UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

(1)

AD-A196 694

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 88-60

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

TITLE (and Subtitle): THE CONCURRENT ENVIRONMENT OF THE SEQUENT BALANCE 8000

5. TYPE OF REPORT & PERIOD COVERED: MS THESIS

6. PERFORMING ORG. REPORT NUMBER

AUTHOR(s): FLOYD DAVENPORT

8. CONTRACT OR GRANT NUMBER(s)

PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: TEXAS A&M UNIVERSITY

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE: 1988

13. NUMBER OF PAGES: 134

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): AFIT/NR, Wright-Patterson AFB OH 45433-6583

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583
19 July 88

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1 Introduction

Concurrent programming is an intense area of research in computer science. There are two types of concurrent programming. Multiprogramming refers to multiple processes executing on a single processor in the same time period by using a method called "Time-slicing". Multiprocessing refers to multiple processes which execute at the same time each on its own processor. This paper deals with the issues of multiprocessing. A process is defined as a section of code which is executed sequentially.

Concurrent programming has become popular for two primary reasons. First, the computer hardware industry has been building more and more complex multiprocessing systems at cheaper and cheaper costs. Second, multiprocessing systems enable programmers to build software systems which run at a speed unobtainable on most single processor systems. New complex applications which require such speed involve database systems, large-scale scientific applications, and real time, embedded systems.

However, there is a problem with these new multiprocessing systems. The problem centers around the fact that the software industry has failed to keep pace with the multiprocessing enhancements produced by the hardware industry. A great amount of research has been accomplished and many models of concurrent processing have been developed that deal with the issues of concurrent programming. However, few working systems have been produced which allow the programmer to effectively use the multiprocessing environment. Those which have been produced provide little or no standardization. The main issues of concurrent programming are process creation, synchronization, and communication.

This paper has two objectives. The first objective is to investigate and document the mechanisms for process creation and control on a commercial multiprocessing system. The Texas A&M Sequent Balance 8000 Multiprocessing System is the target of this objective. The second objective is to add a new mechanism to this existing system that easily and clearly expresses process creation.

Section 2 of this paper discusses the hardware of the Sequent Balance 8000 Multiprocessing System. Section 3 introduces the DYNIX Operating System and discusses the mechanisms provided for process creation and control. Examples are included for each mechanism in order to clearly demonstrate the functionality. Section 4 discusses the implementation of a precompiler for the C programming language which provides programmers with the "cobegin–coend" construct. This construct allows programmers to easily create concurrent processes. Section 5 discusses the problem of synchronization and provides software solutions to the mutual exclusion problem.

THE CONCURRENT ENVIRONMENT OF THE SEQUENT BALANCE 8000

by

Floyd Davenport, Captain, USAF

Texas A&M University

Submitted in Partial Fulfillment of the Requirement
for the Degree of
MASTER OF COMPUTER SCIENCE

1 July 1987

Table of Contents

1 Introduction

Concurrent programming is an intense area of research in computer science. There are two types of concurrent programming. Multiprogramming refers to multiple processes executing on a single processor in the same time period by using a method called "Time-slicing". Multiprocessing refers to multiple processes which execute at the same time each on its own processor. This paper deals with the issues of multiprocessing. A process is defined as a section of code which is executed sequentially.

Concurrent programming has become popular for two primary reasons. First, the computer hardware industry has been building more and more complex multiprocessing systems at cheaper and cheaper costs. Second, multiprocessing systems enable programmers to build software systems which run at a speed unobtainable on most single processor systems. New complex applications which require such speed involve database systems, large-scale scientific applications, and real time, embedded systems.

However, there is a problem with these new multiprocessing systems. The problem centers around the fact that the software industry has failed to keep pace with the multiprocessing enhancements produced by the hardware industry. A great amount of research has been accomplished and many models of concurrent processing have been developed that deal with the issues of concurrent programming. However, few working systems have been produced which allow the programmer to effectively use the multiprocessing environment. Those which have been produced provide little or no standardization. The main issues of concurrent programming are process creation, synchronization, and communication.

This paper has two objectives. The first objective is to investigate and document the mechanisms for process creation and control on a commercial multiprocessing system. The Texas A&M Sequent Balance 8000 Multiprocessing System is the target of this objective. The second objective is to add a new mechanism to this existing system that easily and clearly expresses process creation.

Section 2 of this paper discusses the hardware of the Sequent Balance 8000 Multiprocessing System. Section 3 introduces the DYNIX Operating System and discusses the mechanisms provided for process creation and control. Examples are included for each mechanism in order to clearly demonstrate the functionality. Section 4 discusses the implementation of a precompiler for the C programming language which provides programmers with the "cobegin–coend" construct. This construct allows programmers to easily create concurrent processes. Section 5 discusses the problem of synchronization and provides software solutions to the mutual exclusion problem.

2 Sequent Balance 8000 Architecture

The Sequent Balance 8000 is a tightly coupled, symmetric, multiprocessor computer with a common pool of shared memory. **Sequent Computer Systems, Inc.** released the Balance 8000 in 1984. The Balance 8000 supports both general purpose, multiuser applications and dedicated parallel applications. The Balance 8000 is based on a Shared Memory Architecture. This means that processes communicate by reading and writing to shared data structures. Processes can execute on any CPU, independent of any other process. The processes use shared memory to communicate and to synchronize activities.

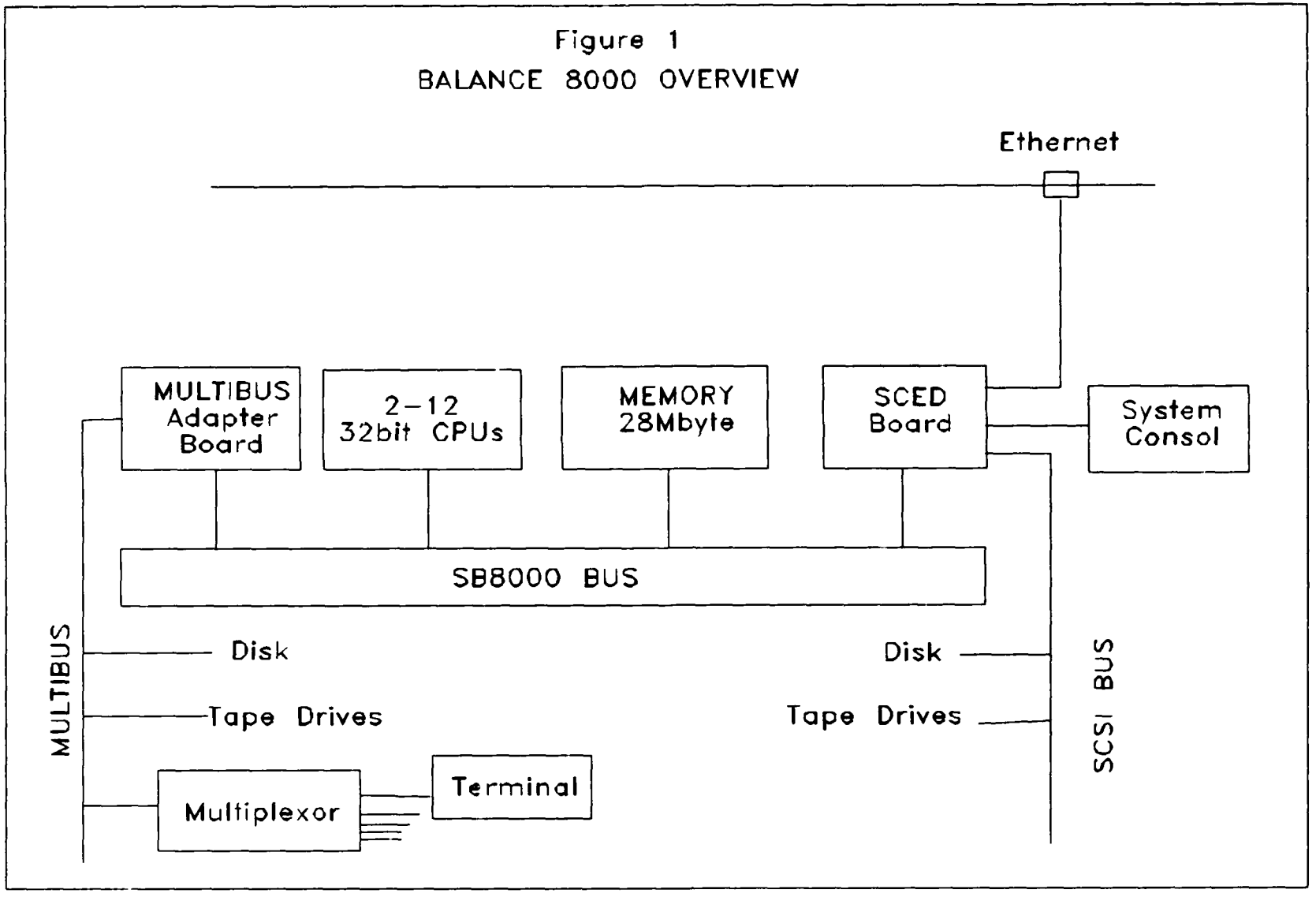
The Balance 8000 operating system, DYNIX, is a version of UNIX 4.2bsd that has been enhanced to provide features of UNIX System V and to exploit the features of the parallel architecture. DYNIX includes a Parallel Programming Library that simplifies the use of shared memory and the system's hardware–based mutual exclusion mechanisms. It also distributes the responsibility of scheduling, handling interrupts, and housekeeping duties among the CPUs.

The Balance 8000 is an expandable, high performance parallel computer. The Balance 8000 has a chassis which can contain 12 card slots into which component boards are placed and configured. The following boards can be used: MULTIBUS adapter board, CPU, Memory module, or a SCED board. The Balance 8000 includes three buses, one system bus and two I/O buses.

2.1 SB8000 Bus

The Balance 8000 is built around a 32 bit wide bus called the SB8000. This bus links the system's CPUs, system memory, and I/O subsystems as shown in Figure 1. The SB8000 supports data packets of 1, 2, 3, 4, or 8 bytes and has a channel bandwidth of 40-Mbytes per second with a sustained data transfer rate of 26.7 Mbytes per second. Optimal performance is obtained by using data packets of 4 or 8 bytes. This common data bus greatly simplifies the addition of system components.

Figure 1
BALANCE 8000 OVERVIEW



2.2 Processor Boards

The Balance 3000 can have from two to twelve NS32032 CPUs (packaged two to a board). The CPUs run at 10MHz and include a floating point unit, a memory management unit, an 8 Kbyte local memory, and an 8 Kbyte cache. Each CPU shares memory via the SB8000 (see Figure 2). Each CPU is identical and can execute both user code and kernel code. Each CPU issues a 24 bit virtual address (every process can access up to 16 Mbytes). The CPU's Memory Management Unit translates that address into a 25 bit physical address. The SB8000 supports a 28 bit address and uses the higher order bits to address the different I/O subsystems. The local memory holds highly used kernel code and data structures to decrease contention on the SB8000. The cache memory also reduces the contention for the SB8000 bus. Cache data is organized into 512 rows each with two eight byte blocks. If a read miss occurs on a processor's cache, a new block of data is read from memory and replaces the least recently used block of cache. If a CPU wishes to write to memory, it will first update its cache if the block resides in the cache. It will then send a write request to the SB8000 bus to update the block in memory. Each processor monitors all writes to memory. If write to memory from another processor addresses a block in cache, the block is marked as out-of-date and a read miss will occur next time it is accessed. The Computer Science Department's system currently contains 10 CPUs.

2.3 Memory Modules

The system can support up to four memory modules with a total of 28 Mbytes of physical memory. An individual process can access up to 16 Mbytes of virtual memory. Each memory module consists of a memory controller (which contains 2 Mbytes of RAM), and optionally, a memory expansion board with 2–6 Mbytes. A memory controller and expansion board occupy one slot in the Balance 8000 chassis. Each memory module can respond to a read request in 300 ns (3 cycles, 2 cycles for a 4 or 8 byte read or write request). Multiple operations are pipelined to enhance performance. Memory modules can also be interleaved if equal sized memory modules are used. It would appear that the Balance 8000 can access up to 32 Mbytes of memory (25 bit physical address and four memory modules with 8 Mbytes each). However, one Mbyte of memory is reserved for each of a possible four MULTIBUS adapter boards.

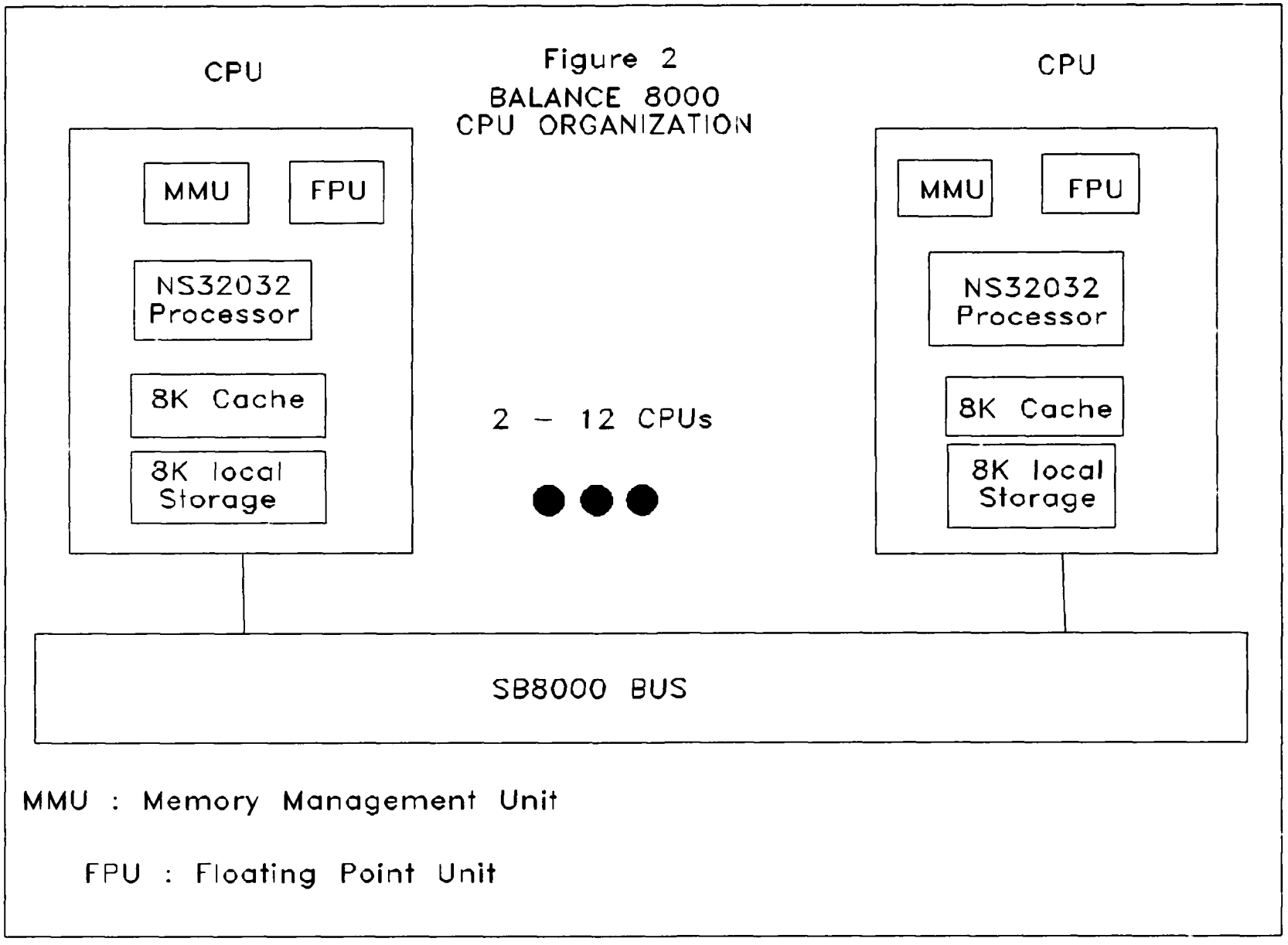


Figure 2
BALANCE 8000
CPU ORGANIZATION

2.4 SCED Board

The Balance 8000 requires at least one SCED board and can contain up to four. The SCED board supports many functions. It connects to the Small Computer Systems Interface (SCSI) bus. This bus is designed to support high speed, high volume data transfers between memory and peripherals such as disk and tape drives. The SCED board allows the Balance 8000 to connect to other systems in the local area using Ethernet. It is used to perform system startup and system diagnostics. The SCED board also provides a RS232-C interface to connect the system console. The SCED board packages data into eight byte blocks to efficiently use the SB8000 bus.

2.5 MULTIBUS Adapter Board

The Balance 8000 can include up to four MULTIBUS adapter boards. The MULTIBUS adapter board connects to MULTIBUS, a general purpose bus protocol that supports a wide variety of terminals, printers, disk units, and tape drives. Peripherals can include RS232-C compatible devices such as a one-half inch tape drive or a 396 Mbyte disk drive. These peripherals can be connected via one or more terminal multiplexors on the MULTIBUS.

2.6 SLIC Bus

The SB8000 includes an independent one bit data path called the System Link and Interrupt Controller (SLIC). This bus is for low level communication (interrupts) between system components. The SLIC bus supports a high speed, synchronous, bit serial, protocol. Every component board on the Balance 8000 includes a Sequent designed VLSI SLIC chip. All SLIC chips are connected to the SLIC bus to manage interprocessor communication, access to kernel data structures, interrupts, diagnostics, and configuration control. Only one operation can be performed on the SLIC bus at a time. If two CPUs both try to use the SLIC bus at the same time, the one with the lowest priority will wait. If both CPUs have the same priority, the one with the highest CPU number succeeds. The CPU priority is based on the priority of the process currently executing on the CPU. To ensure that a CPU will eventually access the SLIC bus, priorities are updated once every second. Processes which have been idle longest receive higher priorities.

