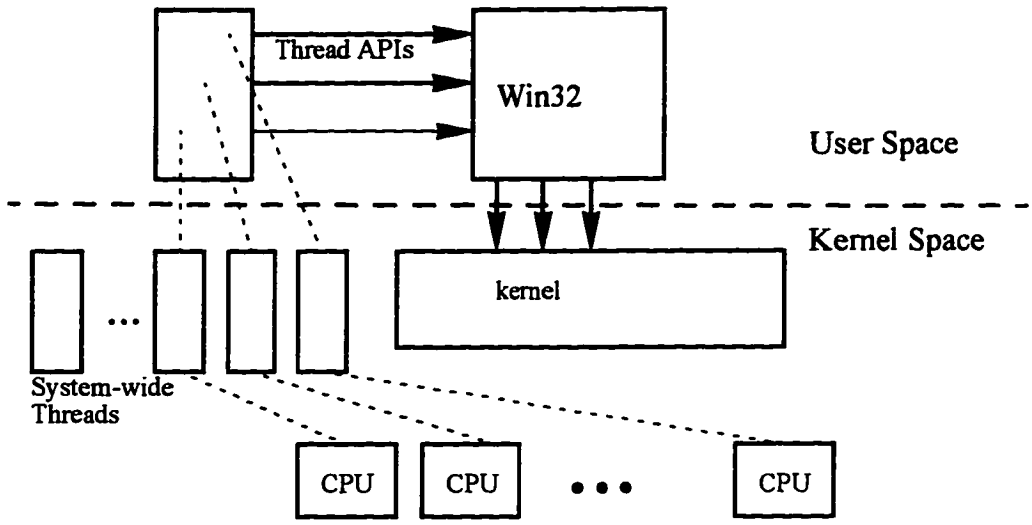


Figure 3.6 NT Threads and CPUs

In the next section we discuss Solaris' and NT's approaches to thread scheduling.

The ability of a system to support multiple threads within the context of a single process' address space provides for scalable low-overhead parallelism; for example, as additional processors become available (e.g., a processor board is added), threads can be created providing the potential of increasing parallelism within the process [25,58]. The use of multiple threads within a process space also provides a natural form of intraprocess communication and can support multitasking services that are transparent to the operating system, thus lending per se to portability [8].

Co-Routines

Some real-time control systems and co-routines, while not "threading" systems, may be considered its predecessors. Within real-time programming a main program that looped through several service routines, that perform a "quick" computation, then return, is a form of co-routining. The co-routine paradigm, however, is closer to the

modern threading view, and operates as follows: a function A calls subfunction B, but B need not "complete" prior to returning control to its caller. That is, subfunction B can perform a certain amount of computation, then RESUME the caller function. Function A, can then allow the subfunction B to resume and so on [9,60] .

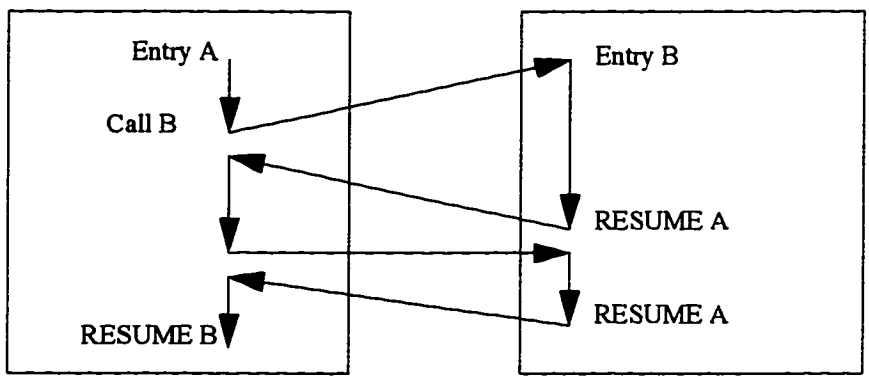


Figure 3.7 Con-Routines

This "jumping" back and forth can be viewed as the multiplexing of two instruction streams, thus simulating a MIMD machine.

Multithreading and Co-routines

In a non-preemptive multithreading environment, multithreading reduces to co-routining. Consider the user level library that is outlined below. It is based on cooperative multithreading. A light-weight-process (LWP) is created by a call to the library function `create_lwp`, that captures the local stack environment (e.g., `setjmp`) of the calling function, and then places that information on a queue of ready threads. The `create_lwp` function now transfers control to the library's scheduler that selects a thread from the ready queue. The scheduler then gives control (via a `longjmp`) to one of the waiting threads. Cooperatively, at some point, this new thread yields control back

to the library's scheduler. In a cooperative environment the thread will need to explicitly yield control. This yielding of control is equivalent to the RESUME of co-routining.

No Operating System Intervention

The above library outline can be developed to run without the operating system becoming involved as demonstrated, for instance, C/C++ threading libraries as in [59] have been developed. These are packages that used non-preemptive threads, and work in systems such as MS-DOS and Windows 3.1. Both threading libraries run atop of MS-DOS: a single user, single tasking system. Unix systems, however, seem to have made the last abstraction that has evolved into the present day threads. The "hello-threads" code fragments below illustrate the programming differences between cooperative multithreading and pre-emptive multithreading systems.

```
foo()
{ int j;
for(i=0; i<10;i++) {
printf("Hello Thread\n");
Yield();
}
}
main()
{
THREAD *hThread;
BOOL original;
Split(&hThread,&original);
if (!original) foo();
while( getchar()!= 'c');
}
```

```
foo()
{ int j;
for(i=0; i<10;i++)
printf("Hello Thread\n");
}

main()
{
beginthread(...,foo,...);
while (getchar() != 'c');
}
```

The previous code fragment on the left uses a cooperative multithreading scheme. Close examination shows that in this implementation, the programmer must explicitly place a call to `Yield()`. Whereas, in the preemptive version on the right,

some combination of the system and library will interrupt the current thread of execution. In the preemptive version it is still possible to explicitly yield control; for example, `thr_yield` in Solaris threads or `Sleep(0)` in NT threads. The Solaris operating system is not aware of the threading that is occurring in the user's space [50,19,28]. This is in stark contrast to NT, whose threads are kernel entities [53,56,57].

Chapter 4: Thread APIs

Rudimentary Threads

As a basic building block to creating a multithreaded program, consider the following two versions of the standard toy program: *Hello World.* Employing "simple" thread creation APIs, 10 independent threads are spun-off from the main program, each of which displays "Hello World" on the current console. Although the semantics of the two systems are the same, in that they produce threads, the underlying software mechanisms differ for each system. The only real syntactic difference between the two code fragments is in the APIs used to create a new thread of control. The two code fragments below illustrate the similarities between NT [54] and Solaris' [19] thread creation APIs.

```
                                /* COMMON CODE */
void hello(void *dummy)
{
printf("Hello World\n");
}
                                /* WINDOWS NT */
#include <process.h>
void main()
{ int i;
  for(i=0; i<10; i++)
    _beginthread(hello, 0, (void *) 0);
  exit_thread(0);
}
                                     /* SOLARIS */
#define _REENTRANT
#include <thread.h>
void main()
{ int i;
 for(i=0; i<10; i++)
  thr_create(NULL, NULL, hello, NULL, NULL,NULL);
}
```

In most cases on a uniprocessor system, both fragments will execute correctly,

displaying 10 distinct "Hello World" messages. However, because of the preemptive nature of the two systems, it is possible that the `printf` call will be interrupted. To ameliorate the situation Solaris and NT made a number of their standard C run-time library functions "thread-safe." This means both threads can concurrently access `printf` without a problem. The details of a thread-safe library will be discussed later in this chapter. In NT, however, the console is not thread-safe. Thus there is the possibility that 10 threads sharing the console will overwrite portions of each other's output.

To protect the console from being overwritten by interlaced `printf` statements, there are a number of schemes available. These schemes will be discussed in greater depth in chapter 6. At the moment, however, we introduce a primitive operation known as a critical-section in NT to protect the console. It acts as a gate-keeper, allowing only one thread to pass. The inclusion of the fragments below in the NT version provides each thread with exclusive access to the console at the `printf` call.

```
CRITICAL_SECTION now_printing;

void hello(void * dummy)
{
 EnterCriticalSection(&now_printing);
  printf("Hello World\n");
 LeaveCriticalSection( &now_printing);
}

main()
{int i;
 InitializeCrtitcalSection(&now_printing);
 ...
}
```

Passing a Structure to a Thread

When creating a thread it is sometimes convenient to pass it data. Threads are passed data through a generic pointer (i.e., `void *`), this is supported by NT, Solaris, MACH [20,23], and POSIX Pthreads [27,29,30]. To illustrate, assume that `N` identical threads are to be created, each with a unique message passed as a parameter. For demonstration purposes a message is simply the current value of an iteration count in a for-loop. Through each repetition of the for-loop, the pointer, `globlal_message`, references a structure allocated via a call to `malloc`. Since malloc's allocations are from the current process' heap, all the process' threads have access right to the address. After the structures allocation it is then initialized, and its address is passed to `_beginthread`. By definition, each thread can accept one parameter of type `(void *)`, thus the `beginthread` call must cast the thread's data. The parameter that is finally passed to the thread is maintained on that thread's stack, and thus is specific to that thread and it exists for the life of the thread.

```
struct  thr_msg { int datum;
          };

struct thr_msg *global;
void worker(void *thr_mm)
{ struct thr_msg *my_msg;
   my_msg = (struct thr_msg *) thr_mm;
   printf("My message is %d\n", my->datum );
   free(my_msg);
}

void main(void)
{ int i;
   for(i=0; i<N; i++) {
      global = (struct thr_msg *) malloc( sizeof(struct thr_msg) );
      global->datum = i;
      _beginthread(thread, 0, (void *) global );
   }
   while ( getchar() != 'c');   /*wait to terminate */
}
```

Thread APIs

Sun's Solaris [19,32] and Microsoft's Win32 [55,56,57]systems each supply a set of APIs that allow a programmer to create, destroy, suspend, resume, and change a thread's scheduling priority. Threads share their process' global and static memory, this includes static local memory. Thus a thread does not inherently have private static memory unless such memory is passed from function to function as an argument, or special features are added to the compiler, language, or library API set to provide private static data for threads. These features will be discussed in the next section.

Solaris' Thread Creation and Destruction

The Solaris thread library interface provides an API to create three types of threads: *bound*, *unbound*, and *daemon*. The distinction among them is that a bound thread is permanently attached to a kernel light-weight process, and scheduled as

though it were another process. The threads library schedules unbound threads onto kernel LWPs. The scheduling mechanism assigns an unbound thread onto an LWP until it completes, blocks, or an unbound thread with a higher priority is ready to run. When one of the aforementioned cases occurs, the threads library will preempt the current thread and schedule another unbound thread on the available LWP. Thus, the threads library does not time-slice threads, but rather *multiplexes* unbound threads onto LWPs in such a way as to keep as many threads active as possible. Only LWPs are timed-sliced by the kernel. In contrast to unbound threads being multiplexed onto LWPs, bound threads own their LWP. This ownership means that if a bound thread blocks, it is not preemptive, and does not allow any other thread use of its LWP. When a thread is created as a daemon thread, it is effectively invisible to the process. It runs in the processes' "background," and is used primarily to provide support services to the other threads, such as garbage collection.

To create a thread there is the library API `thr_create()`, it uses six parameters, and its prototype is:

```
int thr_create( void * stk_base,  size_t skt_size,
          void *(*func)(void *),  void * arg,
          long flags,  thread_t *id);
```

A thread is destroyed either through an explicit call to `thr_exit()`, `thr_destroy()`, or implicitly through its termination. If `main()` calls `thr_exit()`, the process' other threads will continue to run. However, if the main thread should call `exit()` or simply returns, then the process itself terminates, thus destroying all its threads.

Suspension and Resumption of Threads

Threads can be suspended in one of three ways: voluntarily through blocking such as waiting on a mutex, forced into suspension by another thread, or created in a suspended state. When a thread is suspended by another thread or created in a suspended state it remains suspended until it is signalled to resume. Solaris' user threads library and NT's threads both have APIs to force the suspension of a thread, and the ability to awaken it. In Solaris the APIs are `thr_suspend` and `thr_continue`, and in NT the APIs are `SuspendThread` and `ResumeThread`. Along with brief descriptions, prototypes of each call are given.

Solaris' Suspend and Continue

```
int thr_suspend(thread_t target_thread_id);
```

If the call is successful, the target thread is no longer executing. There is no count kept of calls to `thr_suspend`, thus once a thread is suspended, this call has no future effects of the thread.

```
int thr_continue(thread_t target_thread_id);
```

The suspended thread is awaken by the call, thus it is placed on thread libraries list of ready threads. If a signal is directed to a suspended thread, the signal remains pending until the thread is continued.