2.7 Mutual Exclusion

In any multiprocessor system based on a shared memory architecture, mutual exclusion is an issue. Mutual exclusion ensures that a sequence of operations acts as an indivisible operation. Any operation on a shared variable should be completed before another process accesses that shared variable. The Balance 8000 solves the mutual exclusion problem by providing programmers with a set of hardware locks. The Balance 8000 can have up to 64K hardware locks (16K locks for each MULTIBUS configured). These locks are physically located on the MULTIBUS Adapter Boards and are known as Atomic Lock Memory (ALM). Each time a lock is accessed, a test-and-set operation is performed. This operation is an atomic operation which will test the state of the lock (LOCKED or UNLOCKED), LOCK the lock if it is UNLOCKED, and return its state. The main purpose of the hardware locks is to ensure mutual exclusion on a set of virtual software locks. The software locks are created by the programmer and placed in an application's shared memory. An application can create as many software locks as will fit in its shared memory. These software locks ensure the mutual exclusion needed by an application for its critical sections. The software locks work the same as the hardware locks, except their operations are not atomic. A process must first obtain a hardware lock before accessing its software locks to ensure that the software lock's operations are atomic. After a process has obtained a hardware lock, it may perform an operation on its software lock. The process will then release the hardware lock for other processes to access. To ensure that multiple processes attempting to obtain the same software lock first obtain the same hardware lock, a relationship must be set up between the two types of locks. DYNIX accomplishes this through a hashing algorithm, where the address of the software lock is hashed to an address of a hardware lock. This implies that many unrelated applications may try to obtain the same hardware lock. Although, this may slightly effect the run time of an application, mutual exclusion is still ensured. The routines found in the DYNIX Parallel Programming Library enable programmers to create and use software locks. By using these routines the operations on the hardware locks become transparent.

3 Parallel Programming Library

This section introduces the routines which comprise the Parallel Programming Library. These routines support multitasking in C, Pascal, and FORTRAN. The library is located in */usr/lib/libpps.a*. These routines can be linked to a program from the library by including the -lpps option in the cc command for C programs, the -mp option when compiling Pascal, or the -F option when compiling FORTRAN programs. The following discussion and examples are limited to the C programming language. Not every routine in the Parallel Programming Library is covered, but most of the routines are discussed and examples are included to illustrate their use. In addition to the routines found in the Parallel Programming Library, the **fork, exit,** and **wait** routines are explained. These three routines are found in any current version of Unix and provide a simple mechanism to create multiple concurrent processes.

DYNIX includes two C header files which contain declarative statements for the Parallel Programming Library routines. Both of these header files reside in the directory */usr/include/parallel*. The header files are named *microtask.h* and *parallel.h*. Refer to Section 3P in the DYNIX Programmer's Manual and Appendix C in this document for information on which file to include for each routine. These files are included in each of the examples that illustrate routines from the Parallel Programming Library.

DYNIX uses two terms to describe parallel programming, microtasking and multitasking. The terms relate to two different methods which are used to partition a program for parallel execution. Microtasking refers to the idea of "Data Partitioning". In this method, a set of data is partitioned into subsets where separate identical processes are created to perform the desired work on each subset of data. The key word is "identical". Each process is an exact duplicate of every other process. The only difference is that each process will work on different data. The classic example is an iterative loop, where each iteration accesses a different set of data. Almost all of the routines in the Parallel Programming Library seem to be geared for this technique. Multitasking refers to the idea of "Functional Partitioning". In this method, functions are separated instead of data. This usually requires a more flexible and dynamic approach. The **fork** routine is used for this type of partitioning. It is a misconception to think that the Parallel Programming Library routines can handle every type of multiprocessing application. The new DYNIX parallel programming routines were never meant to replace the basic **fork**, but to extend its capabilities in certain contexts.

The use of shared memory is also often misunderstood. In the C programming language, a global variable is not shared between separate processes (however, in Pascal, global variables are shared). Any variable or structure which is to be shared between processes must be declared as **shared**. The key word **shared** must proceed the type of a variable within its declaration. You **must** link a program with the Parallel Programming Library to use shared memory, even if you do not use any routine from the library. Remember, it is **your** responsibility to provide for any synchronization needed between processes to ensure correctness.

3.1 Fork

The **fork** creates a new process. The new process's instructions, user-data, and system-data segments are almost exact copies of those of the old process. The old process which issued the **fork** is called the "parent" and the newly created process is called the child. The only difference between the two processes is that the child has a unique process id (PID) and a different parent process id (the PID of the old process). The **fork** returns an integer. After the **fork**, both processes (the parent and the child) receive a return. The parent process will receive the PID of the newly created child. The child will receive a 0.

Example 1 shows a process which issues a **fork** to create a child. Both processes then print out what was returned by the **fork**. The output of the example follows and shows two numbers returned by the **fork**, 12538 and 0. 12538 is the PID of the child and was returned to the parent. O was returned to the child.

Notice the system call **setbuf**. This command sets the size of the buffer which writes to a file. I used the command to set the buffer size of the standard output file (terminal) to zero (NULL). When a parent creates a child, the child gets a copy of the parent's open file descriptors. This means that each can overwrite what the other has written by writing to the same buffer. The buffer size is set to zero, so that what ever is written by parent or child immediately goes to the terminal. To fully ensure that no output is lost, one must perform I/O within mutually exclusive areas called critical regions. Mutual exclusion will be demonstrated later.

```
                    /***************/
                    /*  Example 1  */
                    /***************/
#include <stdio.h>

/*  This program demonstrates the Fork system call.  Fork will create  a */
/*  new process known as the child process.  The parent process is the   */
/*  process which executed the Fork.  The child process is almost an     */
/*  exact copy of the parent.  Both the child and parent process will    */
/*  resume execution after the Fork.  Fork will return the value of 0 to */
/*  the child process and will return the Process ID (pid) of the child  */
/*  to the Parent process.  In this example both processes print out the */
/*  value returned from the Fork.                                        */

main ()
{
      int pid;
      setbuf (stdout, NULL);
      printf ("Start of Test\n");
      pid = fork ();
      printf ("pid returned is: %d\n", pid);

}
```

```
Start of Test
pid returned is: 12538
pid returned is: 0
```

3.2 Getpid

Any process may find out its PID by issuing a call to **getpid**. The following example shows how to issue the call to **getpid** and print a process's PID. In example 2, the parent first calls **getpid** and prints out its PID. The parent then issues a **fork**. Notice that the **fork** is within an *if* statement. If the **fork** returns a zero, **getpid** is used to obtain and print the PID. This is the normal method of designating the child and the section of code the child is to execute. The parent, who receives a nonzero response, skips that section of code and terminates.

```
                        /***************/
                        /*  Example 2  */
                        /***************/
#include <stdio.h>

/*    This program will use the system call "getpid" to find out the  */
/*    process's ID.  After printing the PID, the process will Fork a  */
/*    new process.  The new child process will also call "getpid" and */
/*    print the result.  Each process created under UNIX has a unique */
/*    PID.                                                            */

main ()
{
       int pid;

       pid = getpid ();
       printf ("Parent Process is %d\n", pid);
       if (fork () == 0) {
          pid = getpid ();
          printf ("Child Process is %d\n", pid);
       }

}
```

```
Parent Process is 10844
Child Process is 10845
```

3.3 Exit and Wait

The **exit** routine terminates a process. A process which performs an **exit** may pass (as a parameter) a short integer value back to the parent which acts as a termination code. The value zero is usually returned to the parent to indicate a normal termination. If the process finds an error, it may wish to exit with a termination code (such as -1) which indicates to the parent process that an error occurred. One may **fork** a child process to execute some segment of code embedded within the program. To accomplish this task, **fork** the child and test for a return of zero as done previously. The child should execute the section of code within the *if* statement. Place an **exit** statement at the end of the *if* statement to terminate the child process. Otherwise, the child process will continue past the *if* statement and execute code which the parent is executing.

The **wait** routine is used by a parent process to wait until a child process has terminated. The **wait** returns the PID of the child which terminated. It also returns the termination code of the child process (normally passed back by the child using **exit**). This code is placed in an integer variable which is passed to the **wait** routine in a parameter. If a process issues a call to **wait** and has no children processes executing, **wait** immediately returns a -1 as the PID. A parent process can not **wait** for a specific child to terminate. If any child terminates, the **wait** is satisfied. If the parent knows the PID of the child, it can test the PID returned by **wait** to find out which child terminated. The parent may have to **wait** through several terminations to find a specific termination. There is no requirement that a parent must wait for a child process, but the parent may choose to wait if it is dependent on some action taken by the child process. In this capacity, the **wait** acts as a synchronization mechanism.

Example 3 uses both **exit** and **wait**. *status* is used to hold the returned termination code of a terminating process. This aspect of **wait** and the **Union** statement is discussed in the next example. This program also demonstrates the capability to nest **fork** calls. In this program, the parent process gets and prints its PID (by use of **getpid**). It then **forks** a child process, prints the message "Parent Running", and issues a **wait** for the child. The child process also gets and prints its PID; **forks** a child process (the grandchild); prints the message "Child Running"; and **waits** for the grandchild. The grandchild process gets and prints its PID, prints the message that it is terminating, and then terminates. After the grandchild terminates, the child prints the PID of the grandchild that terminated. It

then prints a message that it is terminating and does so. The parent process then prints the PID of the child that terminated and terminates.

This program has two interesting points. The first is the order of the output. Notice that the parent is still executing as the child executes and the child is still executing as the grandchild executes (concurrency). However, the child blocks execution until the grandchild has terminated and the parent blocks until the child has terminated. The second point is that the termination of a process is only seen by its immediate parent. In other words, the parent process did not notice the creation, execution, or termination of the grandchild process.

```
                        /***************/
                        /*  Example 3  */
                        /***************/
#include <stdio.h>
#include <sys/wait.h>

/*    This program creates three processes.  The main process will print */
/*    its PID and Fork a new process.  The child process prints its PID  */
/*    and Forks a new process.  The Grandchild process prints its PID    */
/*    and Exits.  Both the parent and child process print a message to   */
/*    show that they are executing concurrently with their children.     */
/*    The Parent then performs a Wait on the Child and the Child         */
/*    performs a Wait on the Grandchild.  When a process terminates with */
/*    an Exit system call, the termination status is returned to the     */
/*    parent process.  The Wait system call also returns the PID of the  */
/*    terminating child process.  In this example the Parent prints the  */
/*    PID of the terminating child process.                              */

main ()
{
       union wait status;
       int pid;

       setbuf (stdout, NULL);
       pid = getpid ();
       printf ("Parent Process is %d\n", pid);
       if (fork () == 0)  {                          /* Fork the Child */
          pid = getpid ();
          printf ("Child Process %d\n", pid);
          if (fork () == 0)  {                       /* Fork the Grandchild */
             pid = getpid ();
             printf ("Grandchild Process is %d\n", pid);
             printf ("Grandchild Exits\n");
             exit (0);                              /* Grandchild Terminates */
          }
          printf ("Child Running\n");
          pid = wait (&status);                      /* Wait for the Grandchild */
          printf ("Grandchild %d Finished\n", pid);
          printf ("Child Exits\n");
          exit (0);                                 /* Child Terminates */
       }
       printf ("Parent Running\n");
       pid = wait (&status);                        /* Wait for the child */
       printf ("Child %d Finished \n", pid);
       printf ("Parent Exits\n");
}
```

```
Parent Process is 10857
Parent Running
Child Process 10858
Child Running
Grandchild Process is 10859
Grandchild Exits
Grandchild 10859 Finished
Child Exits
Child 10858 Finished
Parent Exits
```

3.3.1 Detecting Errors using Exit and Wait

How does one use the **wait** routine to catch a bad termination code? Study example 4. In this example a process **forks** a child process and immediately **waits** for that child to terminate. The child process gets and prints its PID and then terminates with an abnormal termination code of 3 (remember that 0 is a normal termination). After the child terminates, the parent tests the *status* for normal termination. The variable *status* is used to hold the termination code returned by **wait**.

There are two ways for a process to terminate. It can call **exit** or it can receive a fatal signal from the system. Although *status* is just an integer, its bits indicate how it terminated. If the rightmost byte of status is zero, then the byte to its left is the child's argument to **exit**. If both are zero, the child terminated normally. If the rightmost byte is nonzero, the first seven bits are the signal number that terminated the child. If the eighth bit is 1, a core dump was taken. The bits of *status* are counted right to left (15, 14, ... 2, 1, 0).

In this example, *status* is declared as type *union wait*. This union has a structure of the three bit fields just described. This allows one to check and print the different codes without performing shift operations. In order to use this union declaration, include the header file *sys/wait.h*. This header file defines the **union** of *status*. In the example, the parent checks *status.w_status*. This is declared in *union wait* as the entire integer. If it is zero, the child terminated normally. If it is not zero, the parent prints a message that the child terminated abnormally and then checks *status.w_termsig*. If this is zero, the child placed the termination code in **exit** and the parent prints this code. If it is nonzero, the child terminated from a fatal signal and the parent prints the signal number. The parent will also check for a core dump. Notice in the output that the termination code of 3 was printed. The **man** command can be used to reference the DYNIX Programmers Reference Manual. The command "man 2 wait" can be used to gather more information on the **wait** routine.

```
                        /***************/
                        /*  Example 4  */
                        /***************/
#include <stdio.h>
#include <sys/wait.h>

/*  This program illustrates the use of the Status value returned */
/*  by the Wait system call.  Once again, you must include the    */
/*  header file <sys/wait.h>.  The Status returned is actually an */
/*  unsigned short integer.  The rightmost 8 bits are set if the  */
/*  operating system terminated the process.  Bits 0 - 6 give the */
/*  terminating code.  Bit 7 is set if a core dump was taken.  If */
/*  the rightmost byte is 0 and the process still terminated      */
/*  abnormally, then bits 8 - 15 contain the status code returned */
/*  by the process through the Exit system call.  In this example */
/*  the child process returns a value of 3.  This allows a user   */
/*  to set up his own termination codes. The Bits of the Status   */
/*  are counted right to left (15, 14 ... 3, 2, 1, 0).            */

main ()
{
      union wait status;  /*  bit field set up by <sys/wait.h> */
      int pid;

      setbuf (stdout, NULL);
      pid = getpid ();
      printf ("Parent Process is %d\n", pid);
      if (fork () == 0)  {                          /* Fork the Child */
         pid = getpid ();
         printf ("Child Process %d\n", pid);
         exit (3);                          /* Child Terminates Abnormally */
      }
      pid = wait (&status);                     /* Wait for the child */

      if (status.w_status != 0)  {              /* Abnormal Termination? */

         printf ("\n\nProcess %d had Abnormal Termination\n", pid);

         if (! status.w_termsig)                /* Terminated by System? */
            printf ("Exit Code: %u\n", status.w_retcode);
         else  {
            printf ("Terminated by System Error: %u\n", status.w_termsig);

            if (status.w_coredump)               /* Core Dump Taken? */
               printf ("Core Dump Taken\n");
         }
      }
}
```

```
Parent Process is 10874
Child Process 10875


Process 10875 had Abnormal Termination
Exit Code: 3
```

3.3.2 Detecting Errors on Forks

When a process **forks** another process, it would be nice to know whether the **fork** was successful. Also, how many processes can one user have executing at one time and how does one detect an error? Example 5 shows how many **forks** can be performed successfully and how to catch an unsuccessful **fork**. In this program, the parent process will print its PID and then start **forking** new children. Each child goes into an infinite loop. This is done to ensure that no child terminates while the parent is still creating new children. The parent keeps count of the number of children created and prints each child's PID. When a **fork** does not succeed, it returns a -1 instead of a new PID. When the parent receives a -1 from a **fork**, it *breaks* out of the loop. The next problem is to determine why the **fork** was unsuccessful.

Notice the two external variables *sys_nerr* and *sys_errlist*. *sys_errlist* is an array of error messages kept by the system. *sys_nerr* is the highest index into the array *sys_errlist*. The external variable *errno* is declared in the header file *errno.h* and will hold the error message number of the unsuccessful **fork**. In this example, when the parent process finds a bad **fork**, it prints the number of children created. It then checks *errno* for an error code. If an error code is found and it is less than *sys_nerr*, *errno* is used as an index into *sys_errlist* to print the error message. The output of this example shows that a user is able to create 25 processes (including the parent). The command **kill** at the bottom of the program allows the parent to terminate all related processes. The **kill** will also terminate the parent process. The command "man 3 sys_nerr" can be used to read about system error messages and the command "man 2 intro" can be used to list every possible error message.

```
                    /***************/
                    /*  Example 5  */
                    /***************/
#include <stdio.h>
#include <errno.h>

/*  This program demonstrates the use of the external variables  */
/*  errno, sys_nerr,sys_errlist to capture an unsuccessful Fork. */
/*  In this program, as many processes are created as the system */
/*  will allow one user to create.  When the system does not     */
/*  allow any more Forks by a user, it returns a -1 in place of  */
/*  the PID and puts the error code into the global variable     */
/*  errno.  errno can be used as an index into the array         */
/*  sys_errlist.  This array holds error messages for each error */
/*  produced by the system.  This program counts the number of   */
/*  processes created and then prints the error message returned */
/*  from the bad Fork.  Each Child process goes into an infinite */
/*  loop to ensure that no-one terminates while I am testing for */
/*  the error condition.  The system call to Kill is used to     */
/*  terminate all children processes.  The 0 says to signal all  */
/*  processes in my user's goup (including myself), the 9 is the */
/*  terminate signal.                                            */

main ()
{
      extern int sys_nerr;
      extern char *sys_errlist[];
      int count, ppid, pid;

      setbuf (stdout, NULL);
      count = 0;
      ppid = getpid ();
      printf ("Parent Process is %d\n", ppid);
      printf ("\n\nCount    Process\n");
      for (;;)  {
         pid = fork ();
         if (pid == 0)                /*  child spins  */
            for (;;) {  }
         if (pid > 0)  {              /*  Increment count and  */
            count ++;                 /*  print PID of child   */
            printf ("  %d        %d\n", count, pid);
         }
         else if (pid < 0)            /*  Error on Fork  */
            break;
      }
      printf ("\nTotal Processes Created: %d\n", count);
      printf ("\nError on Fork\n");
      if ((errno > 0) && (errno < sys_nerr))
         printf ("%s \n", sys_errlist[errno]);
      kill (0,9);                                 /*  kill the children  */
}
```

```
Parent Process is 10887

Count     Process
  1         10888
  2         10889
  3         10890
  4         10891
  5         10892
  6         10893
  7         10894
  8         10895
  9         10896
  10         10897
  11         10898
  12         10899
  13         10900
  14         10901
  15         10902
  16         10903
  17         10904
  18         10905
  19         10906
  20         10907
  21         10908
  22         10909
  23         10910
  24         10911

Total Processes Created: 24

Error on Fork
No more processes
```

3.4 M_Fork and M_Kill_Procs

The **m_fork** routine creates a number of child processes and assigns the same subprogram to each of them. The number of children created will be the number of processors available divided by two. Each of the child processes will execute the subprogram passed to **m_fork** as a parameter. After a child has executed the subprogram, it spins waiting for the parent to assign it another subprogram via a new **m_fork**. The **m_kill_procs** routine terminates all child processes which were previously created by an **m_fork**.

In example 6, the main procedure creates five child processes (the system had 10 available processors). Each child executes the function "sayhi". After each child process has executed the function, the parent terminates them with a call to **m_kill_procs**. In the function "sayhi", each child process prints the message "Hello from process PID". Notice that the example includes the two header files *parallel/microtask.h* and *parallel/parallel.h*.

Remember that each child process has a copy of its parent's open file descriptors and can thus overwrite another's output. To ensure that no output is lost, a critical region is used. Only one process can enter the critical region at a time. This is accomplished by using the **m_lock** and **fflush** routines. The **m_lock** routine sets up a hardware lock which only one process at a time can access. Once a process obtains the lock, it can enter its critical region. If it does not obtain the lock, it spins and continues to try to access the lock. A process which has access to the lock releases it by calling the routine **m_unlock**. The routine **fflush** is used to flush the output buffer so that no process will overwrite any previous output. **m_lock** is used at this time only to print output. It is discussed in more detail later.

```
                        /***************/
                        /*  Example 6  */
                        /***************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

/*  This program creats a number of child processes, each executing    */
/*  the procedure "Sayhi".  The procedure "sayhi" prints out the       */
/*  message "Hello from Process PID.                                   */
/*  M_fork will create N processes where N is (the number of           */
/*  available processors) / 2.  The m_kill_procs system call will      */
/*  terminate any processes created by m_fork.                         */
/*  The use of m_lock and m_unlock are used here to ensure mutual      */
/*  exclusion when each process is printing.  Each process shares the  */
/*  same file pointer to stdout, so one process may overwrite another  */
/*  process's output.  The ffiush routine is used to empty the buffer  */
/*  to stdout before another process can overwrite the buffer.         */

sayhi ()
{
       int pid;

       m_lock ();
       pid = getpid ();
       printf ("Hello from Process %d\n", pid);
       fflush (stdout);
       m_unlock ();
}

main ()
{

   m_fork (sayhi);
   m_kill_procs ();

   printf ("Program Over\n");

}
```

```
Hello from Process 10944
Hello from Process 10948
Hello from Process 10947
Hello from Process 10945
Hello from Process 10946
Program Over
```

3.4.1 Fork versus M_Fork

It is a misconception to think that the call to **m_kill_procs** sets up a barrier that the parent will not pass until each child has terminated. Example 7 is exactly as the previous one except for a print statement between the calls to **m_fork** and **m_kill_procs**. Actually, the **m_fork** itself sets up a barrier. After an **m_fork** has been issued, the parent will not continue until each child has executed the designated function. In the output, we see that the message "Program Over" is not printed until after each child has printed the message "Hello from process PID". This is a major difference between **fork** and **m_fork**. The **fork** routine allows the parent to continue executing while the children execute. In fact, with **m_fork** the parent does execute while the children are executing, but as one of the children! In the previous examples, five new processes were **not** created. Four processes were created and the fifth execution of "sayhi" was performed by the parent. This will become important when the **m_single** and **m_multi** routines are discussed. To show that the parent process actually performs as a child when using **m_fork**, notice that the parent's PID, 10977, is also the PID of one of the child processes.

```
                                /***************/
                                /*  Example 7  */
                                /***************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

/*  This program creates a number of processes which print out the  */
/*  message "Hello".  m_lock and m_unlock are used to ensure mutual */
/*  exclusion while each process is printing.  In this program,     */
/*  notice that a call to printf is made before the created child   */
/*  processes are terminated by m_kill_procs.  This shows a basic   */
/*  difference between fork and m_fork.  When an m_fork is made,    */
/*  the parent process becomes one of the child processes and       */
/*  prints one of the "Hello's".  The parent process does not       */
/*  continue executing after the m_fork call until all the created  */
/*  child processes have completed execution of the m_forked        */
/*  procedure.  Also notice that one of the children processes has  */
/*  the same PID as the parent process.                             */

sayhi ()
{
      int pid;
      m_lock ();
      pid = getpid ();
      printf ("Hello from Process %d\n", pid);
      fflush (stdout);
      m_unlock ();
}

main ()
{
   int pid;

   pid = getpid ();
   printf ("Parent's PID is %d\n", pid);
   fflush (stdout);
   m_fork (sayhi);
   printf ("Program Over\n");
   m_kill_procs ();
}
```

```
Parent's PID is 10977
Hello from Process 10977
Hello from Process 10981
Hello from Process 10978
Hello from Process 10979
Hello from Process 10980
Program Over
```

3.5 M_Set_Procs and CPUS_Online

When a **m_fork** is executed, the number of processes which execute the designated subprogram is the number of available processors divided by two. In fact, this is only the default value and the programmer has more control than just using this default. The **cpus_online** routine returns the number of available processors on the system. The **m_set_procs** routine declares the number of processes to execute a designated subprogram in parallel on subsequent calls to **m_fork**. The total number of processes which can be running in parallel using the **m_fork** routine is the number of processors online minus one. If a programmer tries a higher number, the default is used.

In example 8, the main process finds out the number of processors online using **cpus_online** and prints the value. It then uses **m_set_procs** to set the number of processes which can execute in parallel to this value minus one. Again, the function "sayhi" is executed by calling **m_fork**. The output shows that ten processors were online at the time and that "sayhi" was executed nine times.

```
                        /***************/
                        /*  Example 8  */
                        /***************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

/*  This program is identical to the previous programs except    */
/*  it sets the number of processes to create.  The cpus_online */
/*  system call returns the number of available CPUs.  The      */
/*  m_set_procs system call will set the number of processes to */
/*  create on each m_fork.  In this example, one less than the  */
/*  number of CPUs available are created.                       */

sayhi ()
{
      int pid;

      m_lock ();
      pid = getpid ();
      printf ("Hello from Process %d\n", pid);
      fflush (stdout);
      m_unlock ();
}

main ()
{
   int num_cpus;

   num_cpus = cpus_online ();
   printf ("Number of Available CPUs is %d\n", num_cpus);
   fflush (stdout);
   m_set_procs (num_cpus - 1);

   m_fork (sayhi);
   m_kill_procs ();
   printf ("Program Over\n");
}
```

```
Number of Available CPUs is 10
Hello from Process 11001
Hello from Process 11007
Hello from Process 11002
Hello from Process 11004
Hello from Process 11005
Hello from Process 11008
Hello from Process 11003
Hello from Process 11006
Hello from Process 11009
Program Over
```

3.6 M_Get_Myid

Processes created by a call to **m_fork** also have a version of **getpid**. The **m_get_myid** routine returns the PID of the calling process. The PIDs are not the same PIDs that are found using the **getpid** routine. When N processes are created using **m_fork** to execute a subprogram, the PIDs range from 0,1,2 ... N-1. The parent process (which also executes the subprogram) has the PID of 0. The fact that these PIDs are not always unique between different users implies that they are not the real PIDs seen by the system kernel, but PIDs used by some executive module which oversees the execution of **m_fork** and the other microtasking routines. The PIDs become quite useful when partitioning a program by the iterations of a loop.

Example 9 shows a process which executes the function "getpid" three times. Each process will get their PID using the **m_get_myid** routine and print its value. Each process also gets their PID using **getpid**. Notice the output shows the PIDs from **m_get_myid** to run from 0 to 2. Notice that the PIDs from **getpid** are different from those using **m_get_myid**. Also, the PID of the parent process matches the **getpid** PID (12617) of the process with the **m_get_myid** PID of 0.

```
                        /***************/
                        /*  Example 9  */
                        /***************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

/*  This program creates three child processes.  Each process will  */
/*  get its process id by using the system call m_get_myid.  Each    */
/*  process prints out its PID.  The m_lock and m_unlock is used to */
/*  ensure mutual exclusion for printing.  The number of processes  */
/*  created by the system call m_set_procs.  When a m_fork is       */
/*  executed, NPROCS copies of the procedure "getid" are executed.  */
/*  Actually, only NPROCS - 1 new processes are created.  The       */
/*  parent process will act as one of the new processes and execute */
/*  one copy of "getid".  The parent process will have the PID of 0 */
/*  and the other processes will have PIDs of 1,2,3, ... NPROCS-1.  */
/*  Each child process will also get its PID using the getpid       */
/*  routine.  Notice that the two are different.                    */

getid ()
{
      int pid1, pid2;

      pid1 = m_get_myid ();          /*  Get my process ID  */
      pid2 = getpid ();
      m_lock ();
      printf ("My process ID using m_get_myid is: %d\n", pid1);
      printf ("My process ID using getpid is: %d\n", pid2);
      fflush (stdout);
      m_unlock ();
}

main ()
{
   int pid;

   pid = getpid ();
   printf ("Parent process is %d\n\n", pid);
   fflush (stdout);
   m_set_procs (NPROCS);   /* Set the number of processes to NPROCS  */
   m_fork (getid);
   m_kill_procs ();        /*  Terminate all processes except the parent  */
}
```

```
Parent process is 12617

My process ID using m_get_myid is: 0
My process ID using getpid is: 12617
My process ID using m_get_myid is: 1
My process ID using getpid is: 12618
My process ID using m_get_myid is: 2
My process ID using getpid is: 12619
```

3.7 M_Lock and M_Unlock

The routines **m_lock** and **m_unlock** have already been shown to ensure mutual exclusion. The mutual exclusion is not just for printing output, but for any section of code which could produce incorrect results if executed concurrently. The **m_lock** routine creates and initializes a hardware lock. This same lock is used in every copy of the subprogram being executed. When a process calls **m_lock**, a type of test-and-set operation is performed on the lock. If no-one is using the lock, the calling process obtains the lock and may proceed. If the lock is in use, the calling process spins, trying to obtain the lock. A process releases the lock by calling **m_unlock**.

Example 10 illustrates the use of **m_lock** and **m_unlock**. This program increments a counter which is shared between three processes. Remember the counter must be declared as shared for it to be placed in shared memory and accessible by each process. The main process creates three copies of the subprogram "counts" to be executed in parallel. Each process will attempt to increment the counter three times. The processes use **m_lock** to ensure that only one process can increment the counter at a time. A process will print its PID and the value of the counter after increasing the counter. The output shows the counter was incremented nine times. Notice the increments of the counter were printed in order. This demonstrates the mutual exclusion. However, notice that the order a specific process obtained the lock, incremented and printed the counter, and released the lock was arbitrary.

```
                        /****************/
                        /*  Example 10  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*    This program illustrates the use of m_lock.  The program counts  */
/*    by increasing the value of a variable in parallel.  The program  */
/*    maintains mutual exclusion with a call to m_lock.  Each copy of  */
/*    the procedure increments the shared variable "count"              */
/*    concurrently, and therefore, mutual exclusion is required.       */

counts ()
{
   int me, i;
   me = m_get_myid ();              /*  Get my PID  */

   for (i=0; i < NPROCS; i++)  {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{
   count = 0;

   m_set_procs (NPROCS);        /*  Create NPROCS processes  */
   m_fork (counts);
   m_kill_procs ();             /*  Terminate all Processes except Parent  */

   printf ("counter over\n");

}
```

```
process 0 says count is 1
process 2 says count is 2
process 1 says count is 3
process 0 says count is 4
process 2 says count is 5
process 1 says count is 6
process 2 says count is 7
process 0 says count is 8
process 1 says count is 9
counter over
```

3.7.1 The Fairness of Locks

A locking mechanism is called "Fair" if a process eventually enters its critical region after trying to obtain the lock. The best example might be a queuing semaphore, which maintains the order of processes trying to obtain the lock in a FIFO fashion. However, **m_lock** does not block and queue a process. On the Sequent, a process simply spins in its processor and repetitively attempts to obtain the lock. This means that if process 1, 2, and 3 each try to obtain a busy lock, there is no way of knowing which process will be the first to enter its critical region.

Example 11 performs the same counting function as the previous example, however, a timing routine has been added to test when a process attempts to obtain a lock and when it actually obtains the lock. The header file *sys/time.h* must be included to use this routine. The structures *timeval* and *timezone* are used with the **gettimeofday** routine to get the desired information. The **gettimeofday** routine returns a timing counter which is incremented every 10 milliseconds. In the function "counts", each process will get its PID and enter a loop to increment *count*. In the loop, each process will call **gettimeofday** before attempting the call to **m_lock**. After a process has enter its critical region, it again calls the **gettimeofday** routine for the time. The process then enters a second loop which performs no service except to waste time. This is done to ensure that more than one process is waiting to enter the critical region. The counter is then incremented and both the counter and times are printed. The process then releases the lock for another process to enter its critical region.

The output shows the counter is incremented in correct order. It also shows the time that each process attempted to enter its critical region and the time it entered the critical region. Notice that process 0 on count 8, attempted to enter the critical region before process 1 at count 7, yet process 1 entered the region first. Both had issued a call to **m_lock**. It is just by chance that process 1 obtained the lock first. The reason for this is both timing and hardware. It may be that process 1 noticed the lock was released before process 0. However, if two processes attempt to obtain the same lock simultaneously, the process with the lowest priority will wait. If both processes have the same priority, the process with the lowest processor number will wait. Starvation results when one process never obtains the lock. On the Balance 8000, starvation will not occur because the priority of a waiting process is increased over time. This is called "Weak Fairness". The command "man 2 gettimeofday" can be used to read about this function further.

```
                        /***************/
                        /*  Example 11  */
                        /***************/
#include <sys/time.h>
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;


/*    This program illustrates the use of m_lock.  The program counts  */
/*    by increasing the value of a variable in parallel.  The program  */
/*    maintains mutual exclusion with a call to m_lock.  The system's  */
/*    calls to gettimeofday are to show when the process attempted     */
/*    to access the lock and when the process obtained the lock.       */
/*    Although not conclusive, it would appear that the m_lock call is */
/*    not fair.                                                        */

counts ()
{
   struct timeval t, r;         /*  Timing Variables  */
   struct timezone t1, r1;

   int me, i, j, k;
   k = 0;
   me = m_get_myid ();        /*  Get my PID  */

   for (i=0; i < NPROCS; i++)  {     /*  Increment Count NPROCS times  */

      gettimeofday (&t, &t1);        /*  Time before Access  */
      m_lock ();
      gettimeofday (&r, &r1);        /*  Time after Access   */


      for (j = 0; j < 10000; j++)      /*  Waste some time  */
         k = j + k;

      count += 1;

      printf ("process %d : count is %d :", me, count);
      printf (" Tried : %d %d Obtained : %d %d\n", t.tv_sec,
               t.tv_usec, r.tv_sec, r.tv_usec);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{
   count = 0;

   m_set_procs (NPROCS);     /*  Create NPROCS processes  */
   m_fork (counts);
   m_kill_procs ();          /*  Terminate all processes except Parent  */

   printf ("counter over\n");

}
```

```
process 0 : count is 1 : Tried : 549147970 510000 Obtained : 549147970 510000
process 1 : count is 2 : Tried : 549147970 510000 Obtained : 549147970 670000
process 2 : count is 3 : Tried : 549147970 510000 Obtained : 549147970 820000
process 0 : count is 4 : Tried : 549147970 670000 Obtained : 549147970 970000
process 1 : count is 5 : Tried : 549147970 820000 Obtained : 549147971 110000
process 2 : count is 6 : Tried : 549147970 970000 Obtained : 549147971 240000
process 1 : count is 7 : Tried : 549147971 240000 Obtained : 549147971 380000
process 0 : count is 8 : Tried : 549147971 110000 Obtained : 549147971 520000
process 2 : count is 9 : Tried : 549147971 380000 Obtained : 549147971 650000
counter over
```

3.7.2 Multiple Locks

How many times can **m_lock** be called within a process? It was thought that it should only be called once for one critical region. However, this is entirely up to the programmer. **m_lock** sets up one lock to be used by the programmer. The programmer is free to use the lock as he wishes. The following program executes a new "counts" function. The new function increments two counters. Each process will call **m_lock**, increment and print a counter value, and call **m_unlock** to release the lock. The process then attempts the same procedure for the other lock. Each process increments each lock three times.