NT's Suspend and Resume

```
DWORD SuspendThread(HANDLE hThread);
```

If the call is successful, the thread is suspended. Each thread has a *suspension counter* that is incremented on each call to `SuspendThread`, if this count in non-zero the thread

remains suspended. If a thread's suspension counter exceeds a set limit defined by the manifest constant MAXIMUM_SUSPEND_COUNT, the call returns an error and the counter is not incremented.

```
DWORD ResumeThread(HANDLE hThread);
```

If the call is successful, it decrements the suspension counter of the thread specified by hThread, and returns the previous suspension count. If the returned suspension count is zero, it means that the thread was not suspended and its counter remains at zero (that is, the suspend count is never less than zero). If the returned value is 1, this indicated that the thread was suspended, and has now been scheduled to run. Note: in order for a thread to leave a suspension, its suspension counter must be decremented to zero by ResumeThread.

Manipulating A Thread's Scheduling Priority

In Windows NT, altering a thread's scheduling priority affects the system-wide scheduling of other processes, whereas changing a thread's priority within Solaris' thread library has little or no system-wide impact. However, care must be taken in both systems. For example, locking conflict can prevent high-priority threads from executing while allowing lower-priority threads execution status. Consider the following uniprocessor scenario given in [18] for three threads: A, B, and C. Further let the priority of the threads have the ordering: *priority*(A) > *priority*(B) > *priority*(C). Now suppose the following events:

1. B and A are asleep;
2. C is running and locks a mutex M;
3. B awakens and preempts C because priority(B) > priority(C);

4. B starts a very long computation;
5. A awakens and preempts B, because priority(A) > priority(B);
6. A attempts to lock mutex M;
7. B is awakened to continue its very long computation.

The high-priority thread A cannot make any progress because the lower priority thread C is holding a lock that A needs. Thus thread A is essentially blocked until the lower priority threads make sufficient progress. One suggestion is to raise the priority of thread C, prior to locking M. However, "the real solution lies with the implementer of your threads library" [18]. Also, since the threads library does not time-slice threads across LWPs, it is possible that a running thread that never relinquishes its hold on an LWP (viz., never blocks) will prevent another thread, of equal priority, from making progress.

Solaris' Get and Set Thread Priority

The threads library of Solaris provides for getting and setting a thread's scheduling priority through the functions `thr_getprio()` and `thr_setprio()`, respectively. This does not affect the priorities used by the kernel's scheduling of LWPs, the entities on which user-level threads are run. It does, however, affect how the thread's library schedules its threads onto available LWPs. A thread's scheduling priority is an integer value from zero, the lowest priority, to the largest integer number (2^{32}), representing high-priority threads. The scheduler will preempt lower-priority threads to accommodate a ready higher-priority thread, if there are no unused LWPs available for the thread.

```
int thr_getprio(thread_t target_thread_id, int *priority);
```

A call to thr_getprio() returns the priority of the thread pointed to by target_thread_id in the parameter pointed to by priority. It returns a zero if the attempt was successful, and a non-zero value to indicate that an error has occurred; for example, when the target thread cannot be found within the current process, it returns a value that corresponds with the manifest constant ESRCH, located in the header file *errno.h*.

```
int thr_setprio(thread_t target_thread_id, int priority);
```

A call to thr_setprio() sets the current priority of the thread specified by the parameter target_thread_id. If the call succeeds, the target thread's priority is set to the value specified by the parameter priority. Otherwise, it returns an error indicating that either the thread was not found in the current process (cf. above), or that the requested priority level was not accepted because it "makes no sense for the scheduling class " [19]. The constant that represents this error is EINVAL.

Controlling Thread Priorities in NT

The NT system is for the kernel to directly schedule all threads based on priorities levels 0 (lowest, know as THREAD_PRIORITY_IDLE) to 31 (the highest class called PRIORIR_TIME_CRITICAL). The fundamental unit of scheduling in NT is the thread, and all threads are scheduled system-wide. The basic policy is to schedule the highest priority threads first, as in a multi-level queue described in [16], then piecewise to step through each of the other priority levels. This policy, in effect, gives application programmers direct input into system-wide scheduling. WIth this power, of course, is the danger of starving some threads; for example, assume that on a 4 processor system there are 4 real-time threads, each of which is (near) always ready to

run. In such a system it is possible to starve all non-real-time threads. In fact, the MS-WIN32 Programmer's reference warns that "...a base priority level above 11 interferes with the normal operation of the operating system"[54]. Using, for example, a REAL_TIME_PRIORITY_CLASS may prevent disk caches from flushing, make user interfaces non-responsive, and so on. NT has two application program APIs for thread priority, they are SetThreadPriority and GetThreadPriority

```
BOOL SetThreadProirity(HANDEL hThread, int Priority);
```

This function will change a thread's priority level, the first parameter is a HANDLE to the target thread, and the second is the priority level. If the function succeeds, it returns TRUE, otherwise it returns FALSE.

```
int GetThreadPriority(HANDLE hThread);
```

Given the HANDLE to a target thread, this function returns the integer value of the thread's current priority. If the call succeeds, it returns the target thread's priority, otherwise it returns THREAD_PRIORITY_ERROR_RETURN.

Variables and Threads

Stack based variables are private to each thread; however, their static global and static local variables are not private. For example, in ordinary programming it is not uncommon to use a static variable within a function to record previous uses of the function. If a function with static local data were to be called by multiple threads, each

```
int counter;        // global
int foo()
{ static first = 0;   // static local
  ....
}
```

would access the same static variable in memory. Automatic variables, because they

are stack-based are instantiated anew on each function call, and are thus *thread-safe*; that is, multiple threads of control can concurrently exists within the function, each with a private copy of the function's variables.

Thread Specific Data

In traditional programming there is only one thread-of-control, and thus all static variables are referenced with respect to that single thread. However, for a multi-threaded program to maintain the semantics of the static variable, the definition of a static variable needs to be extended to make a class of static unique to a thread. This is new class is called *thread specific data* (TSD). Solaris' and NT both provide functionality to support TSD through set of APIs that can access a data pointer based on the calling thread. Both supply functions to create, destroy, get, and set thread specific data. In the following description the prefix `thr_` is used for Solaris' APIs and `Tls` denotes NT's TSD APIs. Operationally, for both NT and Solaris, the first step is to create a key (Solaris) or allocate a slot (NT), via the functions `thr_createkey` and `TlsAlloc`. When a function needs to access its TSD it calls a function to retrieve it. To initialize the value of TSD storage, the APIs `thr_setspecific` and `TlsSetValue` need to be called. TSD storage need only be initialized once, and this is typically controlled by the use of ordinary static value that acts as a flag. The functions `thr_getsepcific` and `TlsGetValue` are employed to access TSD storage, these APIs need to be called once on entry. Finally, both systems provide APIs to deallocate TSD keys (or slots); they are the `thr_destroy` and `TlsFree` APIs. In the end of chapter 6 an example is given that used Solaris' TSD functions.

Chapter 5: Cooperation among Asynchronous Threads

The purpose of this chapter is to provide an introduction to software methods that are used to control access to data and to synchronize threads. The methods that will be discussed can be found in Solaris' thread library and in the NT system. First, however, an explanation is given as to why access control and synchronization are needed and to suggest how computer hardware might support these activities.

By Default Statements are not Atomic

Before proceeding to show the need for access control of data and the need for thread synchronization, it must be made clear what is meant by an atomic instruction. An atomic instruction is an instruction that completes without interruption. This can be made clearer by examining when an instruction is not atomic. Consider, for example, the programming statement `A=A+1`. This can be unwrapped to show a number of assembler language instructions.

```
MOVE REG1, A
INC REG1
STORE A, REG1
```

This sequence runs the risk of being interrupted, between instructions and thus the statement `a=a+1` is not atomic. As a more insidious example, consider the C langauge's increment operator (i.e., ++) [33,34]. In the statement `A++`, as shown below in

assembler code, it appears atomic.

```
INC A
```

However, there are two "hidden" phases that a CPU must step through. In the first phase, a CPU must fetch the contents of memory location *A*, place it into an internal register, and then increment that register. In its second phase, the new value must be placed back into memory location *A*. The statement loses its atomicity between the first and second phases, where memory location *A* is exposed to possible modification from another CPU or a DMA transfer, for example. Thus, `A++` cannot be assumed atomic. Generally, statements are not atomic, unless explicitly stated to be such.

Support of Atomic Activities

This section examines how a processes' access to the system can be controlled. This section examines four basic methods that can be employed to restrict a processes' access to the computer: hardware, traps, software, and abstract operations.

Hardware support

"All multiprocessors include hardware mechanisms to enforce atomic operations" [68]; for instance, a CPU physically outputs an electrical signal to other hardware components to temporarily disable them. As an specific example of hardware support, Intel Corp's X86 family of microprocessors provides an instruction prefix called `LOCK`. When used as a prefix to Intel's exchange instruction (i.e., `XCHG mem, reg`), it physically causes one of the CPU's pins called to be activated[1], providing a hardware signal that can be used to inhibit other CPUs from gaining access to

1. Specifically, it's pin V3 on the Pentium processor.

memory; thus the instruction (LOCK XCHG mem, reg) becomes atomic. In other words, the CPU executing the instruction will have exclusive access to the memory bus while it completes a LOCKed instruction. Another example is Sun's Sparc system's atomic Load and Store Unsigned Byte (ldstub) instruction. In a multiprocessing environment the aforementioned examples are meant to be exploited to ensure the exclusive access of shared memory.

As a more concrete example, consider extending the non-atomic increment operator of the C programming language as discussed in the previous section. Using Intel technology we can make it an atomic operation. Intel's assembler LOCK prefix can be appended to the INC operation making it an atomic operation. Currently, in NT an integer is 4 bytes (i.e., 32 bits) wide, and it is represented in Intel assembler language as a double-word (dword). Thus, the increment operation can be rewritten as an atomic statement. Using Microsoft's Visual C++ IDE, the code fragment below provides an example.

```
#define ATOMIC_INC(z)     _asm { lock inc dword ptr [(z)] }

int A;              /* Note this works only for a 'simple' -- or ++
*/

ATOMIC_INC(A);      /* This A++ should now be atomic */
```

Software Support

The software perspective of "atomic activities" is viewed from that of cooperative processes. That is, if two or more autonomous processes are working "together," there needs to be some form of information passing. It is here that a problem arises because the information (i.e., shared storage) can be changed at unpredictable moments. A number of algorithms have been developed in high-level languages that

provide for portions of a processes' code to behave atomically among cooperating programs. This was first demonstrated by Dijkstra[2] [42]. These algorithms support atomic access to data across processes. When a block of statements is required to execute atomically, those statements are often referred to collectively as a *critical section*. Thus a critical section can only be executed by one process at any given time. This does not mean that instructions such as `c=c+1` are now atomic, but rather that, if the statement is shared among processes, each process has an atomic view of that statement (i.e., exclusive access). In otherwords, the access to data that is shared among "cooperating" processes can be made to behave atomically. Algorithms that are based on the assumptions of strong ordering of memory or the atomicity of read/write operations are not portable. That is, assuming that memory is always strongly ordered or that read and write operations are atomic is not fail-safe. It is possible for a programmer to take advantage of a particular platform's features, such programs are not considered portable. In fact, Intel's Pentium Users Manual states "Strong ordering ... may not be implemented in future processors. ..." further the manual recommends the use of explicit synchronization over memory access. Also, not all read/writes are atomic; for example, on a certain system, integer numbers may never need more than one memory access, regardless of their alignment (or due to its size filling the data bus in one fetch), but another system may be have narrower bus or be sensitive to data alignment.

---

2. In his 1965 paper lecture notes, Dijkstra credits the Dutch Mathematician T. J. Dekker with "the first correct solution..." to the problem of coordinating sequential processes.

Memory models

In a strongly ordered memory system, memory accesses will be serialized, thus reflecting program execution order across processes. For optimization purposes some systems may reschedule memory access: these are weakly ordered memory systems. To see where a problem can occur let's consider Peterson's mutual exclusion algorithm. A form of the algorithm follows, and it makes use of three variables to provide alternating mutual exclusive access to memory shared by two concurrent processes.

```
/* Thread 1 */

fool()
{ while (1) {
p1 = TRUE;
f = 2;
while (p1 && f==2)
      ;
/* exclusive area */
share_me++;
p1 = FALSE;
```

```
/* Initialization */
int share_me;

p1 = FALSE;
p2 = FALSE;
f = 1;

beginthread(foo...);
beginthread(boo...);
```

```
/* Thread 2 */

foo2()
{ while (1) {
p2 = TRUE;
f = 1;
while (p2 && f==1)
      ;
/* exclusive area */
share_me++;
p2 = FALSE;
```

In the program fragments above, the functions `fool()` and `foo2()` are concurrent threads that employ Peterson's algorithm to effect an atomic increment of the variable `share_me`. To begin, the shared variables `p1` and `p2` are initialized to `FALSE`. This means that neither thread requires access to its respective critical section. Also the shared variable `f` is set to `1` (i.e., `f=1`), it is used to indicate a "favored" thread, in this case, the favored thread is `fool()`. Prior to a thread entering its critical section it first indicates that it wants access by setting its corresponding `p` variable to `TRUE`; for example, if `fool()` is about to attempt access, it sets `p1=TRUE`. The next step is to give the other thread preference by setting the flag `f` to indicate that the other

thread should have access first. This favored treatment will force a thread to loop until the other thread releases its hold on its exclusive section. The exclusive section is released when the thread holding it resets its corresponding p flag (e.g., p1=FALSE), thus releasing the other thread from its busy-wait while-loop.

The algorithm fails if both processes enter their critical sections at the same time. The following condition will cause such a failure: p1=p2=FALSE. Let's begin our analysis assuming p1 and p2 are both TRUE, and say, f=2 (wlg). This means thread two is now the favored process, and should drop through its while-loop. But let's freeze both threads just before they each access a value of p1 and p2, and take a look at memory. Suppose that the initial values of p1 and p2 (i.e., both are TRUE) and are still pending in memory, when thread 1 running on CPU 1 requests p2, and thread 2 running on CPU 2 requests p1. While each thread will see its reads and writes in program order, across CPUs this may not be true, that is, unless the system is specifically designed to adhere to the sequential memory model. Thus it is possible that thread 1 will read the old value of p2, and thread 2 will read the old value of p1.

Abstract Operations for Atomicity

A number of high-level approaches have been developed for insuring atomic activities [11,14,17]. The next section in this chapter will exclusively deal with some of these primitive building blocks.

Mutual Exclusion.

In general, when two or more threads share an object or variable each thread should have exclusive access, this is known as *mutual exclusion*.

Counting Semaphore

A semaphore is a synchronization primitive[3]: it provides a building block to control access to a shared object. The need for synchronization cannot be understated, in fact, a shared object is often referred to as "protected", when some method guarantees that one and only one thread can access it at any given instant in time. In general, the object can be any non-sharable or limited share resource; for example, a record, an integer, or communications line. A semaphore is, itself, a protected counting variable, that is defined over non-negative integers, and can be operated on by two atomic functions[4]: `signal()` and `wait()`.[5] Given the semaphore, S, wait and signal are defined as follows:

```
signal(S)
{
if (S>0)
then S=S-1;
else (thread must block on S);
}
```

```
wait(S)
{
if (threads are waiting on S)
then (unblock a waiting thread);
else S=S+1;
}
```

This protected counting variable can be exploited in a number of ways to provide access control or synchronization among threads. As an example of access control, a semaphore, S1, can be used to control utilization of, say, resource, R1. S1 is then initialized to the number of threads that can simultaneously access R1, and prior to each thread's attempt to acquire the resource, it must first perform a signal operation[6].

---

3. Originally, it as define over the cooperation among two or more sequential processes, however, a thread can be considered a special case of a process.
4. There needs to be a means to initialize a semaphore; for example, sema_initail(&S, N), and this function need not be atomic, but must be called only once.
5. Originally known as P and V operations, respectively. P was short for proberen te verlagen (try and decrease), and V meant verhogen (to increase).

There are two possible outcomes of a call to the signal function: non-blocking or blocking. If the count of threads is less than the allowed maximum, S1, then `signal` returns to its caller and is said to be non-blocking: otherwise, `signal` will block itself (i.e., the current thread) waiting on semaphore S1. When a thread is ready to release resource R1, it calls `wait`. The call to `wait` will either release a blocked thread that is waiting on S1, or it will note, via `S1=S1+1`, that R1 is available for access by one more thread. In the code fragments below, the resource R1 allows 4 threads to safely compete. A functional extension is the ability to test if a resource is available, to allow for an alternative action.

```
SEMAPHORE S1;
...
sema_init(&S1, 4);
...
R1(...)
{
signal(&S1);
/* RESOURCE IS HERE*/
wait(&S1);
}
```

```
worker(...)
{
...
R1(...);
...
}
/* worker 1 */
```

```
worker(...)
{
...
R1(...);
...
}
/* worker N */
```

Binary Semaphore

There is a special case of the general semaphore, called the *binary semaphore*, *mutex*, or *critical section*. It is initialized to either a one or a zero. Typically, a one (1) means that the resource is initially available, and a zero (0) means that it is not initially available. It is commonly used to provide synchronization or mutually exclusive access to a shared object, for instance, a global counter. In addition there is the exten-

6. It is very important to note that if more than one programmer is involved, they all must cooperate on this point.

sion `try_lock`, this test to determine if a given protected variable is currently locked. It allows the option of not having to block if the object is currently unavailable; for example, if the resource is currently locked, then go on to do something else.

Recursive Mutex

Another type of mutex is recursive. In a recursive mutex, a thread can lock the same mutex repeatedly, and it must be unlocked as many times: this is a semantic extension to the mutex primitive discussed earlier. In Solaris and Windows NT, a thread is allowed to repeatedly lock the same mutex variable, and for each lock, the system will automatically increment a counter. For the thread to release the mutex variable, it must release the lock as many times as it acquired the lock, and this can only be done by the owner. That is, no other active thread can successfully release another thread's mutex lock.

Design Factors

Designing a parallel system requires that careful consideration be given to selecting the "right" protection mechanism, since each vendor's system provides varying degrees of additional built-in functionality to the basic semaphore. The NT system, for instance, has two implementations of the binary semaphore: mutex and critical section. The critical section is a plain-vanilla protection scheme that is not managed by the system's kernel and operates very quickly, whereas its mutex counterpart is a kernel object that provides additional features and is slower. The mutex of Solaris and NT each have the functionality of being able to be used for inter-processes synchroniza-

tion or access control. However, when a thread in NT acquires a mutex lock, it is also given "ownership" of the mutex by the kernel. This "feature" can be exploited when a thread terminates and does not release its mutex(s). Microsoft refers to such mutex an *abandoned* mutex. When the kernel finds that a mutex's owner no longer exists, it will automatically release the mutex and return a condition code of WAIT_ABANDONED. This means that the next thread that attempts to lock this abandoned mutex will be able to detect that there was a problem.

The textual location of a semaphore, mutex, or a critical-section within a multi-threaded program will generally be as a global[7], thus providing each of the process' threads with access rights. In considering the granularity of protection of shared files, lists, or arrays it may be inefficient to place the entire structure under the auspices of a single mutex. It is not, in fact, uncommon to provide finer grained control; for example, given a list of items that are more often read, than written to, each item can be protected by a unique mutex that only locks when a writer attempts access: this creates a list of many readers and one writes[8]. Although it is possible to construct the functionality, systems such as Solaris' have built-in library support (e.g., `rw_lock`).

As another example of granularity, consider the generate-and-test paradigm, where a generator creates an array of jobs postings for workers, and each worker will need to check through the array to find an assignment. If workers block (or spin-wait) while accessing assignments this can cause unnecessarily delays; i.e., wait while either the generator or another worker is busy accessing the posting. Whichever the case, it is

7. Multithreading represents a class of programs that clearly benefits from the judicial application of global variables.
8. Also referred to as the readers and writers problem.

possible that other workers will race to this posting and block: causing a pile-up of blocking workers. This potential pile-up, however, is avoidable by stepping over the "busy" posting. To implement this stepping-over the programmer needs to take advantage of the test_mutex primitive. Operationally, it will first test to determine if a lock exist, if it does, it will return a message to that affect; however, if there is no lock, it will then lock the semaphore and report as such. Solaris provides an explicit library function called `mutex_trylock()`. While NT does not provide a specific function call to `try_lock`, it can be built using its slower binary semaphore (ie., `mutex`).

```
Result = WaitForSingleObject(hMutex, (DWORD) 0);
```

Conditional Variables

There are times when threads need to synchronize on events. One method to accomplish this is to use a busy-wait loop, e.g., `while(!my_event)`. It provides a cheap alternative to calling a more expensive synchronization function. Given the generic code fragment below with the global variables `my_event` and `mu_event`, a thread *j* can release any number of threads that are spin-waiting on the boolean `my_event`: once released, the threads can continue to execute. Not withstanding that the processor time spend spinning can far out weight the cost of putting the thread to sleep.

A shortcoming in the code below is that thread *j* must know, apriori, what event(s) the other threads are waiting on. A mutex alone is not sufficient, recall when a thread is blocked there is no means for it to "watch" for a flag to change and then awaken---this is where the conditional variable applies.

The solution is to have a thread examine its condition of interest (e.g.,

```
 /* wheel spinner */          /* Thread j */
foo()                         boo()
{                             {
...                           ...
while ( !my_event);           my_event = TRUE;
...                           ...
}                             }
```

count==6), and if it is not true then go to sleep, else it should proceed. Once asleep it should be awaken only when an event occurs that signals a change in the condition of interest, at which point the thread will again test the condition and decide if it should continue, or sleep. Originally developed within the context of solving the problem of mutual exclusion in monitor entry and exit [15], to that end, a structure called a conditional variable was defined with three operations on it. In the context of multithreaded programming, the structure is exposed, that is, the implied lock during monitor entry is now made explicit to the programmer[9], thus when strict coding rules are enforced the semantics are unchanged. Assuming C is a conditional variable, the two atomic functions below express its semantics.

- `Cond_wait(C, mutex)` : the thread calling this function is suspended and is place on a FIFO queue, and its mutual exclusion variable is unlocked, and it is done atomically.

- `Cond_signal(C)` : If the conditional variable's queue is not empty, then awaken the thread at the head of the FIFO queue.

The conditional variable is a higher level abstraction than the mutex, and there are of course other associated functions that create and destroy conditional variables, as well as other extensions; for example, `broadcast`. In systems such as Solaris and the POSIX definition of Pthreads the conditional variable is implemented through a

9. Of course, the lock need not be used, but the resulting effect would be a weaker form of the conditional variable.

sequence of instructions. What follows is an illustration of the application of Solaris' conditional variable.

Solaris' Conditional Variable

A conditional variable permits a thread to block until signalled. Conditional variables can be used among the threads of a single process, or among processes if it is declared in a mutually shared memory segment. Solaris supplies six function calls to support conditional variables that perform creation, indefinite wait, finite wait, signal a condition, broadcast a condition, and destroy a conditional variable. In a multithreaded program a conditional variable is declared with:

```
#include <thread.h>
cond_t tea_time;
```

Before it can be used it must be initialized once, but can be re-initialized safely if the programmer can guarantee that no thread, or process, is waiting on it. Initialization of a conditional variable is accomplished via a call to `cond_init`.

```
int cond_init( cond_t *cvp, int type, int arg );
```

The first parameter is a pointer to the conditional variable. The second parameter (`type`) indicates whether the conditional variable is to used locally or among processes. If it is to be used within the current process space only, then `type` becomes the manifest constant `USYNC_THREAD`. Otherwise it is `USYNC_PROCESS` which indicates that it is to be used among different processes. The third parameter, `arg`, is, as of this writing, not used and may set to any integer value

```
err = cond_init( &tea_time, USYNC_THREAD, 0 );
if (err == EFAULT ) {
      printf("illegal address for conditional variable \n");
      exit(0);
}
```

Once initialized, a thread can use a conditional variable to block itself and wait indefinitely to be awakened or block for a fixed amount of time. An indefinite wait is through a call to `cond_wait` and a fixed wait is through a call to cond_timedwait.

```
int cond_wait( cond_t *cvp, mutex_t *mp );
int cond_timedwait( cond_t *cvp, mutex_t *mp, timestruc_t *abstime );
```

The first parameter is a pointer to the conditional variable, the second parameter is a pointer to a mutex that protects the testing and changing of the condition(s) that work in cooperation with the conditional variable. The third parameter, in `cond_timedwait`, is the absolute time-of-day that the function will not block past.

To illustrate how these two conditional waits can be used, suppose that a certain thread pours milk into tea, but the tea is currently too weak and needs to steep. In the code below, the while-loop checks the variable `strength`, under the protection of the mutex mp_strength. If it finds that the tea is not right, it then blocks with a call to cond_wait, otherwise, it passes through the while-loop to pour milk.

```
mutex_t mp_strength;

mutex_lock( &mp_strength );
while ( strength != strong )
 cond_wait( &tea_time, &mp_strength );
pour_milk();
mutex_unlock( &mp_strength );
```

There are a two important notes to make at this time: 1) that there can be any number of tea pouring threads, of which, only one is allow to pour milk at a time; 2) if

any of these threads is blocked, then some other thread must provide a signal to release it. In this case the signal would come from a thread whose job is to actually measure the current strength of the tea: this will be explored later in this section on conditional signals.