The output shows that each lock was incremented to the value of nine and that the increments were all in order from 1 to 9. However, notice that near the end of the output, process 0 increments count2 before process 1 has incremented count1. In other words, you can not predict which counter will be incremented next. This is because the same lock is used for both critical regions. This is a rather inefficient method since the counters are independent of each other.

```
                                /****************/
                                /*  Example 12  */
                                /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count1, count2;

/*  This program illustrates the use of m_lock.  In this program  */
/*   m_lock is used twice.  M_lock uses just one lock.  However,  */
/*   You can have multiple occurrences of m_lock in a routine.    */
/*   In this program two shared counters are incremented by all   */
/*   copies of "counts", however, only one lock is used for both  */
/*   count variables.                                             */

counts ()
{
   int me, i;
   me = m_get_myid ();

   for (i=0; i < NPROCS; i++)  {
      m_lock ();
      count1 += 1;
      printf ("process %d says count1 is %d\n", me, count1);
      fflush (stdout);
      m_unlock ();

      m_lock ();
      count2 += 1;
      printf ("process %d says count2 is %d\n", me, count2);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{

   count1 = 0;
   count2 = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();

   printf ("counter over\n");

}


process 0 says count1 is 1
process 2 says count1 is 2
process 1 says count1 is 3
process 0 says count2 is 1
process 2 says count2 is 2
process 1 says count2 is 3
process 0 says count1 is 4
process 2 says count1 is 5
process 1 says count1 is 6
process 2 says count2 is 4
process 0 says count2 is 5
process 1 says count2 is 6
process 0 says count1 is 7
process 2 says count1 is 8
process 0 says count2 is 7
process 1 says count1 is 9
process 2 says count2 is 8
process 1 says count2 is 9
counter over
```

3.7.3 Omission and Commission Errors

The last example showed how a programmer can use the calls to **m_lock** and **m_unlock** for his own purposes. This can become a very dangerous situation. What would happen if a process tried to nest calls to **m_lock**. In other words, tried to obtain a lock it already had. Would the system allow the process to continue? No! The system does not care which process has obtained a lock and who will get it next. It only ensures that only one process at a time can use the lock. The following example is the original "counts" program minus a call to **m_unlock** after leaving the critical region. This is an inherent problem with locks and semaphores. The program deadlocks right away. After the first process obtains the lock, no other process can continue. The process which first obtained the lock also waits when it calls **m_lock** on the second iteration. The output shows that process 0 obtained the lock, incremented and printed the counter. However, now every process is waiting for the lock because process 0 forgot to release it. Hit CTRL-C to kill a program which has deadlocked.

```
                        /****************/
                        /*  Example 13  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*  This program attempts to increment a counter within a critical  */
/*  region.  However, the program forgets to unlock the critical    */
/*  region and deadlocks the program.                               */

counts ()
{
   int me, i;
   me = m_get_myid ();

   for (i=0; i < NPROCS; i++)  {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);

   }
}

main ()
{

   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();

   printf ("counter over\n");

}
```

```
process 0 says count is 1
```

3.8 M_Sync

The **m_sync** routine causes a process to spin until all processes which were **m_forked** have reached the same point and have called **m_sync**. The routine is used to synchronize all processes which were created by a call to **m_fork**. The best way to explain **m_sync** is by example. Example 14 again increments a shared counter. Each process which executes "counts", will increment the counter N times, where N is the number of processes created multiplied by the process's PID + 1 (PID obtained by **m_get_myid**). Each process will execute an outer loop three times (three processes are created). Each process will then enter another loop and will increment and print the shared counter. This next loop is executed PID + 1 times. This varies with each process and better demonstrates **m_sync**.

The output shows the three iterations imposed by the outer loop. Within each iteration, the count is incremented six times (once by process 0, twice by process 1, and thrice by process 2). Notice that all of iteration 0 is performed before iteration 1 begins, even though process 0 is finished and ready to continue. Process 0 waits for the other two processes to finish and synchronize before continuing to the next iteration.

```
                        /****************/
                        /*  Example 14  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*  This program creates NPROCS processes to execute the procedure  */
/*  "counts".  Each process will increment a shared counter N times */
/*  where N is NPROCS * (PID + 1).  m_sync is used to synchronize   */
/*  all processes before each outer loop iteration.                 */

counts ()
{
   int me, i, j, iterations;

   me = m_get_myid ();
   iterations = me + 1;

   for (i=0; i < NPROCS; i++)  {
      for (j=0; j < iterations; j++)  {
         m_lock ();
         count += 1;
         printf ("iteration %d process %d says count is %d\n", i, me, count);
         fflush (stdout);
         m_unlock ();
      }
      m_sync ();           /*  Synchronize all processes  */
   }
}

main ()
{

   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();

   printf ("counter over\n");

}


iteration 0 process 0 says count is 1
iteration 0 process 1 says count is 2
iteration 0 process 2 says count is 3
iteration 0 process 1 says count is 4
iteration 0 process 2 says count is 5
iteration 0 process 2 says count is 6
iteration 1 process 1 says count is 7
iteration 1 process 2 says count is 8
iteration 1 process 0 says count is 9
iteration 1 process 1 says count is 10
iteration 1 process 2 says count is 11
iteration 1 process 2 says count is 12
iteration 2 process 2 says count is 13
iteration 2 process 0 says count is 14
iteration 2 process 1 says count is 15
iteration 2 process 2 says count is 16
iteration 2 process 1 says count is 17
iteration 2 process 2 says count is 18
counter over
```

3.9 M_Park_Procs and M_Rele_Procs

A process created by **m_fork** will spin waiting for more work after it has executed the subprogram named by the **m_fork**. If no more work is to be done, then the process can be terminated by a call to **m_kill_procs**. What if there is more work for each process to do, but the parent process must do some initial work sequentially? You could leave the processes spinning while the parent executes. This is very wasteful of processor time. You could terminate the processes and then recreate them when needed. This solution is wasteful of processing time due to the amount of overhead needed to recreate the processes. Each **m_fork** requires a certain amount of time to copy the named subprogram to different processors for execution.

The **m_park_procs** routine suspends execution of each process which was previously created by a call to **m_fork**. These processes still exist at the different processors, but they are no longer spinning. They have been blocked and are not active. The **m_rele_procs** routine resumes the execution of processes which have been blocked by a previous call to **m_park_procs**. These two routines are very useful and allow a programmer to reuse processes without wasting processor time or processing time.

Example 15 executes the original "counts" procedure. The main process creates three processes to execute "counts" in parallel. After the processes are finished, the parent process calls **m_park_procs** to block the processes from execution. The parent prints the message "Take a Rest". It then unblocks the processes with a call to **m_rele_procs** and calls **m_fork** to create three processes to again execute "counts". **m_fork** is smart and will not create new processes if they already exist. The output shows that the counter was incremented nine times. The processes took a rest and then the counter was incremented nine times again. Notice that the counter was incremented to the value of nine on the first **m_fork** and then was incremented to the value of 18 on the second **m_fork**. This shows that the counter does not lose its value between **m_forks** and suggests that the processes only receive a pointer to shared variables. If the processes were terminated between the two calls to **m_fork**, the results would be the same. If you need a shared variable to be reset between calls to **m_fork**, you must do it yourself.

```
                        /****************/
                        /*  Example 15  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*    This program illustrates the use of the m_park_procs call.   */
/*    Three processes are created which count by incrementing a    */
/*    shared counter.  After NPROCS iterations of counting, the    */
/*    parent process parks each process and prints a message.      */
/*    After the message is printed the parent releases the         */
/*    processes and continues counting.                            */

counts ()
{
   int me, i;
   me = m_get_myid ();          /*  Who am I  */

   for (i=0; i < NPROCS; i++)   {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{
   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_park_procs ();               /*  Park all children  */

   printf ("\n\n Take a Rest \n\n");
   fflush (stdout);

   m_rele_procs ();               /*  Release the children  */
   m_fork (counts);               /*  Put children to work  */
   m_kill_procs ();

   printf ("counter over\n");
}


process 0 says count is 1
process 1 says count is 2
process 2 says count is 3
process 0 says count is 4
process 1 says count is 5
process 2 says count is 6
process 1 says count is 7
process 0 says count is 8
process 2 says count is 9


 Take a Rest

process 0 says count is 10
process 2 says count is 11
process 1 says count is 12
process 0 says count is 13
process 2 says count is 14
process 1 says count is 15
process 0 says count is 16
process 2 says count is 17
process 1 says count is 18
counter over
```

3.9.1 The Inflexible M_Park_Procs

There is one major problem with **m_park_procs**. If you want to execute "counts" three times and then execute it later only two times, you must terminate the original processes and create two new ones. This problem goes back to the call to **m_set_procs**. **m_set_procs** not only tells how many processes can execute in parallel, but also tells **m_fork** how many processes to create. DYNIX will not allow you to reset the number of processes you need for the next **m_fork** without first calling **m_kill_procs** to terminate the current processes.

Example 16a illustrates this point. The main process creates three children to execute "counts". After they are finished, the main process terminates them with a call to **m_kill_procs**. The main process resets the counter, prints a message, and then resets the number of processes needed to two by calling **m_set_procs**. These two processes are then created by a call to **m_fork** and execute "counts". The output verifies the program. If the number of processes had not been reset to two using **m_set_procs**, the **m_fork** would have created three (due to the previous value of **m_set_procs**). If the original processes had not been terminated by a call to **m_kill_procs** before resetting the number needed to two, DYNIX would have ignored the call to **m_set_procs**. This is illustrated by example 16b. In this example, the original processes were not terminated before resetting **m_set_procs**. Notice that three processes were used to execute "count" after the message "Take a Rest".

```
                        /*****************/
                        /*  Example 16a  */
                        /*****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define TWO 2
#define NPROCS 3

shared int count;

/*    This program illustrates the use of the m_park_procs call.   */
/*    Three processes are created which count by incrementing a    */
/*    shared counter.  After NPROCS iterations of counting, the    */
/*    parent process kills each process and prints a message.      */
/*    After, the message is printed the parent creates two new     */
/*    processes and continues counting.                            */

counts ()
{
   int me, i;
   me = m_get_myid ();           /*  Who am I  */

   for (i=0; i < NPROCS; i++)   {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{
   count = 0

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();              /*  Kill all children  */

   printf ("\n\n Take a Rest \n\n");
   fflush (stdout);
   count = 0;                        /*  reset counter  */

   m_set_procs (TWO);              /*  Create two processes  */
   m_fork (counts);               /*  Put children to work  */
   m_kill_procs ();

   printf ("counter over\n");
}
```

```
process 0 says count is 1
process 2 says count is 2
process 1 says count is 3
process 0 says count is 4
process 2 says count is 5
process 1 says count is 6
process 0 says count is 7
process 2 says count is 8
process 1 says count is 9


 Take a Rest

process 0 says count is 1
process 1 says count is 2
process 0 says count is 3
process 1 says count is 4
process 0 says count is 5
process 1 says count is 6
counter over
```

```
                        /*****************/
                        /*  Example 16b  */
                        /*****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define TWO 2
#define NPROCS 3

shared int count,

/*    This program illustrates the use of the m_park_procs call.   */
/*    Three processes are created which count by incramenting a    */
/*    shared counter.  After NPROCS iterations of counting, the    */
/*    parent process  prints a message.                            */
/*    After, the message is printed the parent resets the number   */
/*    processes to two.  Notice that three processes are created.  */
/*    The system ignored the m_set_procs routine because the       */
/*    parent process did not kill all the children processes       */
/*    first.                                                       */

counts ()
{
   int me, i;
   me = m_get_myid ();          /*  Who am I  */

   for (i=0; i < NPROCS; i++)   {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

main ()
{
   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);

   printf ("\n\n Take a Rest \n\n");
   fflush (stdout);
   count = 0;                        /*  reset counter  */

   m_set_procs (TWO);               /*  Create two processes  */
   m_fork (counts);                /*  Put children to work  */
   m_kill_procs ();

   printf ("counter over\n");
}
```

```
process 0 says count is 1
process 2 says count is 2
process 0 says count is 3
process 1 says count is 4
process 2 says count is 5
process 0 says count is 6
process 1 says count is 7
process 2 says count is 8
process 1 says count is 9


 Take a Rest

process 2 says count is 1
process 1 says count is 2
process 0 says count is 3
process 2 says count is 4
process 1 says count is 5
process 0 says count is 6
process 2 says count is 7
process 1 says count is 8
process 0 says count is 9
counter over
```

3.9.2 The Efficiency of M_Park_Procs and M_Rele_Procs

When **m_fork** is called, what is copied to the new processors? Does **m_fork** only copy the subprogram named in the parameter? Or does **m_fork** give each new processor an entire copy (data segment, instruction segment, and system data segment) of the process which calls it (like **fork**)? This is an important question. If you execute the function "counts" three times and then wish to execute the function "sayhi" three times, should you terminate the original processes before **m_forking** "sayhi" or should you just block them and release them when needed? If you do not terminate them, will the **m_fork** expect the function "sayhi" to be on each of the processors or will it have to copy the function to them? If the **m_fork** must copy the function "sayhi" to the processors, do you save any time by using **m_park_procs** instead of **m_kill_procs**?

The answer is that **m_fork** copies the entire environment of the calling process to the new processors. Therefore, on subsequent calls to **m_fork**, no new data or instructions need to be copied. This means that time is saved by using **m_park_procs** and **m_rele_procs** instead of **m_kill_procs**.

Three examples are used to demonstrate this point. Again, the **gettimeofday** routine is used to test times. The first two examples have a main process and two functions, "counts" and "sayhi". Example 17a executes three copies of "counts" using **m_fork** and then terminates the processes with **m_kill_procs**. The main process calls **gettimeofday** to find the time before it executes "sayhi". It then **m_forks** "sayhi" and calls **gettimeofday** again. The output shows that it took approximately .14 seconds to create and execute the three copies of "sayhi".

Example 17b is exactly like the first except that after "counts" is executed, the processes are blocked with a call to **m_park_procs**. **gettimeofday** is called and then the processes are released to execute "sayhi". After each process is finished executing "sayhi", **gettimeofday** is called again. In this case the output shows that the time to release the processes and execute "sayhi" was approximately .02 seconds. This shows that blocking the processes is much quicker. This program was run several times to ensure consistent results.

However, were both subprograms ("counts" and "sayhi") copied in the initial **m_fork**? Maybe the creation of the processes using **m_fork** has more overhead than copying the additional function ("sayhi"). Example 17c has a main process and one function, "sayhi". This program creates and executes "sayhi" three times. It then blocks the processes, gets the time, releases the processes, executes the same function "sayhi", and again gets the time. In this program nothing needs to be copied on the second execution of "sayhi". The output shows the time to still be approximately .02 seconds. This shows that the entire parent process is copied to each new processor on an initial **m_fork**.

```
                        /****************/
                        /*  Example 17a */
                        /****************/
#include <sys/time.h>
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*  This program is used to show the time it takes to          */
/*  create processes.  Two routines are used in the program.   */
/*  First, a number of proceses are created to run the first   */
/*  routine and then they are killed.  Then, the time it       */
/*  takes to create new processes to do the other simple       */
/*  routine is recorded.                                       */

counts ()
{
   int me, i;
   me = m_get_myid ();           /*  Who am I  */

   for (i=0; i < NPROCS; i++)   {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

say_hi ()
{
   int me;
   me = m_get_myid ();         /*  who am I  */

   m_lock ();
   printf ("Process %d says Hello\n", me);
   fflush (stdout);
   m_unlock ();
}

main ()
{
   struct timeval t, r;
   struct timezone t1, r1;

   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();               /*  Kill all children  */

   gettimeofday (&t, &t1);
   m_fork (say_hi);              /*  Put children to work  */
   gettimeofday (&r, &r1);
   m_kill_procs ();

   printf ("Time before say_hi is  %d %d\n", t.tv_sec, t.tv_usec);
   printf ("Time after say_hi is  %d %d\n", r.tv_sec, r.tv_usec);

}
```

```
process 0 says count is 1
process 2 says count is 2
process 1 says count is 3
process 0 says count is 4
process 1 says count is 5
process 2 says count is 6
process 0 says count is 7
process 2 says count is 8
process 1 says count is 9
Process 0 says Hello
Process 2 says Hello
Process 1 says Hello
Time before say_hi is: 549149061 530000
Time after say_hi is:  549149061 670000
```

```
                        /****************/
                        /*  Example 17b */
                        /****************/
#include <sys/time.h>
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*  This program is used to record the time it takes to execute a     */
/*  routine on existing processes.  First, the processes are created  */
/*  to execute a separate counting routine.  Then they are parked     */
/*  and then released to execute a simple routine which               */
/*  prints a message.  The "gettimeofday" system call is used to      */
/*  record the times.                                                 */


counts ()
{
   int me, i;
   me = m_get_myid ();           /*  Who am I  */

   for (i=0; i < NPROCS; i++)    {
      m_lock ();
      count += 1;
      printf ("process %d says count is %d\n", me, count);
      fflush (stdout);
      m_unlock ();
   }
}

say_hi ()
{
   int me;
   me = m_get_myid ();         /*  who am I  */

   m_lock ();
   printf ("Process %d says Hello\n", me);
   fflush (stdout);
   m_unlock ();
}

main ()
{
   struct timeval t, r;
   struct timezone t1, r1;

   count = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_park_procs ();               /*  Park all children  */

   gettimeofday (&t, &t1);
   m_rele_procs ();              /*  Release all Children  */
   m_fork (say_hi);              /*  Put children to work  */
   gettimeofday (&r, &r1);
   m_kill_procs ();

   printf ("Time before say_hi is  %d %d\n", t.tv_sec, t.tv_usec);
   printf ("Time after say_hi is   %d %d\n", r.tv_sec, r.tv_usec);

}
```

```
process 0 says count is 1
process 2 says count is 2
process 1 says count is 3
process 2 says count is 4
process 0 says count is 5
process 1 says count is 6
process 2 says count is 7
process 0 says count is 8
process 1 says count is 9
Process 1 says Hello
Process 0 says Hello
Process 2 says Hello
Time before say_hi is: 549149378 500000
Time after say_hi is:  549149378 520000
```

```
                        /****************/
                        /*  Example 17c */
                        /****************/
#include <sys/time.h>
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count;

/*  This program is used to record the time it takes to execute a    */
/*  routine on existing processes.  First, the processes are created */
/*  to execute the routine "say_hi".  Then, they are parked,         */
/*  released, and then they re-execute "say_hi".  I am interested in */
/*  the time to re_execute the processes.                            */


say_hi ()
{
   int me;
   me = m_get_myid ();         /*  who am I  */

   m_lock ();
   printf ("Process %d says Hello\n", me);
   fflush (stdout);
   m_unlock ();
}

main ()
{
   struct timeval t, r;
   struct timezone t1, r1;


   m_set_procs (NPROCS);
   m_fork (say_hi);
   m_park_procs ();                /*  Park all children  */

   gettimeofday (&t, &t1);
   m_rele_procs ();               /*  Release all Children  */
   m_fork (say_hi);               /*  Put children to work  */
   gettimeofday (&r, &r1);
   m_kill_procs ();

   printf ("Time before say_hi is: %d %d\n", t.tv_sec, t.tv_usec);
   printf ("Time after say_hi is:  %d %d\n", r.tv_sec, r.tv_usec);

}
```

```
Process 0 says Hello
Process 1 says Hello
Process 2 says Hello
Process 0 says Hello
Process 1 says Hello
Process 2 says Hello
Time before say_hi is: 549149628 780000
Time after say_hi is:  549149628 790000
```

3.10 S_Lock and S_Unlock

The **s_lock** and **s_unlock** routines are very similar to the **m_lock** and **m_unlock** routines except they give the programmer the flexibility of using more than one lock. A lock is created by declaring a variable to be of type **slock_t**. The lock must also be declared as **shared**. The **s_init_lock** routine initializes a memory-based lock. Both of these actions were done for the programmer when using **m_lock**. After a lock is created and initialized, a programmer may use the lock to ensure mutual exclusion using the **s_lock** and **s_unlock** routines. This is done exactly as before using **m_lock** and **m_unlock**. However, now a process can create multiple locks and use them in different contexts.