Suppose the tea is not ready for milk; the thread will then block, and be added to the conditional variable `tea_time`'s waiting queue. At this point the function `cond_wait` will need to also release the mutex's lock on the variable `strength`. This allows other threads access to the variable. When a thread signals the conditional variable `tea_time`, the associated waiting queue wakes up one of its blocked threads. This newly awakened thread will atomically return to the point at which it went to sleep (i.e., `cond_wait`); this includes its competing to regain the mutex lock it held (i.e., `mp_strength`). Once the thread has acquired its lock, it re-tests the tea's strength. The need to re-test the condition `strength!=strong` is necessary because 1) the condition may have changed between being awakened and acquiring the lock; 2) a signal or `fork()` may have caused a pre-mature awaking. In this case `cond_wait` will then returned the error message `EINTR`.

To examine how a timed conditional wait operates, suppose that we want to guarantee that milk will be added to the tea at some fixed point in time. That is, if the tea is not strong enough by some `TIME_OUT`, then some other thread may not be doing its job, so pour milk anyway. As with the indefinite conditional wait, the timed wait's condition(s) needs to be re-evaluated. Also, an additional test may include `cond_timedwait()`'s return value to check whether it returned because of a time-out (i.e., `ETIME`).

```
timestruc_t Time_To_Wait;
...
mutex_lock( &mp_strength );
Time_To_Wait.tv_sec = time(NULL) + TIME_OUT;
while ( strength != strong ) {
   err = cond_wait( &tea_time, &mp_strength, &Time_To_Wait);
   if (err == ETIME ) {
      printf("Just cannot wait any more to pour the milk\n");
      break;
    }
}
pour_milk();
mutex_unlock( &mp_strength );
```

Thus far we have covered initialization and waiting for conditions. Now we discuss two different methods to signal a conditional variable. These are the cond_signal() and cond_broadcast() function calls. They are used to signal a conditional variable's waiting queue to release one or all blocked threads, respectively. If a conditional signal is sent and there are no waiting threads, then the signal has no effect. The signatures of these functions are:

```
int cond_signal( cond_t *cvp );
int cond_broadcast( cond_t *cvp );
```

The cond_signal(&cvp) call unblocks a single thread that is waiting for the conditional variable cvp. This function should be used under protection of the same mutex as its "corresponding" cond_wait. If this is not done then it is possible that the signal will be missed, causing an infinite wait. For example, a tea pouring thread (cf. above) can be between finishing its test condition (i.e., strength!= strong) and about to block on its conditional variable, when the conditional variable tea_time is signalled. Thus the signal that should have released the tea thread is lost. The following code illustrates correct usage.

```
mutex_lock( &mp_strength );
  strength = strength + drawing_factor;
  err = cond_signal( &tea_time );
  if (err == EFAULT ) {
     printf("Invalid conditional reference \n");
     exit(0);
  }
mutex_unlock( &mp_strength );
```

There may be a time when it is desirable to awaken all the current threads that are waiting on a conditional variable. This can be accomplished through the function call cond_broadcast.

```
int cond_broadcast( cond_t *cvp );
```

When called, it will release all the current threads that are waiting on a given conditional variable. As with the case of the cond_wait, these released threads will attempt to re-acquire their mutex locks. It is important to note that the use of the mutex will enforce serial access to each contending thread's conditional test; and, the order with which completing mutex locks are acquire is not defined. In fact, any completing thread that has memory access to the mutex, can successfully attempt to lock the mutex. Therefore, there should be no assumption about the ordering of threads released via a condition broadcast.

Again, consider the tea paradigm. Assume that there is a tea party that needs many cups (threads) of tea filled with milk (or, lemon and honey). Then, if there is no milk available, the pour_milk() function should block until milk is made available; for example,

```
mutex_lock( &mp_milk );
while ( milk_demand <= milk_pitcher )       /* check milk supply */
      cond_wait( &got_milk, &mp_milk );
milk_pitcher = milk_pitcher - milk_demand; /* add milk to cup */
mutex_unlock( &mp_milk );
```

Here, any pour_milk() thread will block if there is not enough milk to satisfy its demand, milk_demand. Once blocked, it can be awoken by a signal from the conditional variable got_milk, and test to determine if there is enough milk. If the milk supply is sufficient, it will drop through the while-loop to pour milk into a tea cup, and then release its hold on the local milk supply. Behinds the scenes, there is a pitcher of milk that services each cup of tea. Milk is added to the pitcher by

```
mutex_lock( &mp_milk );
   milk_pitcher = milk_pitcher + added_amount; /* add milk to pitcher */
   cond_broadcast( &got_milk );                /* release all tea threads */
mutex_unlock( &mp_milk);
```

When milk is added to the pitcher, it broadcast the information. Thus allowing all of the waiting threads to scramble for milk. Lastly, a condition variable can be "destroyed" by removing its state information:

```
int cond_destroy(cond_t *cvp);
```

Therefore we conclude the tea party with a final example of removing, from the system's memory, the conditional variables got_milk and tea_time.

```
cond_destroy( &got_milk );
cond_destroy( &tea_time );
```

Joining Threads

On occasion it is necessary to wait for one or more threads, that is, some thread puts itself to sleep until awakened by the rendezvous of a specific thread(s). This is akin to barrier synchronization[4,5], and similar to UNIX's `waitpid` [31] system call.

Solaris, POSIX's Pthreads, and NT each provide the functionality to allow a thread of execution to block, until a specified thread (or, optionally, any thread) terminates. In order to wait for multiple threads[10], multiple joins need to be written. In NT, there is no specific function called `join`, or `wait` for threads. In the NT approach, each thread is a system object, and thus the APIs `WaitForSingleObject` or `WaitForMultipleObjects` can be used to wait for a single thread, or a set of threads, respectively.

10. In Solaris if multiple threads are waiting on a common thread, only one can succeed, and the other will produce and error (i.e., ESRCH).

Chapter 6: Shared Memory and Forks

Shared memory

The Unix operating system provides mechanisms for interprocess communication (IPC), and one such method is shared memory. Unlike other IPC mechanisms, such as the pipe, the use of shared memory does not require kernel intervention [31]. Unix provides a number of APIs that can be used to establish the sharing of memory among processes.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>   /* shared process memory
functions */
#include <errno.h>

#define PERMS     (0666) /* Permission */
int     shmid;       /* id for shared data segment */
int * shmptr;        /* pointer to actual shared datum
*/
```

Using the following code, we will look at three *shared-memory* APIs shared-memory-get (shmget), shared-memory-attach (shmat), shared-memory detach (shdt). It is important to detach shared-memory to avoid "zombie" like segments that require that the system be re-booted to remove them.

```
main(void)
{
     shmid = shmget(IPC_PRIVATE, sizeof(int), PERMS |
IPC_CREAT );
     if( shmid == (void *) -1) {
         perror("Cannot find or open segment");
         exit(0);
     }
     shmptr = (char *) shmat(shmid, (char *)0, 0 );
     if (shmptr== (void *) -1) {
         perror("Shared memory binding error");
         exit(0);
     }
     /* the pointer shmptr is now available */
     shmdt(shmptr); /* finished with the pointer */
}
```

The function shmget attempts to create (or open) a shared memory segment, returning either a valid identifier, or (-1) if it fails. In order for a process to reference an open segment it must be bound to that segment. A segment is bound to a process via a call to shmat, which will return an actual pointer to the shared segment. Source code accesses the segment's data through standard pointer deferencing techniques. The fragment above was tested on the Solaris 2.5.1 OS. Not illustrated in the above code are the two additional shared-memory APIs that detach a segment (shmdt) and remove a segment formthe system (shmctl). When a process no longer needs a shared memory segment, it can detach itself via a call to shmdt. This call is similar to free() and would resemble:

```
if ( shmdt( shmptr)<0 ) perror("Cannot detach segment");
```

To remove the segment from memory the call would be:

```
if( shmctl( shmid, IPC_RMID, (struct shmid_ds *) 0)<0)
      perror("Cannot remove shared memory");
```

Forks

By employing the system call `fork`, a (parent) process can create a new process in the Unix environment. It effects the establishment of an independent process referred to as a child process, and this manifest process is identical to its parent. The child is given a duplicate copy of the parent's address space, this includes file descriptors and the contents of each variable. In fact, the child's execution commences at the very point at which the parent resumes, that is, following the return from the call to `fork`. To differentiate between a parent and its child process, the call to fork will return a valid process ID to the parent and a zero to the child process. As an illustration, consider the code fragment below.

```
pid_t    child_id;
int      status;
...
hello(char *c)
{
   printf(" hello %s\n", c);
}
main()
{
child_id = fork();                    /* parent creates a child process */
if (child_id == -1) {
     perror("Cannot create child process");
     exit(0);
}
if (child_id == 0) {                  /* both parent and child start here */
     hello("Cheryl");                 /* must be the child process */
     exit(0);                         /* exit child process */
}
waitpid(child_id, status, 0);/* parent will wait here for child process */
}
```

It is important to note that the parent and all its children processes execute asynchronously, thus , if there is any shared data segment that contains a non-constant data, that data must be protected from any potential race condition via a mutex,

semaphore, or other protective scheme. In Solaris' multithreaded environment, `fork()` also includes the duplication of all of the parent process' threads of control, including its light-weight-processes. This is in contrast to the `forkl()` system call that duplicates the parent's address space in the child, but then only duplicates the thread of control that is called `forkl()`. Lastly, `vfork()` gives the parent's address space to the child, and creates a single thread of control, as with `forkl()`. The parent of a `vfork()` process get its address space returned only when the child exits or calls `exec()`.

Forks and Threads

Updating libraries and programs so that they are thread-safe or multithreaded, can be very difficult and costly. This section examines the cost of employing processes (heavy-weight task) in place of threads, as an alternative to making libraries and programs thread-safe. Our rational is that multiprocessing systems are becoming less expensive and more powerful, and while threads provide an excellent means to achieve low-cost, low-overhead parallelism, we considered it worth examining the use processes to create, not "thread-safe" but parallel safe libraries and programs.

Creating a process through the Unix `fork` system call and the creation of a thread using Solaris' `thr_create`, are similar in that both create a new thread of control. There are, however, two major differences.

1. when a thread is created with thr_create it remains within the current process' context, thus all global and static variables are accessible to it. This is in stark

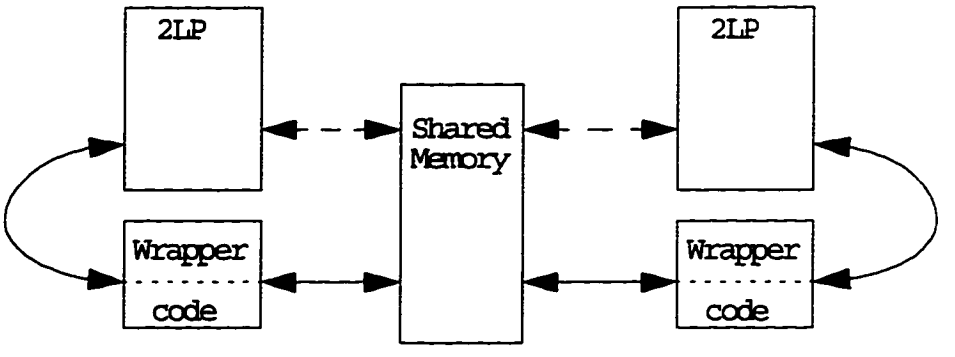
contrast to a forked process that exists within a separate context and special features are required to provide communication;

2. the fork's creation of a new process adversely impacts the system's global scheduling of resources, whereas, by default, when `thr_create` is called, the threading library schedules its the threads on LWPs, given it by the system. Thus, it is possible that any number of threads can be created and not adversely impact system-wide scheduling.

Clearly, the above mentioned differences make threads attractive. However, this attractiveness can be lost when attempting to rewrite a program or library to make it thread -safe. For example a program may make extensive use of global data, thus requiring that instances be placed in giant mutexed structures, or altering function calls to fit pointers to these structures. In "organically" developed code, the relationship among variables may not be clear. While these programs can be "systematically" made thread-safe, it may be easier, and cost effective, to create a new process thread through system APIs.

Forking 2LP

The goal is to have 2LP fork such that there is one version for each processor in the system, with a common memory segment. Thus each version of 2LP will be able to communicate via the common memory that is shared. The shared memory can provide for the distribution of work among the various 2LP processes. The diagram below illustrates the communication paths.



In the diagram above each solid line represents communication through a function call, and dashed line represents (direct) communication through deferenced memory. Previous work on 2LP utilized a LAN [2,3] for parallelization. In comparing LAN parallelization to "`fork()`" parallalization, the application of wrapper code (i.e., external C functions for communications and control) is present in PIG and `fork`. The cost of using the external function calls from 2LP is identical. In contrast, the cost of using communications over an LAN is clearly is greater, thus forking has a reduced cost. In favor of PIG is its robustness in terms of being able to select from a pool of available LAN processors.

Shared memory, mutexs, and forking

The code fragment below provides an outline on how we setup shared memory in 2LP. A structure is defined for the shared space, it can contain integers, doubles, mutes, semaphores, conditional variable, and arrays for example. In this example, we use a shared mutex, `mp_best`, to protect the variable `best` from being corrupted by multiple processes.

```
typedef struct _sharemem {
              int best;
              mutex_t mp_best;
              /* other shared variables */
} SHAREMEM;

SHAREMEM *memp;          /* pointer to shared memory */
int         shmid;       /* shared memory id */

/* create a shared segment */
shmid = shget(IPC_PRIVATE, sizeof(SHAREMEM), PERMS | IPC_CREAT);

/* get access to the shared segment */
memp = (SHAREMEM *) shmat(shmid, (char *) 0, 0 );

/* setup a mutex to protect data */
mutex_init( &memp->mp_best, SYNC_PROCESS, 0);
```

With the above code in tact, a process, is ready to fork children that can communicate via shared memory.

Benchmark

To benchmark the speed of process creation and destruction on a "live" system, a program that forked N child processes was run, where each child simply terminated. The idea is to have some measures of the cost of using a fork, without regard to other implementation details that may influence the outcome; for example, avoiding the intergration of shared memory. As a contrast, the `fork()` call was converted into `thr_create`, where each thread terminated.

In table 1, it can be seen that under a heavy system load threads had a consistently lower creation and deletion time then forks. Also, in terms of reliability, it appears that threading can support a much larger set of workers without the system returning a core dump or a "`resource temporarily unavailable`" message because of of work load. An explanation for this is that Solaris's thread library is not part of the

system's kernel, thus there is no direct system-wide impact in running threads because the library schedules threads onto LWPs given to it by the system. This is in stark contrast to NT, where each thread is a system entity and its existence has an impact on overall system performance; in fact, under NT 3.51 running a similar thread creation benchmark, the system appeared limited at about 213 threads per process, at which point, if the threads are computationally intensive, the system becomes sluggish.

Table 1: Threads versus Processes

Thread/ Process Count	Time (seconds)				
	fork()		thr_create(...)		
	Heavy Load	Light Load	Heavy Load	Light Load	Bounded (light-load)
2	0.05	0.03	0.06	0.03	0.03
5	0.06	0.03	0.06	0.03	0.03
10	0.10	0.04	0.06	0.05	0.04
20	0.14	0.05	0.07	0.06	0.05
50	0.80	0.27	0.08	0.08	0.08
100	1.00	0.13	0.11	0.13	0.13
200	2.98	0.23	0.15	0.14	0.25

* denotes that the system could not always produce the desired number of requested processes.
** denotes that the system could not produce the full number of processes once.
heavy load --- other user processes were on the system (e.g., running a 2LP model)
light load --- no other users were on the system.

Chapter 7: Thread-Safe Constraint Programming

Our goal was to build the first thread-safe constraint programming system. To that end, we redesigned the 2LP constraint programming system and extended it to support multiple threads. Thus, we produced a new constraint based system that can exploit parallel programming paradigms based on threading APIs.

This new multithreaded constrainted based programming system has shown itself portable onto both an Intel Pentium based platform and a Sun Sparc based platform with little more than a re-mapping of threading APIs. Further, the research shows that with the advancement of shared-memory multiprocessing technology that the application of the traditional Unix `fork()` can be a viable alternative to reworking a system so that it is thread-safe.

Logic Programming + Linear Programming = 2LP

2LP is a programming language developed at the Logic-Based System Laboratory of Brooklyn College, CUNY[1]. It is a hybrid language that is able to express algorithms using a declarative or procedural approach. It contains much of the syntax of the C programming language [36]. It incorporates a specialized variable type called *continuous*. This variable type models the continuous variable that is found in linear algebra and linear programming.

As an example of the `continuous` variable type consider the problem of maximizing the function $4x + 5y$ over the region defined by $x + 3y <= 15$ and $2x + y <= 10$ in

the following graph. In 2LP, the problem can be encoded as:

```
2lp_main()
{
     continuous x, y;
     x + 3*y <= 15;  // constraint 1
     2*x + y <= 12 ; // constraint 2
     max:
       4*x + 5*y;      // optimize
     printf("x=%3.1f and y=%3.1f\n", x, y);
}
```

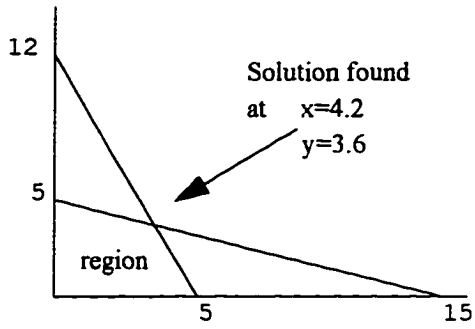


Figure 7.1: constraint graph

Within the 2LP code example, x and y are declared to be of type continuous and are used to enforce two constraints. Operationally, as constraints appear within the source code, the 2LP solver attempts to assimilate them; thus forming an evolving region known as the *feasible region* in linear programming [80]. If a new constraint is added and it is not consistent within the current feasible region, the solver will reject it, and the constraint is said to have *failed* [1]. In the case of failure 2LP will attempt to *backtrack* to a previous step in the code where an alternative action was possible. Such a step is known as a *choice-point*. These are the same notions of *choice-point*, *backtracking*, and *failure* that are used in the logic programming of Prolog. An overview of Prolog and constraint logic programming is given in appendix 1.

The keyword max is a reserved word operator that forces the 2LP solver to find a vertex in the region defined by the current constraints such that the expression following max is maximized. The output of the previous program shows a solution as x=4.2 and y=3.6. However, had the list of constraints been inconsistent, that is,

one of them fail and no alternative be available, then the system would have output: "No feasible solution exists."

2LP has four types of looping constructs: and, sigma, or, and c_or. As an example of its looping and backtracking ability consider the basic *knap-sack* problem. In this problem the objective is to find a combination of the weighted binary variables (x_i) such that the combination's sum is equal to N. The following equation

$$\left(\sum_{i=1}^{N} ix_i = N \right), x_i in \{0, 1\} \forall i$$

embodies the mathematics of the problem, and the coding example that follows illustrates various features of the 2LP language as used in solving this simple problem.

```
#define N   3

2lp_main()
{
 continuous x[N];
   sigma( int i=0; i<N; i++)        // shorthand notation
      (i+1)*x[i] == N;              // sum must equal N
   and( int i=0; i<N; i++)          // logical and-loop
      either x[i]==0; or x[i]==1; // logical persistent disjunction
   display(x);
}

display(continuous x[])             // an output procedure
{
   and( int i=0; i<N; i++)          // logical and used as a for-loop
      printf(" %3.1f ", x[i]);
}
```

Here an array of N *continuous* variables are declared. The sigma loop construct provides a notation to write constraint expression; for example, it can substitute for x[0] + 2*x[1] + 3*x[2] +...+ N*x[N-1] ==N. This is followed by a logical andloop, which attempts execute the persistent disjunctive statement: either-or. Effectively, the and-

loop attempts to assign either a 0 or a 1 to the continuous variable `x[i]`. Specifically, it attempts to bind `x[i]==0`, and if that binding fails, it will then attempt to bind `x[i]==1`. Next, the function `display` is called, and its job is to print the bindings that were successfully given to `x[0]`, `x[1]`, ..., `x[N-1]`. Here, the and-loop is used as an ordinary `for`-loop of C. Graphically the program searches a binary tree, and it does so using the depth-first search method that is inherent to 2LP.

Below is the partially annotated search tree of the binary knap-sack prob-

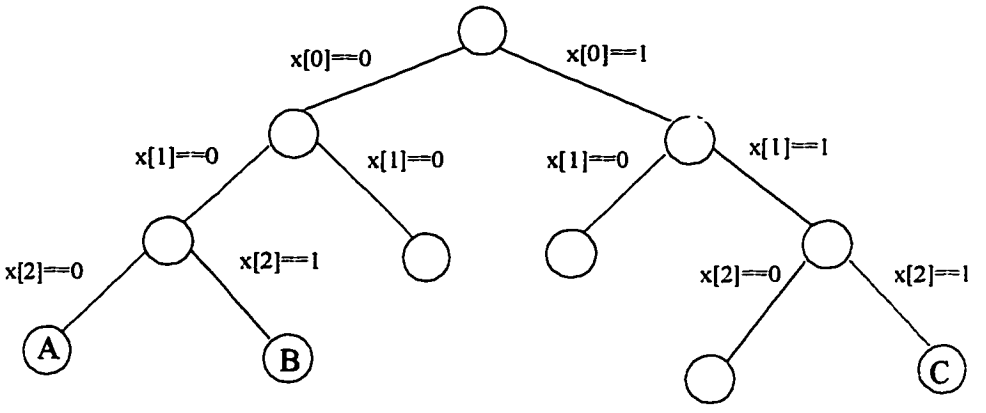


Figure 7.2: Knapsack search tree

lem. Note that some search paths lead to *failure*, such as nodes *A* and *C* in the tree above, while other nodes (e.g., B) end in *success*. Further examination will show that its branches can be explored in parallel.

Itinerary-Based Parallel 2LP

Atay [2] exploited 2LP's declarative nature to develop an OR-parallel implementation as in Prolog. The method was called the *itinerary* and it allowed 2LP's internal mechanism to replicate a 2LP process, assign work among the repli-

cated 2LP processes, and maintain a "picture" of the past and current search space. The parallelism that was provided was transparent to the programmer; the underlying mechanics of 2LP controlled all parallel activities. This transparent mechanism removed the burden of parallelism from the shoulders of the programmer, and produced impressive speedups over sequential versions. On the downside, this approach restricted the class of programs that could be made parallel (*amenable* programs [2]) and "locked" in the parallelism of the search strategy to the list of variants made available by the 2LP system itself. The itinerary mechanism was successfully ported to a number of systems which included a network of Sun workstations, Intel's iPSC Hypercube, and the BBN Butterfly architectures. These systems supported various memory models: the tuple space of C-Linda, PVM's message passing, and the shared memory model.

An *itinerary* is an array of integers representing the various paths leading from the start of the search tree to a current node in a search tree. That is, an itinerary "leads a process to a point in the search tree" [2]. Operationally, an initial worker would give directions to other workers through an itinerary integer array. Each of the workers would then proceed independently. Workers were able to communicate with each other via the itinerary. Associated with this communication was the need for a *cut-off level*, it represented a level in the current search tree where a worker could not venture past when backtracking. It was used because the work back-up the tree was either already accomplished, assigned to another worker, or has been pruned. The cutoff level also served as a means to vary work distributions. To understand the

itinerary method consider the following search tree in figure 7.3.

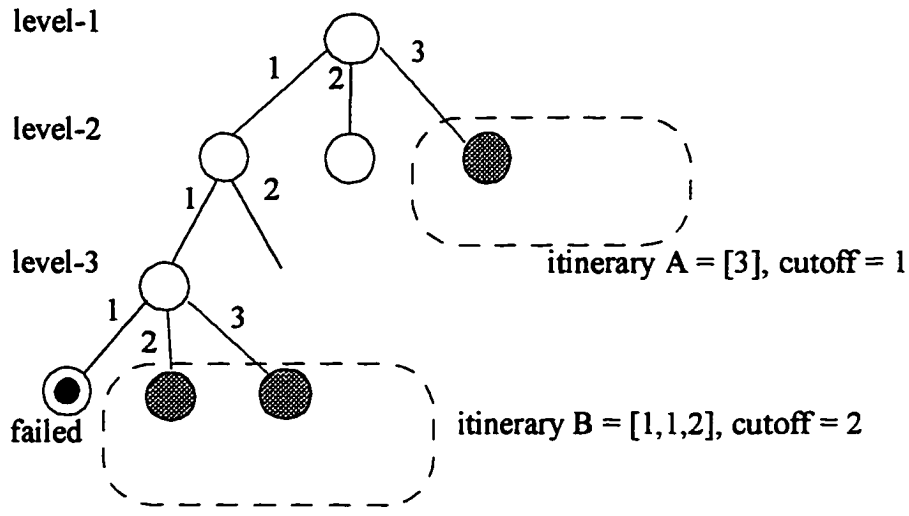


Figure 7.3: Itinerary search tree

In the search tree, itinerary *A* will cause a worker to search the subtree beginning at level-1 choice 3. Also, because its cutoff is 1, its search is restricted to the subtree starting at node 3. Itinerary *B*'s path through the search tree starts at level-3 choice 2, with its cutoff level given as 2; thus it is not permitted to backtrack up the tree to level-2. As a result, it confines its search to the subtrees that can be chronologically generated at level-3 choice-2, then level-3 choice-3.

Parallel Integer Goal (PIG) Programming

The next effort in 2LP's development, in terms of parallelization, was accomplished by Arnow, McAloon, and Tretkoff [3], where 2LP was specifically integrated atop the distributed processing library (DP) to specifically address integer goal programming problems. The end result was referred to as parallel integer goal program-

ming (PIG) paradigm [3]. This method of parallelization utilized the DP software library to farm-out work over a LAN. To accomplish this task, five functions were created , in what is referred as *glue* code. These functions included:

- `pig_start()` is called once by each remote 2LP process.
- `pig_stashset()` is used to initialize a global set of "work orders."
- `pig_getwork()` is called by a 2LP process to get a work order.
- `pig_split()` is optionally called by a 2LP process when the amount of global work orders is too low.
- `pig_brag()` allows a processes to broadcast its best solution.

These functions provided a means for the 2LP process (model) to communicate with a PIG support library, and for the support library to communicate with the 2LP model. Below is a diagram of the PIG system.

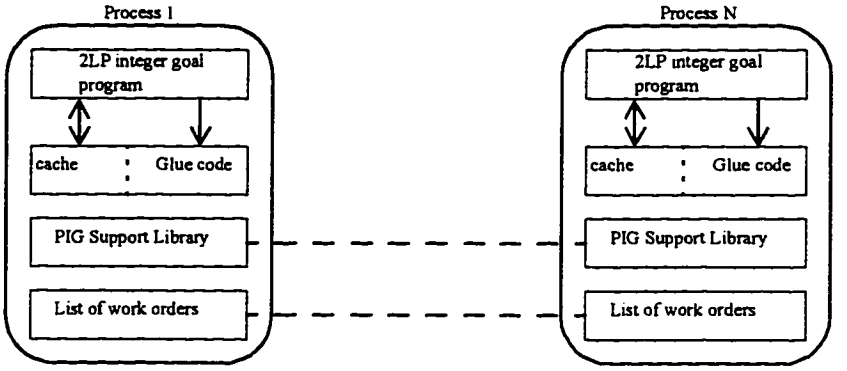


Figure 7.4: PIG diagram

The PIG programming paradigm provides the following template to generate work and to acquire work in a 2LP model.