The following program increments two different shared counters. Three processes are created and each will increment the two counters three times. The main process begins by initializing two locks *lock1* and *lock2*. The main process then calls **m_fork** to create the processes and execute "counts". Each process gets its PID and then enters a loop to increment *count1*. *lock1* is used to ensure mutual exclusion while incrementing *count1*. After a process is finished with the first loop, it enters a second loop and begins to increment *count2*. *lock2* is used to ensure mutual exclusion while incrementing *count2*. Notice that the addresses of the locks are used as parameters to the routines **s_init_lock**, **s_lock**, and **s_unlock**. The DYNIX programmer's manual says to declare a lock to be a pointer to type **slock_t** and to pass these pointers. This does not work. However, if you declare the variables to be of type **slock_t** and pass their addresses, everything works just fine.

The output shows that each counter was incremented nine times. Notice that the value of 7 is not printed for *count1*. This is because process 2 had entered the second loop to increment *count2* and had overwritten the output buffer before *count1's* value of 7 could be printed. The counters are independent between the two loops and were incremented correctly, but the output buffer is shared between the two loops. Since *lock1* has no effect on *lock2*, there is no mutual exclusion between the two loops and output can be lost. Both counters could be placed in one loop using both locks.

```
                        /****************/
                        /*  Example 18  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count1, count2;
shared slock_t lock1, lock2;          /*  Declare the Locks  */

/*  This program illustrates the use of locking variables to  */
/*  ensure mutual exclusion.  Two locks are created by the    */
/*  declaration of type "slock_t" and the initialization call */
/*  "s_init_lock".  This program increments two  counters in  */
/*  parallel.  Each of three processes will concurrently      */
/*  increment the counter and print its value.  Two critical  */
/*  regions are implemented with the two locks to ensure that */
/*  the incrementing of a counter and printing its value      */
/*  appear as atomic instructions.                            */

counts ()
{

   int me, i;
   me = m_get_myid ();             /*  Who am I  */

   for (i=0; i < NPROCS; i++)  {
      s_lock (&lock1);              /*  Lock the critical region  */
      count1 += 1;
      printf ("process %d says count1 is %d\n", me, count1);
      fflush (stdout);
      s_unlock (&lock1);             /*  Unlock the critical region  */
   }

   for (i=0; i < NPROCS; i++)  {
      s_lock (&lock2);               /*  Lock the critical region  */
      count2 += 1;
      printf ("process %d says count2 is %d\n", me, count2);
      fflush (stdout);
      s_unlock (&lock2);             /*  Unlock the critical region  */
   }
}

main ()
{

   s_init_lock (&lock1,&lock2);      /*  initialize the locks  */
   count1 = 0;
   count2 = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
   m_kill_procs ();

   printf ("counter over\n");

}
```

```
process 2 says count1 is 1
process 1 says count1 is 2
process 0 says count1 is 3
process 2 says count1 is 4
process 1 says count1 is 5
process 2 says count1 is 6
process 2 says count2 is 1
process 1 says count1 is 8
process 2 says count2 is 2
process 0 says count1 is 9
process 1 says count2 is 3
process 2 says count2 is 4
process 0 says count2 is 5
process 0 says count2 is 6
process 1 says count2 is 7
process 0 says count2 is 8
process 1 says count2 is 9
counter over
```

3.11 S_Init_Barrier and S_Wait_Barrier

The **s_init_barrier** routine initializes a barrier as a rendezvous point for N processes, where N is passed as a parameter to **s_init_barrier**. The *s_wait_barrier* routine delays each calling process in a busy-wait spin until exactly N processes have called **s_wait_barrier**. In C, a barrier is declared as a shared data structure of type **sbarrier_t**. The function of the **s_wait_barrier** routine is exactly like the **m_sync** routine with the added flexibility of specifying the number of processes to synchronize. It also allows the creation of multiple barriers.

The following program again increments two shared counters, *count1* and *count2*. The main process will create and initialize two locks and two barriers; one for each counter. The main process creates three processes to execute "counts". Each process gets its PID and enters an outer loop. The outer loop is designed to demonstrate the use of the barriers. Again, I have two inner loops to increment each counter separately. *lock1* and *lock2* provide mutual exclusion for *count1* and *count2*, respectively. The main process has initialized each barrier to wait for three processes (the number created). Each process calls **s_wait_barrier** between each inner loop. All processes synchronize on *barrier1* after incrementing *count1*. All processes synchronize on *barrier2* after incrementing *count2*.

The output shows that each counter was incremented 27 times, nine times per outer loop. Notice that each counter is fully incremented and printed before the next loop has begun. This time no output is lost.

```
                        /****************/
                        /*  Example 19  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count1, count2;
shared sbarrier_t barrier1, barrier2;      /*  Declare the Barriers  */
shared slock_t lock1, lock2;               /*  Declare the Locks  */

/*  This program illustrates the use of locking variables to  */
/*  ensure mutual exclusion.  Two locks are created by the    */
/*  declaration of type "slock_t" and the initialization call */
/*  "s_init_lock".  This program increments two  counters in  */
/*  parallel.  Each of three processes will concurrently      */
/*  increment the counter and print its value.  Two critical  */
/*  regions are implemented with the two locks to ensure that */
/*  the incrementing of a counter and printing its value      */
/*  appear as atomic instructions.                            */
/*  Two barriers are declared by the type "sbarrier_t" and    */
/*  initialized by the call to "s_init_barrier".  The two     */
/*  barriers are used to synchronize all the processes after  */
/*  incrementing each counter.  The call to s_wait_barrier    */
/*  will block the calling process until  NPROCS processes    */
/*  have made the call.                                       */

counts ()
{

   int me, i, j;
   me = m_get_myid ();             /*  Who am I  */

   for (j=0; j < NPROCS; j++)  {
      for (i=0; i < NPROCS; i++)  {
         s_lock (&lock1);                /*  Lock the critical region  */
         count1 += 1;
         printf ("process %d says count1 is %d\n", me, count1);
         fflush (stdout);
         s_unlock (&lock1);              /*  Unlock the critical region  */
      }
      s_wait_barrier (&barrier1);        /*  All processes synchronize  */

      for (i=0; i < NPROCS; i++)  {
         s_lock (&lock2);                /*  Lock the critical region  */
         count2 += 1;
         printf ("process %d says count2 is %d\n", me, count2);
         fflush (stdout);
         s_unlock (&lock2);              /*  Unlock the critical region  */
      }
      s_wait_barrier (&barrier2);        /*  All processes synchronize  */
   }
}

main ()
{
   s_init_barrier (&barrier1, NPROCS);  /*  initialize the barriers  */
   s_init_barrier (&barrier2, NPROCS);
   s_init_lock (&lock1);                /*  initialize the locks  */
   s_init_lock (&lock2);

   count1 = 0;
   count2 = 0;

   m_set_procs (NPROCS);
   m_fork (counts);
```

```
    m_kill_procs ();

    printf ("counter over\n");

}
```

```
process 0 says count1 is 1
process 2 says count1 is 2
process 1 says count1 is 3
process 0 says count1 is 4
process 2 says count1 is 5
process 1 says count1 is 6
process 0 says count1 is 7
process 2 says count1 is 8
process 1 says count1 is 9
process 0 says count2 is 1
process 2 says count2 is 2
process 1 says count2 is 3
process 0 says count2 is 4
process 1 says count2 is 5
process 2 says count2 is 6
process 1 says count2 is 7
process 0 says count2 is 8
process 2 says count2 is 9
process 1 says count1 is 10
process 2 says count1 is 11
process 1 says count1 is 12
process 0 says count1 is 13
process 2 says count1 is 14
process 1 says count1 is 15
process 0 says count1 is 16
process 2 says count1 is 17
process 0 says count1 is 18
process 1 says count2 is 10
process 0 says count2 is 11
process 2 says count2 is 12
process 1 says count2 is 13
process 0 says count2 is 14
process 2 says count2 is 15
process 1 says count2 is 16
process 0 says count2 is 17
process 2 says count2 is 18
process 1 says count1 is 19
process 2 says count1 is 20
process 0 says count1 is 21
process 1 says count1 is 22
process 0 says count1 is 23
process 2 says count1 is 24
process 1 says count1 is 25
process 0 says count1 is 26
process 2 says count1 is 27
process 0 says count2 is 19
process 2 says count2 is 20
process 1 says count2 is 21
process 0 says count2 is 22
process 2 says count2 is 23
process 1 says count2 is 24
process 0 says count2 is 25
process 2 says count2 is 26
process 1 says count2 is 27
counter over
```

3.12 M_Single and M_Multi

How can a programmer print out a message within a **m_forked** subprogram? Not every process should print the message. How can you perform any type of I/O only once (such as reading a counter value)? The subprogram could be written so that only a specific process (based on PID) performs the read while the others wait. The **m_single** and **m_multi** routines suspend the execution of all child processes while the parent (process 0) performs some sequential task (I/O). The **m_single** places the children in a spin while the parent continues execution. The parent calls **m_multi** to resume the execution of the children. The children do not execute the code between the call to **m_single** and the call to **m_multi**.

Example 20 performs the same function as the previous program. It increments two shared counters. Again two locks and two barriers are used to ensure mutual exclusion and synchronization. However, this program prints a message between each iteration of the outer loop. Only one process should print the message (the parent). The print command is encapsulated within an **m_single** / **m_multi** block. The output shows that the iteration number was printed only once (by the parent) after each iteration of the outer loop.

```
                        /****************/
                        /*  Example 20  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3

shared int count1, count2;
shared sbarrier_t barrier1, barrier2;      /*  Declare the Barriers  */
shared slock_t lock1, lock2;               /*  Declare the Locks  */

/*  This program illustrates the use of locking variables to  */
/*  ensure mutual exclusion.  Two locks are created by the    */
/*  declaration of type "slock_t" and the initialization call */
/*  "s_init_lock".  This program increments two counters in   */
/*  parallel.  Each of three processes will concurrently      */
/*  increment the counter and print its value.  Two critical  */
/*  regions are implemented with the two locks to ensure that */
/*  the incrementing of a counter and printing its value      */
/*  appear as atomic instructions.                            */
/*  Two barriers are declared by the type "sbarrier_t" and    */
/*  initialized by the call to "s_init_barrier".  The two     */
/*  barriers are used to synchronize all the processes after  */
/*  incrementing each counter.  The call to s_wait_barrier    */
/*  will block the calling process until  NPROCS processes    */
/*  have made the call.                                       */
/*  The m_single and m_multi system calls are used to allow   */
/*  the parent process to print a message.  These calls       */
/*  suspend all processes except the parent and only the      */
/*  parent is allowed into this critical region.              */

counts ()
{

   int me, i, j;
   me = m_get_myid ();             /*  Who am I  */

   for (j=0; j < NPROCS; j++)  {
      for (i=0; i < NPROCS; i++)  {
         s_lock (&lock1);                /*  Lock the critical region  */
         count1 += 1;
         printf ("process %d says count1 is %d\n", me, count1);
         fflush (stdout);
         s_unlock (&lock1);              /*  Unlock the critical region  */
      }
      s_wait_barrier (&barrier1);        /*  All processes synchronize  */

      for (i=0; i < NPROCS; i++)  {
         s_lock (&lock2);                /*  Lock the critical region  */
         count2 += 1;
         printf ("process %d says count2 is %d\n", me, count2);
         fflush (stdout);
         s_unlock (&lock2);              /*  Unlock the critical region  */
      }
      s_wait_barrier (&barrier2);        /*  All processes synchronize  */

      /*  Parent process prints a message  */

      m_single ();
      printf ("\nIteration %d Completed\n\n", j+1);
      fflush (stdout);
      m_multi ();
   }
}

main ()
```

```
{
    s_init_barrier (&barrier1, NPROCS);   /*  initialize the barriers  */
    s_init_barrier (&barrier2, NPROCS);
    s_init_lock (&lock1);                 /*  initialize the locks  */
    s_init_lock (&lock2);

    count1 = 0;
    count2 = 0;

    m_set_procs (NPROCS);
    m_fork (counts);
    m_kill_procs ();

    printf ("counter over\n");

}
```

```
process 0 says count1 is 1
process 1 says count1 is 2
process 2 says count1 is 3
process 0 says count1 is 4
process 1 says count1 is 5
process 2 says count1 is 6
process 0 says count1 is 7
process 1 says count1 is 8
process 2 says count1 is 9
process 2 says count2 is 1
process 0 says count2 is 2
process 1 says count2 is 3
process 2 says count2 is 4
process 0 says count2 is 5
process 1 says count2 is 6
process 2 says count2 is 7
process 0 says count2 is 8
process 1 says count2 is 9

Iteration 1 Completed

process 0 says count1 is 10
process 1 says count1 is 11
process 2 says count1 is 12
process 1 says count1 is 13
process 0 says count1 is 14
process 2 says count1 is 15
process 1 says count1 is 16
process 0 says count1 is 17
process 2 says count1 is 18
process 1 says count2 is 10
process 2 says count2 is 11
process 0 says count2 is 12
process 1 says count2 is 13
process 2 says count2 is 14
process 0 says count2 is 15
process 1 says count2 is 16
process 2 says count2 is 17
process 0 says count2 is 18

Iteration 2 Completed

process 0 says count1 is 19
process 2 says count1 is 20
process 1 says count1 is 21
process 0 says count1 is 22
process 2 says count1 is 23
process 1 says count1 is 24
process 0 says count1 is 25
process 2 says count1 is 26
process 1 says count1 is 27
process 2 says count2 is 19
process 0 says count2 is 20
process 1 says count2 is 21
process 2 says count2 is 22
process 0 says count2 is 23
process 1 says count2 is 24
process 2 says count2 is 25
process 0 says count2 is 26
process 1 says count2 is 27

Iteration 3 Completed

counter over
```

3.13 M_Next

The **m_next** routine increments a global counter. The counter is initialized to zero each time the **m_fork**, **m_single**, or **m_sync** routines are called. You may obtain the value of the counter simply by calling **m_next** which returns its integer value. Each time **m_next** is called, it returns its current value and then increments the counter. These two steps are accomplished atomically to ensure that no two processes will see the same value of the counter.

Example 21 illustrates the use of the **m_next** routine. In this example, "counts" is called to increment a shared counter nine times. The main routine creates three processes to execute "counts" in parallel. In this particular example, the main routine does not care how many times a process increments the counter, just as long as the counter is incremented nine times. Each process does not know how many times it should increment the counter. It only knows the number of times the counter should be incremented, nine. By calling **m_next** before incrementing the counter, each process can see if the counter has been incremented the correct number of times. If the counter has not been incremented nine times, increment the counter, otherwise return. The output shows that the counter was incremented nine times. However, each process did not increment the counter three times as in previous examples. Process 1 incremented the counter four times. The **m_next** routine is most useful for dynamic applications. For example, an application where each process performs the same task on a set of data, but the amount of data is not known until run time.

```
                        /****************/
                        /*  Example 21  */
                        /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3
#define N 9
shared int count;

/*    This program illustrates the use of m_lock.  The program counts  */
/*    by increasing the value of a variable in parallel.  The program  */
/*    maintains mutual exclusion with a call to m_lock.  Each copy of  */
/*    the procedure increments the shared variable "count"              */
/*    concurrently, and therefore, mutual exclusion is required.       */
/*    The program also shows how to use the  global counter m_next.    */
/*    Each time m_next is called, its value is incremented.  Each      */
/*    process will increment count until it has been incremented       */
/*    nine times.                                                      */

counts ()
{
    int me;
    me = m_get_myid ();              /*  Get my PID  */

    while (m_next () <= N)  {
        m_lock ();
        count += 1;
        printf ("process %d says count is %d\n", me, count);
        fflush (stdout);
        m_unlock ();
    }
}

main ()
{
    count = 0;

    m_set_procs (NPROCS);        /*  Create NPROCS processes  */
    m_fork (counts);
    m_kill_procs ();             /*  Terminate all Processes except Parent  */

    printf ("counter over\n");

}
process 0 says count is 1
process 1 says count is 2
process 2 says count is 3
process 1 says count is 4
process 0 says count is 5
process 1 says count is 6
process 0 says count is 7
process 2 says count is 8
process 1 says count is 9
counter over
```

3.14 Matrix Multiply

So far only very simple (and somewhat useless) programs have been used to illustrate the function of the Parallel Programming Library. Example 22 demonstrates "Data Partitioning". The program is to multiply two six by six matrices. Each row of matrix A will be multiplied by every column of matrix B to produce a new row in matrix C. It appears very natural to partition the data by rows. Therefore, "row" is a routine that will multiply one row of matrix A by every column in matrix B to produce a new row in matrix C.