```
2lp_main()
{
        start_up();                    // call an external startup routine
        and(;;) {
          if  get_work(goal, work);
                then preform(goal, work);
                else return;           // exit
        }
}
```

Thread-Safe Constraint Logic Based Programming

This thesis research has produced the first "thread-safe" constraint programming system; e.g., Ilog is not thread-safe, and CPLEX is---but it is a run-time library. To accomplish the conversion of the standard single-threaded version of the 2LP system a number of choices were made has how to create a new system that maintained all the semantic power of the 2LP modeling system. To build the thread-safe system we: 1) closely examined 2LP's internal structure to develop a "consistent" design method to make any and all internal changes; 2) created new variable classes to support multiple threads; and, 3) developed software support for threading 2LP models that would supportability between Microsoft's Windows NT and Sun's Solaris operating systems.

2LP's Internal Structure

The first task was to divide the 2LP system into functional components with as little overlap (in variables) as possible. In some respects this was a simple task; because of 2LP's modular design it naturally divided into a user-interface, parser, error-handler, memory management, run-time interpreter, and a linear constraint solver. There were, however, a number of cases where certain variables acted as an

epoxy, gluing modules together making their classification difficult. This required major reworking of 2LP's internal structure to get to the point where static storage could be collected into a single structure. This new structure captures the global state of the run-time environment. It is used to pass all pertinent information along to other modules via function calls.

The diagram below illustrates the usage of data and structures among 2LP's various components. The bytecode is the compiled 2LP model, it consist of code as a

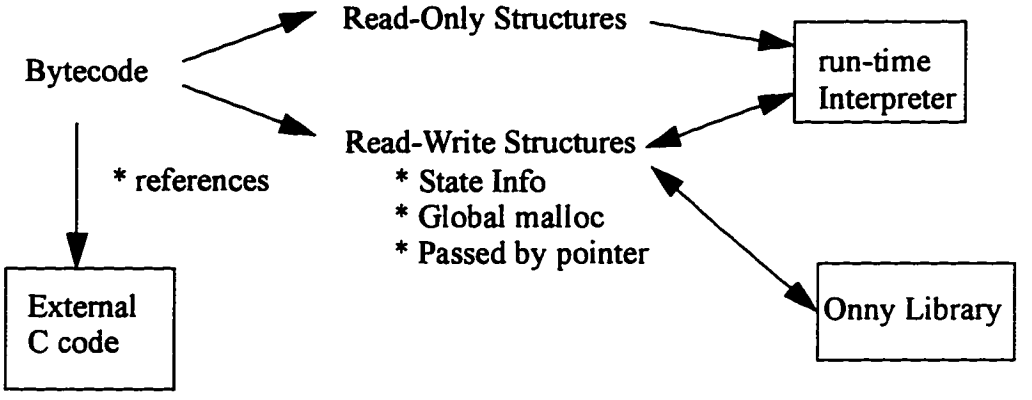


Figure 7.5: 2LP's data structures

read-only data structures that are read by the 2LP run-time interpreter. A part of the bytecode contains references to data structures that are read and written to directly by a model, and indirectly by the attached constraint solver, in this case, 2LP's own *onny* library. Also, shown is a wrapper module that was developed to permit the bytecode to access external C functions that coordinate activities among a model's running threads. In summary, the many substantial changes dealt with dividing 2LP's variables such that multiple threads were able to execute and the design goals became:

- minimize the need of locks in order to maximize concurrence;
- pass all necessary state information in a single minimal structure to minimize stack operations during function calls;
- keep the 2LP bytecode as read-only data to be shared among all the running threads, and `malloc` any necessary thread-specific data;
- argument the system with wrapper code to support coordination among a model's threads.

Figure 7.6 diagrams the thread-safe 2LP constraint-based system that was developed, referred to as TSAFE in benchmarks. This new system has been adapted to port to current multithreading thread APIs, these include Solaris [19] and NT [54]. Threading is accomplished by calling a thread API to start-off the run-time interpreter. Currently, 2LP can create multiple versions of the interpreter by simply creating a copy of its initial start-up state. Each new version (or thread) can then run independently, or work cooperatively with additional support functions.

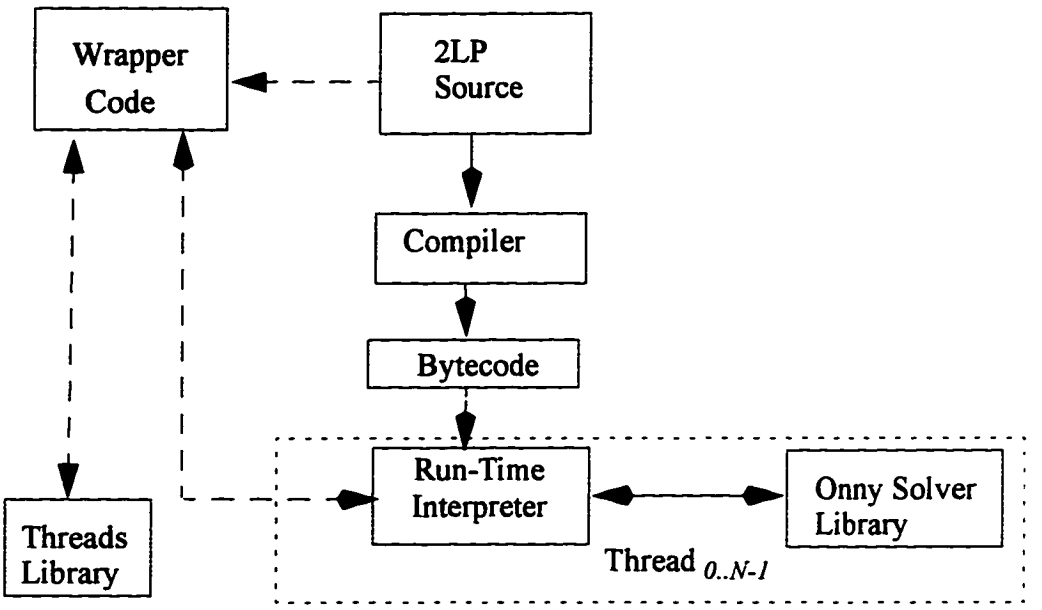


Figure 7.6: 2LP diagram

The Cost of Threading

To measure the cost of threading 2LP, we ran two benchmarks on the TSAFE version, a 0-1-2 knapsack problem and a set-covering problem from netlib at the Operation Research Center at the Imperial College, London, UK. These benchmark problems are compared against the single-thread version referred to as the *standard version*, and are shown in tables 1 and 2 ,below

Table 1: Cost of Threading on a Crossbar system

Version	Solaris on Sparc crossbar		
	knapsack	set-covering	
	Time	Time	Nodes
STANDARD	3:23	1:18	154

Table 1: Cost of Threading on a Crossbar system

Version	Solaris on Sparc crossbar		
	knapsack	set-covering	
	Time	Time	Nodes
TSAFE	3:40	1:00	145

Table 2: Cost Threading on a Bus system

Version	Solaris on Intel bus		
	knapsack	set-covering	
	Time	Time	Nodes
STANDARD	2:50	0:59	478
TSAFE	2:58	1:05	619

Global Thread Specific Data

To support multiple versions on a running model that cooperatively executed in a threaded environment, we created two thread specific variable classes. Descriptive names of these new 2LP class are the *thread-specific global* (TSG) and the *shared static global* (SSG). The TSG variable is global to a model, and its scope is restricted to a single thread. Thus, within a 2LP program (i.e., Model), all procedures have direct access to it, and each thread has a unique instance. The SSG variable can be accessed by any executing thread.

External Support

An aspect of multiple threads of a single model is that the threads are able to cooperatively work towards finding a solution. Thus each instance must be able to

communicate. To accomplish this we chose a method similar to that of the PIG paradigm's control, that is, we use 2LP's ability to call external functions. These functions were then customized to manage the communications aspects not easily integrated into 2LP's internal mechanisms. To support the creating and receiving of work among 2LP threads these include:

1. `initial()`: a set of calls to add_work to start a 2LP program to give-away work.
2. `add_work()`: allows a thread to give away work.
3. `get_work()`: allows a thread to get work, and is also used to signal that all searching has come to an end.
4. `broadcast()`: when a thread has found a plausible solution, it alerts other threads
5. `lock()` and `unlock()`: primarily used to lock the console for displaying results.
6. `whoami()`: returns a thread's unique identification, and is used to farm out work.
7. `thread()`: it creates additional threads through the 2LP run-time interpreter. The newly created threads each have a unique thread id (from 1 to N), with 0 being the initial thread's id. Currently, the only thread permitted to created additional threads is thread 0, and new threads begin at the top a a 2LP program.
8. `inc()`: it atomically increments a given SSG variable

These functions are designed similar to monitors, in that, only one thread is allowed in at a time. For example, any number of 2LP threads can attempt to get a new work assignment, via a call to get_work(), but only one thread is permitted in the function at any time. To provide a portable system, our implementation was

developed to circumvent NT's lack of a conditional variable, thus we elected to use a spin-wait loop, that would allow threads in and out of the external wrapper code. An outline of the get_work under Solaris is:

```
void get_work( int *length, ...)
{    while( mutex_trylock( &lockwork)==EBUSY);
     if (not_spinning) spinner++;
     if (spinner == EVERYONE)  {   // No more work, but maybe later
                        *length = NO_MORE_WORK;
                         mutex_unlock( &lockwork);
                         return;
    }
     if (NO_WORK )  {          // No more work, at the moment
                        *length = SPIN_WAIT;
                         mutex_unlock( &lockwork);
                         return;
    }
   spinner--;       // I'm no longer a spinner
   // get work assignment
  *length = value>0
  mutex_unlock( &lockwork);
}
```

A 2LP program's interface to get_work would appear as:

```
work=0;
      and(;;) {
          get_work( work, ... );
          if work=SPIN_WAIT; then continue;
          if work=NO_MORE_WORK; then break;
          do_work(...);
     }
mop_up();
}
```

Control over Parallelism

The parallelism is hidden in [2], and was specific to integer goal programming in [3]. In this new version, 2LP has been made thread-safe, and its parallelism is becoming native to the system. The coding sample below re-visits the earlier 0-1 knap-sack problem, and convert it to a 0-1-2 choice problem to illustrate how to statically partition a search among threads.

```
#define   N    4
int id;                          // Thread Specific Global
continuous  x[N];                // Thread Specific Global
extern void whoami(int &);       // wrapper code prototypes
extern void lock();
extern void unlock();
int cnt;

2lp_main()
{     cnt=2;
      thread(cnt);        // create additional threads
      whoami(id);
      sigma( int i=0; i<N; i++)  (i+1)*x[i] == N;
            if id==0; then {
                  x[0]==0;
                  and( int i=1; i<N; i++)
                        either x[i]==0; or x[i]==1; or x[i]==2;
            }
            if id==1; then {
                  x[0]==1;
                  and( int i=1; i<N; i++)
                        either x[i]==0; or x[i]==1; or x[i]==2;
            }
            if id==2; then {
                  x[0]==2;
                  and( int i=1; i<N; i++)
                        either x[i]==0; or x[i]==1; or x[i]==2;
            }
      lock();             // lock the console
       display();         // display will work for all threads
      unlock();           // unlock the console
      }
}
```

Using a unique id code, the work is divided among three threads, where each

thread independently searches for a solution. A 2LP program receives its id by a call to the external function whoami(). Specifically, the code above divides the work of finding solution equally among the threads. The thread with id=0 starts its search by binding x[0]==0, with the second thread, id=1, beginning its search with x[0]==1, and lastly, the third thread, id=2, commences its search with x[0]==2.

Employing more structured programming techniques, it possible to make the above code "clearer" by hiding the details of dividing the work. This is illustrated in the 2LP program below, and will define our knap-sack benchmark.

```
#define    N      45
#define    C      N
continuous x[N];      // TSG

2lp_main()
{
 int id;              // Local (inherently private)
      cnt = 2;        // Global thread specific
      thread(cnt);
      whoami(id);
      sigma( int i=0; i<N; i++)  (i+1)*x[i] == C;
      worker(id);
}
void worker(int path)
{
      x[0]==path;
      and( int i=1; i<N; i++)
            either x[i]==0; or x[i]==1; or x[i]==2;
}
```

When executed, the above code will access its id, then establishes a constraint. In this case the constraint is bound regardless of the id. When the procedure worker is called it is given an id that corresponds to a branch in the problem's search space. This allows three threads to execute through the problem's search space in parallel. Also, the same benchmark was run using the version of the 2LP that creates

new processes to achieve parallelism. In table 3, see below, two different hardware architectures (a crossbar versus a bus) were used to benchmark the ability of our new thread-safe constraint system. Both of the systems provided 4 SMP processors, and at least 512M bytes of DRAM.

Table 3: Solaris benchmarks with multiple threads

Solaris 2.5.1		Sparc Crossbar		Intel Pentium Bus	
benchmark	threads/ processes	Threads	Forks	Threads	Forks
Knapsack	1	3:30	3:40	3:30	3:20
	2	2:10	2:00	1:51	1:34
	3	1:15	1:24	1:08	0:58
	4	1:01	1:09	0:56	0:51
	5	1:02	0:59	0:58	0:55
	9	0:59	0:58	0:59	0:57

In our knap-sack benchmark, the work load was statically assigned to each thread in the program via a unique id. A problem with this method is that some threads may work harder than others, and when these lightly loaded threads complete their search, they terminate, leaving the others complete they heaver loads. Thus, initial, the amout of parallelism can equal the number of available processors, but as threads complete, the amount of parallelism decreases. Figure 7.7 illustrates this situation.

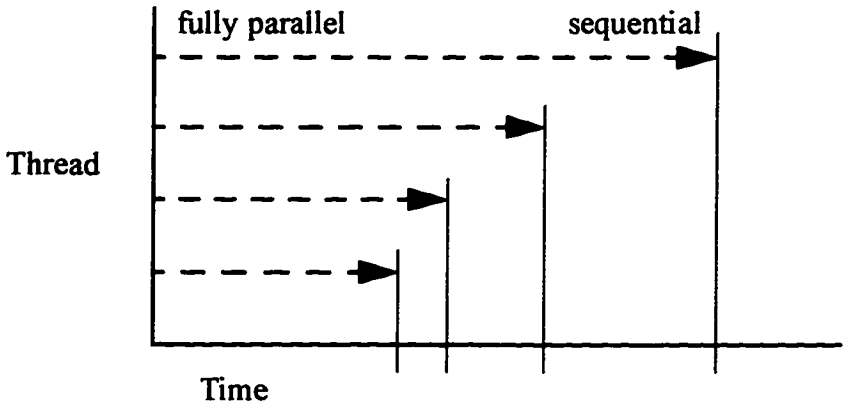


Figure 7.7 Decreasing Parallelism

Aside from threads (or forks) executing independently, there are functions that provide for worker threads to place work aside for other threads. This allows for work to dynamically shared among threads and creates a non-deterministic search atmosphere.

Non-deterministic Search in 2LP

With its foundations in logic programming, 2LP employs a depth-first search method. When, however, multiple threads (or forks) exists, the search method becomes *non-deterministic*. To exemplify this point consider the following search tree. Within this tree the node labeled as the root demarcates the beginning of a search, shaded circles represent explored nodes, and blank circles are nodes that have not been searched but are known choices in the tree's search space. To search the tree, N (concurrent) threads are expanding nodes in depth-first order. As each thread expands a node, it generates an alternative node that is not immediately expanded,

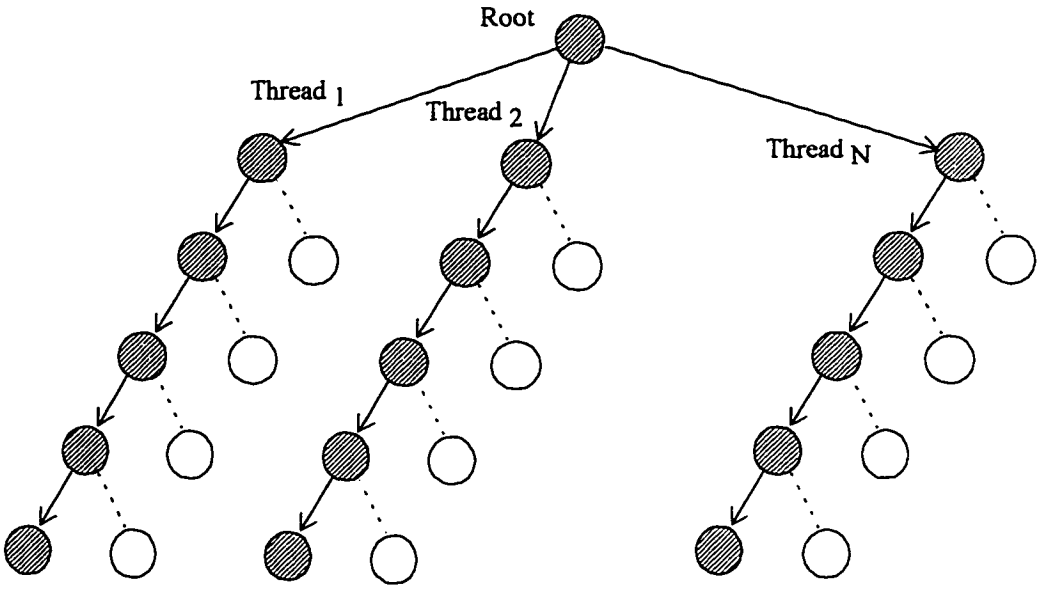


Figure 7.8: non-deterministic search

but rather, placed on a list of unexplored nodes (similar to the itinerary in [2], and the workorders in [3].) Essentially, nodes are added to a list of unexplored nodes in a random order. This is because threads are assumed to execute asynchronously with respect to each other. Thus, guided by such a list, the search is non-deterministic [92]. Threads begin a new search after they complete their current one. A search is completed if either its current path has been exhausted or there is no means for it to improve on some globally available answer.

To benchmark the threaded version's ability to distribute work dynamically we used the same set-covering problem applied earlier, only this time, it is modified to `get_work()` and `add_work()`, and threads do not use their `id` to acquire work

assignments, but rather, work is taken by threads on a first-come-first-serve basis.

Table 4: Dynamic load balancing benchmarks

		Solaris				NT	
		Sparc Crossbar		Intel Pentium Bus		Intel Pentium	
Benchmark	threads	time	nodes	time	nodes	time	nodes
set-covering	1	1:04	145	1:04	619	0:48	214
	2	0:56	221	32	128	0:46	307
	3	0:27	118	0:27	140	0:28	208
	4	0:30	168	0:26	148	0:28	208
	5	0:34	124	0:35	151	0:24	170
	6	0:39	151	0:36	161	0:55	561
	7	0:54	303	0:33	146	0:40	229
	8	0:57	131	0:54	320	0:30	152
	9	0:58	127	0:49	193	0:41	176
	10	1:08	242	0:45	261	1:04	209
	20	1:05	282	1:46	243	1:19	210
	100	9:30	475	6:13	417	3:19	384

Summary

With the addition of the threading variable classes and the support functions previously cited, we have demonstrated the first thread-safe constraint based programming environment. The data in tables 3 and 4 show that there is an improvement in 2LP's performance. We expect that system will take advantage of processors as they are made available. Also, we took the opportunity to "thread" 2LP by using Unix's `fork()` and shared-memory system calls, and found:

1. It took far less work to develop a parallel version of 2LP, requiring only standard

Unix system calls;

2. The start-up cost of generating one process-per CPU was small compared to the length of time in computation;
3. It was easier to parallelize 2LP through Unix's traditional forking and shared-memory calls then to efficiently rewrite 2LP so that it is multithreaded; and,
4. We have been able to preserved the new features of the threaded version in the forked version.

Current hardware trends are toward declining memory cost and tightly-coupled multiprocessor systems, this seems to favor the traditional Unix `fork()` call for a non *real-time* class of problems; for example, (albeit an observation is subjective) the start-up time we experienced as users was insignificant.

Appendix 1: The Prolog Programming Language

Prolog stands for *Programming in Logic*, and its roots are in mathematical logic. It is a declarative programming language that emerged in the early 1970s. Its principle devlopers were Robert Kowalski, Alain Colmerauer, and David Warren; i.e., the developer of an efficient abstract Prolog machine: the Warren Abstract Machine (WAM) [78].

A prolog program is a set of facts and clauses (i.e., if-then rules) that are interpreted through a mechanism that includes pattern matching, recursion, and the notation of built-in backtracking [75,76,77]. The Prolog interpreter attempts to combine facts and clauses together through a method called *unification*. Informally, unification is simply a syntactic pattern matching scheme, where upper-case symbols represent variables (that match anything), and lower-case symbols that represent constant *terms* (that is, entities that are atomic). For example, the variable X can unify with *9*, or *apple*, but the constant term *x* will unify with neither *9* nor *apple*. When a syntactic match cannot be made it is called a *failure*, thus *x* failed to unify with *9* and *apple*.

Generally, a fact in prolog is written in a prefix notation; for example, Paul is male can be written as `male(paul)`. The interpretation of the *relation* male on paul is through the eyes of the programmer. Clauses (or, facts) in prolog are called *horn clauses*, which are clauses that contains, at most, one conclusion [70]. For instance, the rule: X is Y's son if Y is a parent of X and X is male can be expressed as:

son(X, Y) if parent(Y, X) and male(X).

When the prolog interpreter is "primed" with facts and rules, it can be *queried* to start its pattern matching and backtracking mechanism; for example, son(jonathan, larry). To illustrate how Prolog operates using unification and backtracking, we will encode the following simple family relations: X is a son of Y if X is male and Y is a parent of X. ; kyle, jonathan, and michael are male; larry is a parent of jonathan and michael.

```
son(X,Y) if male(X) and parent(Y, X).
male(kyle).
male(jonathan).
male(michael).
parent(jonathan, larry).
parent(michael, larry).
```

With the above facts and rule priming the prolog interpreter, we will now query it to find a son of larry, using the query: son(X, larry). Starting from the textual top of a prolog program, the interpreter begins all matching attempts.

1. Unification matches son(X, larry) with the first rule and thus *binds* larryto Y
2. With the success of the first unification under its belt, the interpreter now attempts to find a match that satisfies both male(X) and parent(X, larry). These two relations are called *subgoals*[1]

3. The interpreter now looks for a match for male(X), and *succeeds* on its first attempt to produce the new binding of jay to X. Also, to facilitate *backtracking*, it marks that it has visited male(jay), this is called a *check-point*.

---

1. note: The textual ordering of subgoals is significant: it determines the order in which facts and rules are found; and can effect whether a recursive rule terminates.

4. With its first subgoal solved, the interpreter advances to its next subgoal, namely, parent(jay, larry).

5. Once more the interpreter looks down its list of facts and rules looking for a match (i.e., parent(jay,larry); however, it *fails* to find any rule or fact to match. Thus this attempt *fails*, and the interpreter *backtracks* to its last success.

6. With its tail tucked, it unbinds X, and returns to its last *check-point*, i.e., step 3. Again it attempts to match male(X), this time succeeding with the binding of jonathan to X

7. (as in step 4) With its first subgoal solved, the interpreter advances to its next *subgoal*, parent(jonathan, larry).

8. Again the interpreter looks down its list of facts and rules. This time it finds an exact match, thus it returns *success*.

9. With all its subgoals solved, the interpreter, returns as an answer to the query(X, larry) the response: X = jonathan

Prolog and Constraint Satisfaction

Prolog is based on unification, and thus its semantics are defined within the context of the universe of all possible terms (syntactic binding) that are derivable from within the program's text. As a result the output from a Prolog program is explicit; for example, simple structures such as male(kyle), or more complex structures X= f(g(apple),orange) can be derived. This is in contrast to, say, the points implicitly defined by x>1 (where x is a *Real* number) cannot be explicitly defined via unification. Thus for many domains, Pro-

log's finiteness of its output is a severe limitation.

Constraint Logic Programming (CLP), on the other hand, adds a constraint satisfaction engine to Prolog's unification mechanism. Thus, a CLP language has all the semantics of Prolog, plus the ability to represent and interpret information implicitly. Consider the (recursive) Prolog program below that computes the factorial of an integer.