The program declares each matrix A, B, and C to be **shared**. The main process then calls **init_matrices** to read in matrix A and B. It sets the number of processes to be created to six by using the call to **m_set_procs**. It then calls **m_fork** to create the processes and to have each execute the function "row". Six processes were created, one for each row in Matrix C.

Upon executing "row", each process immediately gets its PID using **m_get_myid**. Remember that the PIDs range from 0 to 5. The row indices of matrix C also run from 0 to 5. This is more than a coincidence. Each process i will produce row i in matrix C by multiplying row i in matrix A by every column in matrix B.

After each process is finished, the main process terminates all the child processes (executing "row") and prints out the results. It would not have accomplished anything to have each process print out its own results, since the output must be sequential.

```
                        /*********************/
                        /*  Example 22       */
                        /*  Matrix Multiply  */
                        /*********************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define N 6


shared int c[N][N], a[N][N], b[N][N];

/*    This procedure multiplies row i of matrix A by    */
/*    each column of matrix B and stores the result     */
/*    in row i of matrix C.                             */


void
row ()
{
    int i,j,k;

    i = m_get_myid ();          /*  Which row do I multiply */

    for(j=0; j<N; j++)  {
       c[i][j] = 0;
       for(k=0; k<N; k++)
          c[i][j] += a[i][k] * b[k][j];
    }
}

/*    This procedure reads in two N by N matrices    */

void
init_matrices ()
{
    int i,j;
    for (i=0; i<N; i++)  {
       scanf ("%d%d%d%d%d%d%d%d%d%d%d%d", &a[i][0], &a[i][1], &a[i][2],
               &a[i][3], &a[i][4], &a[i][5], &b[i][0], &b[i][1], &b[i][2],
               &b[i][3], &b[i][4], &b[i][5]);
    }
}


/*  This program multiplies two N by N matrices, A and B to get  */
/*  matrix C.  The program is executed in parallel by creating   */
/*  N processes with m_fork.  Each child process will multiply   */
/*  row i of matrix A by each column of matrix B to get row i of */
/*  matrix C, where i is the PID of the process.  All three      */
/*  Matrices are in shared memory for each process to access.    */
/*  Since each process is writing to a separate row in C, no     */
/*  synchronization to access memory is necessary.               */

main()

{
    void init_matrices (), row ();
    int i,j;
    init_matrices ();                       /*  read in matrices  */

    m_set_procs (N);
    m_fork (row);                   /*  create 6 processes  */
    m_kill_procs ();

/*  print out each matrix  */
```

```
printf ("       MATRIX A                    MATRIX B                    MATRIX C\n");
printf ("       --------                    ---------                   ---------\n\n");
   for (i=0; i<N; i++)  {
      for (j=0; j<N; j++)
         printf ("%3d ", a[i][j]);
      printf ("     ");
      for (j=0; j<N; j++)
         printf ("%3d ", b[i][j]);
      printf ("     ");
      for (j=0; j<N; j++)
         printf("%3d ", c[i][j]);
      printf("\n");
   }
}
```

```
      MATRIX A                    MATRIX B                    MATRIX C
      --------                    ---------                   ---------

  2   2   2   2   2   2       3   3   3   3   3   3       66  66  66  66  66  66
  3   3   3   3   3   3       4   4   4   4   4   4       99  99  99  99  99  99
  4   4   4   4   4   4       5   5   5   5   5   5      132 132 132 132 132 132
  5   5   5   5   5   5       6   6   6   6   6   6      165 165 165 165 165 165
  6   6   6   6   6   6       7   7   7   7   7   7      198 198 198 198 198 198
  7   7   7   7   7   7       8   8   8   8   8   8      231 231 231 231 231 231
```

3.15 Shared Memory

Shared memory is a very effective and efficient mechanism for communication. Any variable which is shared between processes and changed by those processes is a type of communication. The difficulty of shared memory is mutual exclusion. This means that only one process should update a shared data item at one time. Otherwise, the processes could produce incorrect results. Example 23 illustrates how shared memory can be used for a more explicit type of communication.

This program creates three processes. The three processes are arranged logically in a circle. In other words, process i can only talk to process $i + 1$ and $i - 1$ (the process on its left and right). The process index will be its PID. Each process will have a mailbox. A process can only write to its mailbox, but can read from any mailbox. The processes are to pass a number from one process to another around the circle. After the number has been passed in a complete circle, the parent will print the number out.

The main program begins by initializing each mailbox to 0 which designates an empty mailbox. The mailboxes are declared as an array in shared memory, so that each process can use indices to read the mailboxes. The main process then creates three processes to execute "nodes". In "nodes", each process immediately gets its PID and computes its neighbor's indices. The parent process ($my\_node = 0$) reads in the value of the number to pass. It then places the number in its mailbox for its left neighbor to read. The parent process then spins while its right neighbor's mailbox is empty (0). Each child process does the same. Once a process can read its neighbor's mailbox, it places the number in its own mailbox for another process to read. When the parent receives the number, it prints the value. Each process prints the PID of the process which will read the number next.

This program shows that not only can a shared variable be used to pass information, but that it can be the foundation for synchronization (the busy-waits).

```
                                /****************/
                                /*  Example 23  */
                                /*  Mailboxes   */
                                /****************/
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define NPROCS 3


shared int mbox[NPROCS];

/*    Each process runs this routine.  The node (process) will pass */
/*    the Card by placing it in its mailbox.  Each Node will read   */
/*    the card from the mailbox on its left.  Node 0 which is       */
/*    actually the parent process, starts by reading in the card to */
/*    pass around the circle.                                       */
/*    Synchronization is accomplished by each Node busy-waiting     */
/*    until its neighbor's mailbox is not empty.                     */

nodes ()
{
    int card;
    int my_node, neighbor, next_node;

    my_node = m_get_myid ();                        /*  get my process id  */
    neighbor = (my_node + (NPROCS - 1)) % NPROCS;   /*  who is my neighbor  */
    next_node = (my_node + 1) % NPROCS;             /*  who reads my mailbox */

    if (my_node == 0)  {
       printf ("Enter the Card to Pass (1-10): ");
       scanf ("%d", &card);
       printf ("\nCard to pass is %d\n", card);
       fflush (stdout);

       /* Place card in my mailbox for my neighbor to read */

       printf ("Node %d passes %d to Node %d\n", my_node, card, next_node);
       fflush (stdout);
       mbox[my_node] = card;

       while (mbox[neighbor] == 0) ;  /*  Busy-Wait until card is in mailbox  */

       /*  Read mailbox and print card */

       printf ("\n\nCard Passed through All Nodes, Returned value is %d\n",
                mbox[neighbor]);
       fflush (stdout);
    }
    else   {            /*  if I am not the parent process  */

       while (mbox[neighbor] == 0) ;  /*  Busy-Wait until mailbox not empty  */

       printf ("Node %d passes %d to Node %d\n", my_node, mbox[neighbor],
                next_node);
       fflush (stdout);

       mbox[my_node] = mbox[neighbor];
    }
}


/*    This is the main process   It starts the nodes and kills them after  */
/*    they have finished                                                    */

main ()
{
}
```

```
    int i;

    for (i=0; i < NPROCS; i++)  {      /* initialize boxes  */
       mbox[i] = 0;
    }

    m_set_procs (NPROCS);     /*  set number of players */
    m_fork (nodes);           /*  start the game        */
    m_kill_procs ();          /*  game is over          */
}
```

```
Enter the Card to Pass (1-10):
Card to pass is 5
Node 0 passes 5 to Node 1
Node 1 passes 5 to Node 2
Node 2 passes 5 to Node 0


Card Passed through All Nodes, Returned value is 5
```

4 Cobegin–Coend Implementation

Section 3 introduced both the **fork** and **m_fork** routines and demonstrated how each is used for process creation. The **fork**, **exit**, and **wait** routines, when used together, provide for process creation, termination, and synchronization. The problem with these routines is that they can become confusing to the programmer. Omission and commission errors are also a threat. The **m_fork** routine is a higher level process creation mechanism. It names the specific routine to be placed in execution and provides for process termination. However, **m_fork** can only create a limited number of processes (the number of available processors minus one) and each process will execute the same routine. This is satisfactory since **m_fork** was designed for data partitioning.

We require a mechanism which will allow a programmer to create as many processes as the operating system allows. This mechanism should also allow the processes to execute different sections of code to support both data partitioning and functional partitioning. Each of these requirements are met by the construct "cobegin–coend". A cobegin–coend construct is a block of code which is a structured method of creating processes. The cobegin–coend structure was derived from Dijkstra's "Parbegin–Parend" construct. The syntax and semantics of the cobegin–coend are as follows.

```
cobegin
        statement 1
        statement 2
        statement 3
        ...
        statement N
coend
```

Each statement within the cobegin–coend block is executed concurrently and may be any valid C statement including function calls or block statements. The execution of code after the cobegin–coend block will not proceed until every statement within the cobegin-coend block has terminated. The cobegin–coend automatically provides for process creation and termination while retaining much of the flexibility of the **fork** routine. This higher level construct allows programmers to easily structure concurrent programs. A reader of a program containing this construct can clearly identify all tasks marked for concurrent execution. Also, this construct is structured (one way in and one way out) and can easily be nested. However, the cobegin-coend is not all powerful. Its major weakness is that it

can not handle dynamic applications. For example, an application which does not know how many separate processes it requires until run time.

4.1 Precompiler Logic

A precompiler was written to implement the cobegin–coend construct. The precompiler was written in C and prepares a C program which contains cobegin–coend blocks for the C compiler. The function of the precompiler is to find cobegin–coend blocks and to transform each block into a set of routines which provide for process creation, termination, and synchronization. The **fork**, **exit**, and **wait** routines are used to provide this functionality. Each of these routines is found in Unix, as well as DYNIX, which adds to the portability of the precompiler. The effect of the precompiler generated code is that each statement within the cobegin–coend block will be **forked** by the parent process and executed by a child process (process creation). After executing a statement. each child process will **exit** (process termination). At the end of the cobegin–coend block, the parent will call the **wait** routine for each child created (process synchronization). An example of the generated code follows:

Before Precompilation

```
cobegin
        statement 1
        statement 2
coend
```

After Precompilation

```
pid = fork ();
if (pid != 0)
        pidarray[jj++] = pid;
if (pid == 0) {
        statement 1
        exit (0); }
pid = fork ();
if (pid != 0)
        pidarray[jj++] = pid;
if (pid == 0) {
        statement 2
        exit (0); }
for (ii=0; ii<2; ii++) {
        pid = wait (&status);
        if (status) {
                jj = 0;
                while (pid != pidarray[jj])
                        jj++;
                printf ("Error on Stmt %d in cobegin block", jj);
        }
}
```

The code generated after precompilation shows that a **fork** was performed for each statement. If the PID (process id) is not zero (designating the parent), add the PID to an array of PIDs. If the PID is zero (designating the child), execute the statement and then **exit**. At the end of the cobegin-coend block, the parent performs a **wait** for each child. The parent tests the status of the terminating child and will print an error message if the child terminated with an error code. The error message contains the statement number of the statement the child executed relative to the beginning of the cobegin-coend block. The reason for printing this error message is to aid the programmer in debugging a program in a concurrent environment. The array pidarray and the integers ii, jj, and pid are inserted for the programmer in order to keep the implementation of the cobegin–coend construct transparent. Notice that the parent process creates a child for each statement and does not execute one of the statements itself. This decision was for simplicity. The precompiler would need to read the source code twice or would need to store full statements in memory in order to allow the parent process to execute one of the statements. This is because the precompiler reads one line at a time and has no look ahead capability. The precompiler can not predict the length of a statement.

Appendix A contains the source code for the precompiler. The function of the precompiler is simple. The most difficult aspect of the precompiler is the recognition and separation of C statements. The statement within the cobegin–coend block can be any valid C statement including blocks of code such as *for*, *while*, and *do* statements. A stack is used to separate the statements within the cobegin-coend block. A user may invoke the precompiler by the command *cobegin filename*. The precompiler will first check to see that the user has entered the filename of a C source file. It then attempts to open the file and to create two more files, a new source file, and a trace file. The new source file will contain the program code after precompilation. The trace file will contain a trace of all stack operations performed by the precompiler. The names of these two files are based on the name of the original source file. If the original source file's name is xxxx.c, the new source file will be named xxxxp.c, and the trace file will be named xxxxt.d. The trace file will only be retained if an error occurs during precompilation.

The precompiler allows up to six source files to be entered by the user at a time. The purpose of the "main" routine is to read the command line arguments and to create a process for each entered file. The "main" routine calls the function "find_block" for each file entered. "find_block" creates and opens all required files and begins a search for any cobegin-coend block in the source file. The keywords "cobegin" and "coend" designate a

cobegin–coend block and each must be in lower case and on a line by themselves. These requirements do not effect the programmer and ease the search for the cobegin–coend blocks. "find_block" will call the function "forkstmts" for each cobegin–coend block found. "find_block" will also insert the declarations for the variables ii, jj, pid, and pidarray. These variables are placed outside the source's "main" routine. This was done because the "main" routine is easy to find and every C program must have this routine. The precompiler supports separate compilation since only routines which contain cobegin–coend blocks need to be precompiled. However, the "main" routine must always be precompiled. "find_block" will close all files after the source program has been precompiled.

The function "forkstmts" is the heart of the precompiler. Its function is to separate the statements of the cobegin–coend block and to call the appropriate functions for inserting any required code. The functions "prforks", "prwaits", and "printexit" are used to insert the required calls to **fork**, **wait**, and **exit** respectively. At the beginning of each statement, "forkstmts" calls the function "prforks". The function "printexit" is called when the end of a statement is found. The function "prwaits" is called when the end of the cobegin–coend block is found. "forkstmts" first initializes the stack by placing a semicolon on its top. It is initially assumed that the end of each statement will be a semicolon. The logic of separating the C statements is as follows:

1. Whenever a symbol is found which matches the top of the stack, pop the stack. The only symbols which are screened by "forkstmts" are quotes, semicolons, left parenthesis, right parenthesis, {, and }.

2. Between each statement, "forkstmts" will search for the keyword "coend". This marks the end of the cobegin–coend block and "forkstmts" will return. If the keyword "cobegin" is found an error is printed and "forkstmts" will return with an error. A new cobegin–coend block as a statement within a cobegin–coend block has no meaning.

3. Newline characters are always recognized. The precompiler reads the old source file one line at a time. The current line is printed to the new source file and a new line is read from the old source file on each newline character.

4. If the quote symbol is found within a statement, ignore every symbol except newline characters until another quote symbol is found.

5. If a semicolon symbol is found and the top of the stack is a semicolon, then pop the stack.

If the stack is empty, the end of the statement has been found.

6. If a left parenthesis symbol is found and the top of the stack is either a right parenthesis or a semicolon, push a right parenthesis symbol on the stack.

7. If a right parenthesis symbol is found and the top of the stack is a right parenthesis, pop the stack.

8. If a do while loop is found at the beginning of a statement, turn on the flag "dostmt". The reason for this flag will become apparent shortly.

9. If a { symbol is found and the top of the stack is a semicolon and dostmt is FALSE, pop the stack and push a } symbol on the stack. In the case of dostmt being TRUE, do not pop the stack and just push on the symbol }. The reason for this is that a do while statement ends in a semicolon symbol, but contains a { } block. If a { is found and the top of the stack is a }, push a } symbol on the stack.

10. If a } symbol is found and the top of the stack is a }, pop the stack and check for empty stack. If the stack is empty, the end of the statement has been found.

11. If the keyword "cobegin" is found within a statement, then push the symbol & on the stack and call "forkstmts" recursively. The & symbol is called a stack separator and is used to designate the empty stack. The stack separator is used to separate different segments of the stack which reflect different cobegin-coend blocks. This allows programmers to correctly nest cobegin–coend blocks.

An important aspect of the precompiler is that it expects to receive a syntactically correct C program. If the syntax of the C source file is incorrect, the results of the precompiler can not be predicted. Any programmer using the precompiler, should first comment out the keywords "cobegin" and "coend" and try to compile the source code. This will inform the programmer if his source code is syntactically correct. The precompiler also attempts to produce structured code. When the precompiler finds a cobegin–coend block, it remembers the column number where the keyword "cobegin" was found. All inserted code is then blocked relative to this column number. Thus, if the precompiler receives a structured program, it will produce a structured program. Also remember that the variable names ii, jj, pid, and pidarray are reserved when using the precompiler. If these names are used in the program, problems could occur.

4.2 Examples of Cobegin–Coend

This section presents a number of examples which will aid the reader in understanding the functionality of the cobegin–coend construct. Each example is very simple. The first four examples illustrate the implementation of the precompiler. Each of these examples presents the new source code after precompilation. These examples are also void of any synchronization between child processes. Again these examples are meant to show the functionality of the cobegin–coend not synchronization primitives. The last examples present some of the classic problems of concurrent environments written in C using the cobegin–coend construct.

4.2.1 Function Calls

Example 24a is composed of two routines which share an array of counters. The main routine wishes to increment each counter by ten, concurrently. The main routine first initializes each counter and then calls the function "add" for each counter. The function "add" accepts an index into the array of shared counters and the value to add to a counter. The main routine calls "add" within a cobegin–coend block. This ensures that every call to "add" is made concurrently. Example 24b shows the changes made to the source code by the precompiler. The syntax and semantics of the code are as described in section 4.1. Notice that the array "pidarray" and the integers ii, jj, and pid are placed before the main routine. The array size is set to 25 because that is the maximum number of processes any one user can create on the current system at one time. This value can easily be changed in the precompiler function "find_block". Also notice that the appearance of the inserted code is in a structured format. In this example, it would have been just as easy to perform the two statements of the "add" function in a block of code within the cobegin–coend block. However, these examples are not presenting realistic situations, but are only meant to show functionality.

```
                    /*****************/
                    /*  Example 24a  */
                    /*****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to that counter.  The main routine            */
/*  initializes each counter to zero and then adds 10 to each   */
/*  counter concurrently using a cobegin - coend block.         */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  add 10 to each counter concurrently  */

   cobegin
      add (0, 10);
      add (1, 10);
      add (2, 10);
      add (3, 10);
      add (4, 10);
   coend

   printf ("Everyone is done\n");
}
```

```
                    /*****************/
                    /*  Example 24b  */
                    /*****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to that counter.  The main routine            */
/*  initializes each counter to zero and then adds 10 to each   */
/*  counter concurrently using a cobegin - coend block.         */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

int pidarray[25];
int status, pid, ii;
static int jj = {1};
main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  add 10 to each counter concurrently  */

   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (0, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (1, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (2, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (3, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (4, 10);
      exit (0);  }
   for (ii = 0; ii < 5; ii++)  {
      pid = wait (&status);
```

```
        if (status)  {
            jj = 0;
            while (pid != pidarray[jj])
                jj++;
            printf ("Error on Stmt %d in cobegin block\n",jj);
        }
    }

    printf ("Everyone is done\n");
}
```

4.2.2 Blocks of Code

Example 25a shows the exact same program shown in example 24a. However, counter 2 will be incremented three times and counter 3 will incremented twice. Again the function "add" accepts the index of the counter and the value to add to the counter. Notice that the block symbols { and } are placed around all the calls to "add" for counter 2 and counter 3. These two blocks of code will be executed concurrently with every other statement in the cobegin–coend block. However, each call within these two blocks are executed sequentially. In this example, no synchronization mechanisms are needed for mutual exclusion since multiple adds on a specific counter are executed sequentially. The blocking of code is important to the cobegin–coend construct because it allows the declaration of local variables and allows the programmer the ability to perform some sequential execution within the cobegin–coend block. Of course calling a function within the cobegin–coend block has the same result, but with the overhead of a function call. Example 25b shows the precompiler output for example 25a. Notice that for each block statement, the precompiler still generated enclosing brackets for the child process. Both pairs of { } are not needed. Both pairs of brackets were retained for simplicity to the precompiler.

```
                    /*****************/
                    /*  Example 25a  */
                    /*****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to that counter.  The main routine            */
/*  initializes each counter to zero and then adds 10 to        */
/*  counters 0, 1, and 4.  The main routine adds 30 to counter  */
/*  2 in increments of 10 and adds 20 to counter 3 in           */
/*  increments of 10.  Counters 2 and 3 are incremented         */
/*  sequentially by using the symbols { and } to block the      */
/*  code.                                                       */


shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  Increment each buffer concurrently  */

   cobegin
      add (0, 10);
      add (1, 10);
      {  add (2, 10);
         add (2, 10);
         add (2, 10);  }
      {  add (3, 10);
         add (3, 10);  }
      add (4, 10);
   coend

   printf ("Everyone is done\n");
}
```

```
                    /*****************/
                    /*  Example 25b  */
                    /*****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to that counter.  The main routine            */
/*  initializes each counter to zero and then adds 10 to        */
/*  counters 0, 1, and 4.  The main routine adds 30 to counter  */
/*  2 in increments of 10 and adds 20 to counter 3 in           */
/*  increments of 10.  Counters 2 and 3 are incremented         */
/*  sequentially by using the symbols { and } to block the      */
/*  code.                                                       */


shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

int pidarray[25];
int status, pid, ii;
static int jj = {1};
main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);   /*  no output buffer  */

   /*  Increment each buffer concurrently  */

   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (0, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      add (1, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      {  add (2, 10);
         add (2, 10);
         add (2, 10);  }
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      {  add (3, 10);
         add (3, 10);  }
      exit (0);  }
```

```
    pid = fork ();
    if (pid != 0)
       pidarray[jj++] = pid;
    if (pid == 0)  {
       add (4, 10);
       exit (0);  }
    for (ii = 0; ii < 5; ii++)  {
       pid = wait (&status);
       if (status)  {
          jj = 0;
          while (pid != pidarray[jj])
             jj++;
          printf ("Error on Stmt %d in cobegin block\n",jj);
       }
    }

    printf ("Everyone is done\n");
}
```

4.2.3 Block Statements

Example 26a is a mild change to example 24a. In this example, each counter is incremented ten times by calling "add" ten times. Each time the function "add" increments the counter by ten. In the cobegin–coend block a *for* loop is placed around each call to "add". This does not mean that 50 processes are created, but that five processes are created, each of which executes one of the *for* loops. Again, this example certainly is not the most effective method for incrementing these loops. It would be best to bypass the ten function calls for each process. Example 26b contains the precompiler output for example 26a. Notice that each entire *for* loop is contained in a block of code to be executed by a child process. The *for* loops could have been different sizes.

```
                    /*****************/
                    /*  Example 26a  */
                    /*****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function  */
/*  "add" receives an index into the array of counters and the   */
/*  amount to add to that counter.  The main routine             */
/*  initializes each counter to zero and then adds 100 to each   */
/*  counter concurrently using a cobegin - coend block.          */
/*  Each counter is incremented to 100 by 10s.                   */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  add 100 to each counter concurrently  */

   cobegin
      for (i = 0, i < 10; i++)
         add (0  10);
      for (i = 0, i < 10; i++)
         add (1  10);
      for (i = 0, i < 10; i++)
         add (2, 10);
      for (i = 0, i < 10; i++)
         add (3, 10);
      for (i = 0    < 10; i++)
         add (4, 10),
   coend

   printf ("Everyone is done\n");
}
```

```
                    /****************/
                    /*  Example 26b */
                    /****************/

#include <stdio.h>

/*  This program increments an array of counters.  The function */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to that counter.  The main routine            */
/*  initializes each counter to zero and then adds 100 to each  */
/*  counter concurrently using a cobegin ~ coend block.         */
/*  Each counter is incremented to 100 by 10s.                  */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

int pidarray[25];
int status, pid, ii;
static int jj = {1};
main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  add 100 to each counter concurrently  */

   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      for (i = 0; i < 10; i++)
         add (0, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      for (i = 0; i < 10; i++)
         add (1, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      for (i = 0; i < 10; i++)
         add (2, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      for (i = 0; i < 10; i++)
         add (3, 10);
      exit (0);  }
   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
```

4.2.4 Nesting Cobegin Blocks

Example 27a illustrates the programmer's capability to nest cobegin–coend blocks. In this example, each of the five counters are to be incremented by different values. Counters 0, 1, and 2 are to incremented until their combined values exceeds 110. Counters 3 and 4 are to be incremented until their combined values exceed 75. Notice that two independent *while* loops are used to test and increment the two sets of counters. These two loops are independent and can execute concurrently. Thus, they are placed within a cobegin–coend block. Also notice that each call to "add" is independent and can be executed concurrently with every other add operation. Thus, each call to "add" within each *while* loop is also placed within a cobegin–coend block. Each cobegin–coend block within a *while* loop provides the synchronization needed by the loop to check the totals of each counter set. Example 27b contains the source code produced by the precompiler for example 27a. The first child process executes the *while* loop containing counters 0, 1, and 2. Also, three child processes are created within the *while* loop, one for each counter. Notice that a structured appearance is maintained.

```
    if (pid == 0)  {
       for (i = 0; i < 10; i++)
          add (4, 10);
       exit (0);  }
    for (ii = 0; ii < 5; ii++)  {
       pid = wait (&status);
       if (status)  {
          jj = 0;
          while (pid != pidarray[jj])
             jj++;
          printf ("Error on Stmt %d in cobegin block\n",jj);
       }
    }

    printf ("Everyone is done\n");
}
```

```
                    /****************/
                    /*  Example 27a */
                    /****************/

#include <stdio.h>

/*  This program increments an array of counters. The function  */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to the counter.  The main routine increments  */
/*  counters 0, 1, and 2 until their values add up to more than */
/*  110.  It increments counters 3 and 4 until their values add */
/*  up to more than 75.  Each addition of a counter is done     */
/*  concurrently.  This program illustrates nesting of cobegin  */
/*  blocks.                                                     */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);   /*  no output buffer  */

   /*  Increment counters concurrently */

   cobegin
      while (count[0] + count[1] + count[2] < 110)  {
         cobegin
            add (0, 7);
            add (1, 10);
            add (2, 15);
         coend
      }
      while (count[3] + count[4] < 75)  {
         cobegin
            add (3, 10);
            add (4, 10);
         coend
      }
   coend

   printf ("Counters 0,1, and 2 added equal %d\n", count[0]+count[1]+count[2]);
   printf ("Counter 3 ond 4 added equal %d\n", count[3]+count[4]);
}
```

```
                    /****************/
                    /*  Example 27b */
                    /****************/

#include <stdio.h>

/*  This program increments an array of counters. The function  */
/*  "add" receives an index into the array of counters and the  */
/*  amount to add to the counter.  The main routine increments  */
/*  counters 0, 1, and 2 until their values add up to more than */
/*  110.  It increments counters 3 and 4 until their values add */
/*  up to more than 75.  Each addition of a counter is done     */
/*  concurrently.  This program illustrates nesting of cobegin  */
/*  blocks.                                                     */

shared int count[5];        /*  array of counters  */

add (i, n)
   int i, n;
{

   count[i] = count[i] + n;
   printf ("Counter %d is now %d\n", i, count[i]);
}

int pidarray[25];
int status, pid, ii;
static int jj = {1};
main ()
{
   int i;

   for (i = 0; i < 5; i++)
      count[i] = 0;
   setbuf (stdout, NULL);    /*  no output buffer  */

   /*  Increment counters concurrently */

   pid = fork ();
   if (pid != 0)
      pidarray[jj++] = pid;
   if (pid == 0)  {
      while (count[0] + count[1] + count[2] < 110)  {
         pid = fork ();
         if (pid != 0)
            pidarray[jj++] = pid;
         if (pid == 0)  {
            add (0, 7);
            exit (0);  }
         pid = fork ();
         if (pid != 0)
            pidarray[jj++] = pid;
         if (pid == 0)  {
            add (1, 10);
            exit (0);  }
         pid = fork ();
         if (pid != 0)
            pidarray[jj++] = pid;
         if (pid == 0)  {
            add (2, 15);
            exit (0);  }
         for (ii = 0; ii < 3; ii++)  {
            pid = wait (&status);
            if (status)  {
               jj = 0;
               while (pid != pidarray[jj])
                  jj++;
```

```
                printf ("Error on Stmt %d in cobegin block\n",jj);
            }
        }
    }
    exit (0);  }
pid = fork ();
if (pid != 0)
    pidarray[jj++] = pid;
if (pid == 0)  {
    while (count[3] + count[4] < 75)  {
        pid = fork ();
        if (pid != 0)
            pidarray[jj++] = pid;
        if (pid == 0)  {
            add (3, 10);
            exit (0);  }
        pid = fork ();
        if (pid != 0)
            pidarray[jj++] = pid;
        if (pid == 0)  {
            add (4, 10);
            exit (0);  }
        for (ii = 0; ii < 2; ii++)  {
            pid = wait (&status);
            if (status)  {
                jj = 0;
                while (pid != pidarray[jj])
                    jj++;
                printf ("Error on Stmt %d in cobegin block\n",jj);
            }
        }
    }
    exit (0);  }
for (ii = 0; ii < 2; ii++)  {
    pid = wait (&status);
    if (status)  {
        jj = 0;
        while (pid != pidarray[jj])
            jj++;
        printf ("Error on Stmt %d in cobegin block\n",jj);
    }
}

printf ("Counters 0,1, and 2 added equal %d\n", count[0]+count[1]+count[2]);
printf ("Counter 3 and 4 added equal %d\n", count[3]+count[4]);
}
```

4.2.5 Dining Philosophers

Example 28 is a solution to the Dining Philosophers problem. In this problem, there are five philosophers. Each philosopher spends his day in two activities, eating and thinking. After spending a certain amount of time thinking, a philosopher will become hungry and want to eat. In this solution, a philosopher must enter the dining room to eat. Only four philosophers are allowed in the dining room at a time. This restriction ensures the absence of deadlock. The dining room contains a large round table with five place settings, one for each philosopher. In the center of the table is a large bowl of spaghetti. There are a total of five forks on the table, one between each place setting. After a philosopher has entered the dining room, he must first pick up the fork on his left and then pick up the fork on his right in order to eat the spaghetti.

This example combines the **s_lock** routines provided by DYNIX and the cobegin–coend construct provided by the precompiler. There are six locks. An array of five locks is declared to represent each of the five forks. Thus, when a philosopher attempts to pick up a fork which is being used (locked), he must wait. If a philosopher picks up his left fork and the right fork is being used, he will not put down the left fork. The lock "room" is used to monitor the amount of philosophers in the dining room. The variable "occupy" holds the number of philosophers in the dining room. To enter the dining room, a philosopher will first obtain the lock "room" and then check the variable "occupy". If "occupy" is less than four, the philosopher increments "occupy", releases the lock, and enters the dining room. If "occupy" is equal to four, he releases the lock and tries again. The main routine creates five philosophers using a cobegin–coend block. Each philosopher receives his philosopher number, the index into "forks" for his left fork, and the index to his right fork. Notice how easy it was to place each philosopher into execution on a separate process by using the cobegin–coend construct. Also, notice how clearly the cobegin–coend block defines the concurrent tasks.

```
                        /*****************/
                        /*  Example 28   */
                        /*****************/

#include <parallel/microtask.h>
#include <parallel/parallel.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0

/*  This program is a solution to the Dining Philosophers problem.    */
/*  In this problem, there are five philosophers.  Each philosopher   */
/*  does two things, he eats and thinks.  In order to eat, a          */
/*  philosopher must enter the dining room, pick up his right fork    */
/*  and pick up his left fork.  The problem is that there are only    */
/*  five forks, one between each of five place settings.  In this     */
/*  solution, the dining room acts as a lock and allows only four     */
/*  philosophers to enter.  Each fork is also a lock.  If a           */
/*  philosopher attempts to pick up a fork and finds that it is       */
/*  already in use, the philosopher will wait for the fork to be      */
/*  placed back on the table.  After a philosopher has eaten, he      */
/*  will put down both forks and leave the dining room to continue    */
/*  thinking.  The forks are declared as an array of locks.  The      */
/*  philosophers receive indices into the array which designate       */
/*  both their left and right forks.                                  */

shared slock_t forks[5], room;
shared int occupy;

think (philnum)
   int philnum;
{
   printf ("Philosopher %d is Thinking\n", philnum);
}

eat (philnum)
   int philnum;
{
   printf ("Philosopher %d is Eating\n", philnum);
}

pickupfork (forknum)
   int forknum;
{
   s_lock (&forks[forknum]);
}

putdownfork (forknum)
   int forknum;
{
   s_unlock (&forks[forknum]);
}

enterroom (philnum)
   int philnum;
{
   int in;
   in = FALSE;

   while (! in)  {
      s_lock (&room);
      if (occupy < 4)  {  /*  Is there room for me to enter  */
         occupy++;
         in = TRUE;
         printf ("Philosopher %d has Entered Dining Room\n", philnum);
      }
      s_unlock (&room);
```

```
    }
}

exitroom (philnum)
    int philnum;
{
    s_lock (&room);
    occupy--;
    printf ("Philosopher %d has left the Dining Room\n", philnum);
    s_unlock (&room);
}

phil (philnum, left, right)
    int philnum, left, right;
{
    int days;

    for (days = 0; days < 5; days++)  {
        think (philnum);
        enterroom (philnum);
        pickupfork (left);
        pickupfork (right);
        eat (philnum);
        putdownfork (left);
        putdownfork (right);
        exitroom (philnum);
    }
}

main ()
{
    int i;

    for (i = 0; i < 5; i++)          /*  creat locks  */
        s_init_lock (&forks[i]);
    s_init_lock (&room);

    /*  Begin ecch Philosopher   */

    cobegin
        phil (0, 4, 0);
        phil (1, 0, 1);
        phil (2, 1, 2);
        phil (3, 2, 3);
        phil (4, 3, 4);
    coend
}
```