```
fact(0, 1).
fact(N, F) if N>0 and fact(N-1, M) and F=N*M.
```

If the program were queried with fact(3,X), through unification, the interpreter will return the correct answer, X=6. But, if the query were fact(X,6), Prolog will fail: this is because the N in the expression N-1, must first be bound to a value; and that is not possible with the given query. In CLP there is no need to syntactically bind the numeric expression. Thus CLP has "greater" expressive power.

Bibliography

[1] K.McAloon, C. Tretkoff " Optimization and Computational Logic," Wiley-Interscience, New York, 1996. ISBN: 0-471-11533-9.

[2] C. Atay "A Parallelization of the Constraint Logic Programming Language 2LP," Ph.D. Thesis, CUNY, 1992.

[3] D. Arnow, K. McAloon, C. Tretkoff "Parallel Integer Goal Programming," Brooklyn College, CUNY, Tech. Report 1995.

[4] M. J. Quinn "Parallel Computing: Theory and Practice," McGraw-Hill, New York 1994. ISBN: 0-07-051294-9.

[5] B. Lester "The Art of Parallel Programming," Prentice-hall, New York, 1993. ISBN: 0-13-045923-2.

[6] S. Kleiman, J. Voll, et al, "Symmetric Multiprocessing in Solaris 2.0," Ca., SunSoft, Inc., 1994.

[7] P.McJones, G. Swart "Evolving the Unix System Interface to Support Multithreaded Programs," Digital SRC Research Report 21, Pa, Sept. 1987.

[8] P. Andleigh, "Unix System Architecture," Prentice-Hall, 1990. ISBN: 0-13-949843-5.J.R. Eykholt, S.R. Kleimen, et al, "Beyond Multiprocessing...Multithreading the SunOS Kernel." Proc. of the Summer 92 USENIX Confer., San Antonio, Tx.

[9] T. Pratt "Programming Languages: Design and Implementation," Prentice-Hall, New York, 1984. ISBN: 0-13-730580-X.

[10] D. Grunwald, R. Neves "Whole-Program Optimization for Time and Space Efficient Threads," ACM Sigplan, Vol. 31, No. 9, Sept 1996. pp50-59.

[11] M. BenAri. "Principles of Concurrent and Distributed Programming," Prentice-Hall, New York, 1990. ISBN: 0-13-711821-X.

[12] A.M. Lister "Fundamentals of Operating Systems" Spring-Verlag, New York, 1984. ISBN: 0-387-91251-7.

[13] H.M. Deitel "Operating Systems," Addison Wesley, New York, 1990. ISBN: 0-201-18038-3.

[14] J.L.W. Kessels, "An Alternetive to Event Queues for Synchronization in Monitors," Comm of the ACM, Vol20, No.7, July 1977, pp 500-503.

[15] C.A.R. Hoare "Monitors: An Operating System Structuring Concept," Comm. of the ACM, Vol. 17, No. 10, Oct 1974. pp 549-557.

[16] E.W. Dijkstra "Solution of a Problem in Concurrent Programming Control," Comm. of the ACM, Vol 26, No 1, pp 21-23.

[17] L. Lamport "A fast Mutual Exclusion Algorithm," Digital Equipment Corp., SRC Research Report 7, Pa., Oct, 1986.

[18] A. Birrell "An Introduction to Programming with Threads," Digital Equipment Corp., SRC Research Report 35, Pa., Jan, 1989.

[19] SunSoft "Solaris: Multithreaded Programming Guide", Prentice-Hall, New York, 1995. ISBN: 0-13-160896-7.

[20] L. Walmer, M. Thompson, "A Programmer's Guide to the MACH system Calls," Tech Report, Dept of Comp Sci., Carnegie-Mellon Univ. Pa., 16 Nov, 1989.

[21] E. Cooper, R Draves, "C Threads," Draft report, Carnegie-Mellon Univ., Pa., 11 Sept, 1990.

[22] R. Baron, D. Black, W. Bolosky et al "MACH Kernel Interface Manual," Dept. of Comp Sci, Carnegie-Mellon Univ., Pa., 23 August 1990.

[23] J. Boykin, D. Kirschen et al "Programming Under Mach," Addison-Wesley, New York, 1993. ISBN: 0-201-52739-1.

[24] T. W. Doeppner Jr. "A Threads Tutorial," Brown Univ. Tech. Report, CS-87-06, 30 March 1978.

[25] T. W. Doeppner Jr. "Threads: A system for the Support of Concurrent Programming," Brown Univ, Tech Report CS-87-11, 16 June, 1987.

[26] C. Northrup, "Programming with Unix Threads," John Wiley & Sons, Inc., New York, 1996. ISBN: 0-471-13751-0.

[27] S. Keiman, D. Shah, B. Smaalders "Programing with Threads," Prentice-Hall, New York, 1996. ISBN: 0-13-172389-8.

[28] D. Stein, D. Shah, "Implementing Lightweight Threads," Summer 1992 USENIX Conference, San Antonio, Tx.

[29] H. Lockhart Jr. "OSF DCE Guide to Distributed Applications," pp 16-19, 196-215, McGraw-Hill, New York, 1994. ISBN: 0-07-911481-4.

[30] B. Nichols, D. Buttlar et al "PThreads Programming," O'Reilly & Assoc. Inc., Cambridge, 1996. ISBN: 1-5659-115-5.

[31] W.R. Stevens, "Unix Network Programming," Prentice-Hall, New York, 1990, ISBN: 0-13-949876-1.

[32] B. Lewis, D. Berg "Threads Primer: A Guide to Multithreaded Programming," Prentice-Hall, New York, 1996, ISBN: 0-13-443698-9.

[33] B.Kernighan, D. Ritchie "The C Programming Language," Prentice-Hall, New York, 1989, ISBN: 0-13-110362-8.

[34] S.P. Harbisopn, G. Steele Jr. "C: A Reference Manual" Prentice-Hall, New York, 1991. ISBN: 0-13-110933-2.

[35] M. Zargham "Computer Architecture: Single and Parallel Systems" Prentice-Hall, New York, 1996. ISBN: 0-13-010661-5.

[36] J. Hayes "Computer Architecture and Organization," McGraw-Hill, New York, 1988. ISBN: 0-07-027366-9.

[37] A. Tanenbaum "Structured Computer Organization," Prentice-Hall, New York, 1990. ISBN: 0-13-854662-2.

[38] K. Hwang, F. Briggs "Computer Architecture and Parallel Processing," McGraw-Hill, 1984. ISBN: 0-07-031556-6.

[39] K. Olukotun, B. Nayfeh et al "The Case for a Single-Chip Multiprocessor," ACM Sigplan, Vol 31, No. 9, Sept 1996. pg 2-11.

[40] B. Brey, "The Intel Microprocessors 8086/8088, 80186, 80286, 80386, and 80486: Architecture, Programming, and Interfacing," Merril, New York, 1991. ISBN: 0-02-314250-2.

[41] Schimmel, C. "UNIX Systems for Modern Architectures," Addison-Wesley, New York, 1994. ISBN: 0-201-63338-8.

[42] H. Stone "High-Performance Computer Architecture" Addison-Wesley, New York, 1990. ISBN: 0-201-51377-3.

[43] W. Giles "Assembly Language Programming for the Intel 80XXX Family," Macmillan, New York, 1991. ISBN: 0-02-342990-9.

[44] Intel Corp, "Pentium Processor User's Manual: Architecture and Programming Manual," Vol.3, 1993. ISBN: 1-55512-195-0.

[45] Intel Corp, "Pentium Processor User's Manual: Pentium Processors Data Book," Vol.1, 1993. ISBN: 1-55512-195-0.

[46] Intel Corp, "Pentium Processor User's Manual: Cache Controller and 82491 Cache SRAM Data Book," Vol.2, 1993. ISBN: 1-55512-195-0.

[47] Intel Corp. "8086/8088 User's Manual: Programmer's and Hardware Reference," 1989. ISBN: 1-55512-081-4.

[48] Intel Corp. "Pentium Processor Family Developer's Manual" Vol.1 Pentium Processors, 1995.

[49] J. Hyde "How to make Pentium Pros cooperate," Byte mag., Vol 21, No 4, April 1996, pp 177-178.

[50] B. Catanzaro, "Multiprocessor Architectures," SUN Microsystem, Prentice-Hall, New York, 1994. ISBN: 0-13-089137-1.

[51] Hyundai "Hyundai's Adaptive Crossbar Architecture for 8-Way SMP Pentium Pro Server", white paper, 1997.

[52] Agerwala et al "SP2 system architecture," IBM Systems Journal, Vol. 34, No. 2, 1995, pp 152-184.

[53] H. Custer "Inside Windows NT," Microsoft Press, Redmond, 1993. ISBN: 1-55615-481-X.

[54] Microsoft Corp. "Win32 Programmer's Reference Guide," Microsoft Press, Redmond, Vol 1-5. 1993.

[55] Microsoft Corp. "Run-Time Library Reference: Microsoft C/C++," Redmond, 1991.

[56] J. Richter "Advanced Windows," Microsoft Press, Redmond, 1995. ISBN: 1-55615-677-4.

[57] B. Myers, E. Hamer "Mastering Windows NT Programming," Sybex, New York, 1993. ISBN: 0-7821-1264-1.

[58] S. Dobson, "Concurrent and Networked Programming with Windows NT," Informatics Dept, SERC Rutherford Appleton Lab., Oxfordshire, UK.

[59] J. English "Multithreading in C++," ACM SIgplan notices, Vol 30, No 4, April 1995. pp 21-27.

[60] D.E. Knuth, "The Art of Computer Programming," Addison-Wesley, New York. 1973. ISBN: 0-201-03809-9.

[61] K. Gharachorloo, "Memory Consistency Models for Shared-Memory Multiprocessors," Research Report, Digital Equipment Corp., Western Research Laboratory, 9 Dec., 1995.

[62] Y. Chong, K. Hwang, "Performance Analysis of Four Memory Consistency Models for Multithreaded Multprocessors," IEEE Trans. on Parallel and Distributed Systems, Vol 6, No 10, Oct 1995. pp 1085-1099.

[63] S. Adve, K. Gharachorloo "Shared Memory Consistency Models: A Tutorial," IEEE Computer, Vol. 29, No. 12, Dec 1996, pg 66-76.

[64] P. Stenstrom "A Survey of Cache Coherence Schemes for Multiprocessors," Computer June 1990, pp 12-24.

[65] T. Sterling, P. merkey, D. Savarese "Improving Application Performance on the HP/ Convex Exemplar," Computer, Vol 29, No 12, Dec 1996. pp 50-55.

[66] M. Flynn "Parallel processors were the future...and may yet be" The Open Channel, Computer, Vol 29, No 2, Dec. 1996. pp 151-152.

[67] M. Flynn "Very High-Speed Computing Systems," Proc. IEEE, Vol 54, Dec 1966, pp 12-27.

[68] M. Dubois, C. Scheurich, F. Briggs "Synchronization, Coherence, and Event Ordering in Multiprocessors," Computer, Vol. 21, No. 2, Feb 1988. pg 9-21.

[69] V. Srini, L. Bushnell "A Crossbar System for Multiprocessors," Univ. of Ca., Berkeley, UCB/CSD Technical Report, October 1989

[70] S. Biglow "Understanding Telephone Electronics," SAMS, 1991. pg 31-35. ISBN 0:-672-27350-0.

[71] M. Atkins, R. Subramanian "PC Performance Tuning," Computer Vol. 29, No. 9, August 1996. pp 47-54.

[72] J. Philbin, J. Elder et al "Thread Scheduling for Cache Locality," ACM Sigplan, Vol. 31, No 9, Sept 1996. pp 60-71.

[73] N. Manjikian, T. Abdelrahman "Fusion of Loops for Parallelism and Locality," IEEE trans. on Parallel and Distributed Systems, Vol. 8, No.2, Feb 1997. pp 193-209.

[74] C. Clos "A Study of non-blocking switching networks," Bell Systems Tech. Journal, Vol 32, pp 406-424, Mar 1953.

[75] L. Sterling, E. Shapiro "The Art of Prolog," MIT Press, Cambridge, 1986, ISBN: 0-262-19250-0.

[76] I. Bratko "Prolog Programming for Artificial Intelligence," Addison-Wesley, New York, 1989, ISBN: 0-20114224-4.

[77] W. Clocksin, C. Mellish "Programming in Prolog," Springer-Verlag, New York, 1984, ISBN: 0-387-15011-0.

[78] R. Kowalski "Logic for Problem Solving," North-Holland, New York, 1979, ISBN: 0-444-00368-1.

[79] P. Winston "Artificial Intelligence," Addison-Wesley, New York, 1992, ISBN: 0-201-53377-4.

[80] S. Chottiner "Mathematics for Modern Management," Harper & Row Pub., New York, 1978. pp 338-341. ISBN: 0-06-041265-8.