4.2.6 Bounded Buffer

Example 29 shows a solution to the Bounded Buffer problem. In this problem, there are two producers and two consumers. Each producer wishes to write to an array of buffers and each consumer wishes to read from the array of buffers. The problem is that the array of buffers is limited in size. So, if the producers write faster than the consumers read, they will overwrite their data. If the consumers read faster than the producers write, they will read either old data or nonexistent data. This is a basic synchronization problem. In this example, the array can hold up to 10 buffers. Each producer will write 10 messages to the array of buffers for a total of 20 messages. Each consumer will read 10 messages. Each buffer will hold two items, the producer number and a message number. The lock "prods_lk" is used to ensure that each producer does not attempt to write to the same buffer. The lock "cons_lk" is used to ensure that each consumer does not attempt to read from the same buffer. These two locks ensure mutual exclusion. Another type of synchronization problem is conditional synchronization. In this example, the variable "empty" holds the number of empty buffers. Each time a buffer is read by a consumer, empty is incremented. The variable "full" is used to hold the number of full buffers. Each time a producer writes to a buffer, full is incremented. A producer can only write to the array of buffers if empty is greater than zero and a consumer can only read from the array of buffers if full is greater than zero. This is referred to as conditional synchronization. The two locks "full_lk" and "empty_lk" are used to ensure mutual exclusion when updating "full" and "empty", respectively. Once again each producer and consumer was created and placed in execution using a cobegin–coend block. This example demonstrates the cobegin–coend construct's capability to create processes which execute different routines.

```
                    /*****************/
                    /*  Example 29   */
                    /*****************/

#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define N 10
#define TRUE 1
#define FALSE 0

/*  This program illustrates the Bounded Buffer problem.  In this      */
/*  program, there are two producers and two consumers.  The producers */
/*  write their ID and a message number to a shared array buffer.  The */
/*  consumers  read the buffer and print a message.  The message tells */
/*  the message number and the producer who wrote it.  The shared      */
/*  buffer can hold 10 messages.  Each producer will write 10 messages */
/*  for a total of 20 messages.  There are four locks to ensure mutual */
/*  exclusion when reading and writing a message and to ensure that    */
/*  buffer does not overflow or underflow.                             */
/*        cons_lk -  only one consumer  may read at a time             */
/*        prod_lk -  only one producer may write at at time            */
/*        empty_lk - mutual exclusion on the variable empty            */
/*        full_lk -  mutual exclusion on the variable full             */
/*  The variable empty tells how many buffers are empty and is         */
/*  initialized to 10.  The variable full tells how many messages are  */
/*  in the array of buffers.  Only one producer may write to a buffer  */
/*  at a time and only if empty is greater than zero.  Only one        */
/*  consumer may read from a buffer at a time and only if full is      */
/*  greater than zero                                                  */

shared slock_t prod_lk, cons_lk, full_lk, empty_lk;
struct entry {
   int prnum;
   int msgnum;
};
shared int in, out, empty, full;
shared struct entry buffer[N];       /*  array of buffers  */

producer (num)
   int num;
{
   int i, cont;

   for (i = 0; i < N; i++)  {

      cont = FALSE;
      while (! cont)  {          /*  wait until there is an empty buffer  */
         s_lock (&empty_lk);
         if (empty > 0)  {
            cont = TRUE;
            empty--;
         }
         s_unlock (&empty_lk);
      }

      s_lock (&prod_lk);              /*  Enter Critical Region  */
      buffer[in].msgnum = i + 1;
      buffer[in].prnum = num;
      in = (in + 1) % N;
      s_unlock (&prod_lk);            /*  Exit Critical Region  */

      s_lock (&full_lk);
      full++;                      /*  Increment # of full buffers  */
      s_unlock (&full_lk);
      sleep (1);
   }
```

```
}


consumer (num)
   int num;
{
   int i, cont;

   for (i = 0; i < N; i++)  {

      cont = FALSE;
      while (! cont)  }      /*  wait for a full buffer  */
         s_lock (&full_lk);
         if (full > 0)  {
            cont = TRUE;
            full--;
         }
         s_unlock (&full_lk);
      }

      s_lock (&cons_lk);          /*  Enter Critical Region  */
      printf ("Message Number: %d\n", buffer[out].msgnum);
      printf ("From Producer Number: %d\n", buffer[out].prnum);
      printf ("By Consummer Number: %d\n", num);
      fflush (stdout),
      out = (out + 1) % N;
      s_unlock (&cons_lk);        /*  Exit Critical Region  */

      s_lock (&empty_lk);
      empty++;                    /*  Increment # of empty buffers  */
      s_unlock (&empty_lk);
      sleep (1);
   }
}


main ()
{
   in = 0;            /*  pointer to buffers  */
   out = 0;

   empty = N;         /*  All buffers are empty  */
   full = 0;

   /*  Start producers and consummers  */
   cobegin
      producer (0);
      producer (1);
      consumer (0);
      consumer (1);
   coend
}
```

4.2.7 Readers/Writers

Example 30 shows a solution to the Readers/Writers problem. This problem is similar to the Bounded Buffer problem. There are a number of readers and writers. Each reader wishes to read a data structure and each writer wishes to write to the data structure. In this problem, any number of readers may read the data structure at a time. A reader does not change the data structure. However, no reader may access the data structure while a writer is writing and only one writer may write at a time. This is a problem of mutual exclusion. In this example, the shared data structure is the integer "value". A reader will read and print "value". A writer simply increments "value" by one. The variable "read_count" holds the number of readers currently reading "value". The variable "wrt" is a flag. "wrt" is TRUE if a writer is writing and FALSE otherwise. The lock "writer_lk" ensures mutual exclusion on both variables. Each reader begins by obtaining the lock and checking "wrt" to see if a writer is writing. If "wrt" is FALSE, the reader increments "read_count" and reads "value". A reader only prints "value" if it has changed since last read. After reading "value", a reader will decrement "read_count". A writer must check "wrt" to see if any other writer is writing and "read_count" to see if there are any readers reading. If "wrt" is FALSE and "read_count" is zero, a writer may proceed. The writer will then set "wrt" to TRUE. After the writer is finished, he will set "wrt" to FALSE. Again, a cobegin–coend block was used to create each reader and writer process.

```
                    /*****************/
                    /*  Example 30   */
                    /*****************/

#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#define TRUE 1
#define FALSE 0

/*  This program demonstrates a solution to the Readers/Writers problem.     */
/*  In this problem, there are three readers and two writers.  The           */
/*  readers will read the variable "value" and if it has changed since they  */
/*  last read it, they will print its value.  The writers constantly try     */
/*  to update the variable "value".  This solution will allow as many        */
/*  readers to access "value" as wish.  However, no readers may access the   */
/*  "value" when a writer is updating it, and only one writer may update     */
/*  "value" at a time.  The variable "read_count" tells how many readers     */
/*  are reading the variable.  The variable  "wrt" is TRUE if a writer is    */
/*  writing.  A reader will proceed only if "wrt" is FALSE.  A writer will   */
/*  proceed only if "wrt" is FALSE and "read_count" is zero.  The lock       */
/*  "writer_lk is used to ensure mutual exclusion on both "read_count" and   */
/*  "wrt".                                                                   */

shared slock_t writer_lk;
shared int value, read_count, wrt;

reader (num)
   int num;
{
   int oldvalue, in;
   oldvalue = 0;
   in = FALSE;

   for ( ; ; ) {        /*  forever do  */
      in = FALSE;
      while (! in) {       /* while I can not enter my crtical section, spin */
         s_lock (&writer_lk);
         if (! wrt)   {         /*  Are any writers writing?  */
            in = TRUE;
            read_count++;
         }
         s_unlock (&writer_lk);
      }

      if (value != oldvalue)  {   /*  If value has changed, print it  */
         oldvalue = value;
         printf ("Reader %d saw value change to %d\n", num, oldvalue);
      }

      s_lock (&writer_lk);
      read_count--;                        /*  Exit Critical Section  */
      s_unlock (&writer_lk);
   }
}

writer ()
{
   int in;

   for ( ; ; ) {
      in = FALSE;
      while (! in) {         /* while I can not enter Critical Section, Spin */
         s_lock (&writer_lk);
         if ((! wrt) && (read_count == 0))  {  /*  Can I write?  */
            in = TRUE;
            wrt = TRUE;
```

```
            }
            s_unlock (&writer_lk);
        }

        value++;        /*  Update value */

        s_lock (&writer_lk);
        wrt = FALSE;            /*  Exiting Critical Section  */
        s_unlock (&writer_lk);
    }
}

main ()
{

    value = 0;
    read_count = 0;
    wrt = FALSE;

    s_init_lock (&writer_lk);

    /*  Start Readers and Writers */
    cobegin
        reader (0);
        reader (1);
        reader (2);
        writer ();
        writer ();
    coend
}
```

4.2.8 Matrix Multiply

Example 31 shows a Matrix Multiply program. This program multiplies two 6 by 6 matrices, A and B, to produce matrix C. This means the each row of matrix A is multiplied by each column of matrix B. This is an example of data partitioning. This solution divides the data by rows of matrix A. Each process will multiply one row of matrix A by every column of matrix B to produce a new row in matrix C. This requires six processes since matrix A has six rows. Each process executes the routine "row" which accomplishes the multiplication. All six processes are again created by a cobegin–coend block. This program is very simple and is used to show the cobegin–coend construct's capability to handle data partitioning. However, notice that the program needed to know the number of rows in matrix A before execution. This shows the cobegin–coend's weakness in a dynamic environment. This weakness could be overcome in this problem by using **m_next**. **m_next** could keep track of the number of rows which have been multiplied. A process could decide to multiply another row of matrix A by checking the value of **m_next**.

```
                        /*****************/
                        /*  Example 31   */
                        /*****************/

#include <stdio.h>
#define N 6
shared int c[N][N], a[N][N], b[N][N];

/*    This procedure multiplies row i of matrix A by    */
/*    each column of matrix B and stores the result in  */
/*    in row i of matrix C.                             */
void
row (i)
   int i;
{
   int j,k;

   for(j=0; j<N; j++)  {
      c[i][j] = 0;
      for(k=0; k<N; k++)
         c[i][j] += a[i][k] * b[k][j];
   }
}

/*    This procedure reads in two 6 by 6 matrices    */
void
init_matrices ()
{
   int i,j;
   printf ("ENTER MATRIX A and B by ROWS\n\n");
   for (i=0; i<N; i++)  {
      printf ("ENTER ROW %d : ", i+1);
      scanf ("%d%d%d%d%d%d%d%d%d%d%d%d",  &a[i][0], &a[i][1], &a[i][2],
             &a[i][3], &a[i][4], &a[i][5], &b[i][0], &b[i][1], &b[i][2],
             &b[i][3], &b[i][4], &b[i][5]);
      printf ("\n");
   }
   fflush (stdout);
}

/*  This program multiplies two N by N matrices, A and B to get   */
/*  matrix C.  The program is executed in parallel by creating    */
/*  N processes with a cobegin. Each child process will multiply  */
/*  row i of matrix A by each column of matrix B to get row i of  */
/*  matrix C, where i is passed to the process.  All three        */
/*  Matrices are in shared memory for each process to access      */
/*  Since each process is writing to a separate row in C, no      */
/*  synchronization to access memory is neccessary.               */

main()

{
   void init_matrices (), row ();
   int i,j;
   init_matrices ();                        /*  read in matrices  */

   cobegin
      row (0);
      row (1);
      row (2);
      row (3);
      row (4);
      row (5);
   coend

/*  print out each matrix  */
```

```
printf ("       MATRIX A                    MATRIX B                    MATRIX C\n");
printf ("       --------                    ---------                   ---------\n\n");
   for (i=0; i<N; i++)  {
      for (j=0; j<N; j++)
         printf ("%3d ", a[i][j]);
      printf ("     ");
      for (j=0; j<N; j++)
         printf ("%3d ", b[i][j]);
      printf ("     ");
      for (j=0; j<N; j++)
         printf("%3d ", c[i][j]);
      printf("\n");
   }
}
```

5 Synchronization

Synchronization mechanisms allow one process to affect the execution of another process. There are two types of process synchronization. First, a process can delay until a specific condition is true. This is referred to as "conditional synchronization". Second, a synchronization mechanism can be used to ensure mutual exclusion. Section 3.10 demonstrates the **s_lock** routine which can be used to ensure mutual exclusion by encapsulating a section of code by the commands **s_lock** and **s_unlock**. The Bounded Buffer problem in section 4.2.6 demonstrated conditional synchronization. In the Bounded Buffer problem a producer can only proceed if the amount of empty buffers is greater than zero. Only a consumer can release an empty buffer and so the consumers effect the execution of producers through a specific condition. Synchronization of processes is based on interprocess communication. For process A to affect process B, process A must communicate some condition to process B. Communication between processes on the Balance 8000 is based on a shared memory architecture. This means that multiple processes communicate by reading and writing to shared data structures in memory. Thus, synchronization of processes in a shared memory architecture is based on setting conditions in memory that multiple processes can detect. Although the Balance 8000 allows programmers to create a large number of locks, its locking routines are based on a set of physical hardware locks. These physical locks ensure mutual exclusion on the software locks which the programmer has created by performing test-and-set operations. This ensures that multiple processes can not obtain the same software lock at the same time. How does a programmer use shared memory to synchronize multiple processes without the help of a hardware test-and-set operation? The following two examples give software solutions to the mutual exclusion problem. After the mutual exclusion problem has been solved, conditional synchronization can be achieved. This is accomplished by placing the condition in shared memory and making it mutually exclusive.

5.1 Peterson's Solution

Example 32 is a software solution to the mutual exclusion problem as presented by Peterson [2]. In this example, two processes wish to increment the same counter. The routine "counts" increments the counter by one and prints out its value. The routines "lock" and "unlock" provide mutual exclusion on this operation using only shared memory. In this solution, each process shares three variables. Two flags are used, one for each process, to indicate whether the process wishes to enter its critical section. The integer "turn" is used to indicate which process may proceed into its critical section. The routine "lock" is called immediately before a processes critical section and the routine "unlock" is called immediately after the critical section. The routine "lock" sets the process's flag to TRUE indicating it wishes to enter its critical section. "lock" then sets the value of "turn" to designate the other process. If "turn" designates the other process and the other process's flag is TRUE, then a process spins until it is either their turn or the other process resets its flag. Notice that if only one process wishes to enter its critical section, then its neighbor's flag will be false and it can proceed. If both processes try to enter the critical section at the same time, "turn" will point to only one of them and that process will proceed. However, after that process is finished, it resets its flag to FALSE and the other process may enter its critical region. This solution is a strongly fair solution since a process will only wait at most one turn before it can enter its critical section.

```
                    /****************/
                    /*  Example 32  */
                    /****************/

#include <stdio.h>
#define TRUE 1
#define FALSE 0

/*  This program is a software solution to the mutual exclusion   */
/*  problem as presented by Peterson.  Two processes wish to      */
/*  increment a counter.  To ensure mutual exclusion each process */
/*  calls the function lock before entering its critical section  */
/*  the function unlock after exiting its critical section.  The  */
/*  processes share two variables, flag and turn.  flag is an     */
/*  array of two flags, one for each processor.  The flag informs */
/*  the other process that you wish to enter your critical        */
/*  section.  To enter its critical section, a process sets its   */
/*  flag to TRUE and sets turn to designate the other process.    */
/*  If the other process wants to enter its critical section and  */
/*  its their turn, then spin.  Otherwise, enter your critical    */
/*  section.  After exiting your critical section, set your flag  */
/*  to FALSE.                                                     */

shared int counter, flag[2], turn;

lock (prnum)
   int prnum;
{
   int j;

   flag[prnum] = TRUE;        /*  I want to enter my critical section  */
   j = (prnum + 1) % 2;
   turn = j;
   while ((flag[j]) && (turn == j)) ;  /* wait for my turn */
}

unlock (prnum)
   int prnum;
{
   flag[prnum] = FALSE;      /*  I have left the critical section  */
}

counts (prnum)
   int prnum;
{
   int i;

   for (i = 0; i < 10; i++)  {
      lock (prnum);
      counter ++;
      printf ("Process %d Increments Counter to %d\n", prnum, counter);
      fflush (stdout);
      unlock (prnum);
   }
}

main ()
{
   counter = 0;
   cobegin
      counts (0);
      counts (1);
   coend
}
```

5.2 Eisenberg and McGuire's Solution

Example 32 showed Peterson's solution to the mutual exclusion problem for two processes. Example 33 shows a software solution for multiple processes as presented by Eisenberg and McGuire [2]. In this example, five processes wish to increment the shared counter. Again, the routines "lock" and "unlock" are used to ensure mutual exclusion by placing "lock" at the beginning of the critical section and "unlock" immediately after the critical section. Each process shares six variables: an array of five flags which designate the state of a process (IDLE, WANTIN, or IN_CS) and the integer "turn" which designates a process that may enter its critical section. This solution is more complicated than Peterson's. The routine "lock" sets the flag of a process to WANTIN and places the value of "turn" in a local variable. The process then spins until the local variable indicates that it is its turn. At this point, the process sets its flag to IN_CS. However, since a local variable was used for the value of "turn", the process does not know which other processes might be in their critical sections. So, the process now spins until no other process is in the state IN_CS. At this point, the process checks to see if "turn" points to it or to an IDLE process. If it is its true, the process enters its critical section. Otherwise, the process gets a new value of "turn" and tries again. The "unlock" routine is executed only by a process which is entering or exiting its critical section. "unlock" sets "turn" to point to the next non_IDLE process in an ordered sequence and sets the flag of the exiting process to IDLE. The "unlock" routine ensures strong fairness in the solution.

```
                    /****************/
                    /*  Example 33  */
                    /****************/

#include <stdio.h>
#define IDLE 0
#define WANTIN 1
#define IN_CS 2
#define N 5

/* This program shows the software solution to the mutual exclusion  */
/*  problem as presented by Eisenberg and McGuire.  There are N       */
/*  processes which wish to increment a counter.  To ensure mutual    */
/*  exclusion while incrementing the counter, each process calls the  */
/*  function "lock" before entering its critical section and calls    */
/*  the function "unlock" after leaving its critical section.  All    */
/*  processes share an array of flags, one per process.  A flag can   */
/*  be in one of three states, IDLE, WANTIN, IN_CS.  They also share  */
/*  the variable turn.  Notice that a process can enter its critical  */
/*  section only if no other process is in its critical section.      */
/*  Also note that the value of turn is modified only when a process  */
/*  enters or exits its critical section.  Once a process has set     */
/*  its flag to IN_CS (thinking that no one else is in their          */
/*  critical section), it waits until turn points to it or the        */
/*  process which turn points to is idle.                             */

shared int counter, flag[N], turn;

lock (prnum)
    int prnum;
{
    int j;

    do {
        flag[prnum] = WANTIN;       /*  I want in my critical section */
        j = turn;
        while (j != prnum)  {       /*  Wait until everyone between   */
            if (flag[j] != IDLE)    /*  me and turn are IDLE          */
                j = turn;
            else
                j = (j + 1) % N;
        }
        flag[prnum] = IN_CS;        /*  I am entering my critical section  */
        j = 0;
        while ((j < N) && ((j == prnum) || (flag[j] != IN_CS)))
            j++;
    }  while ((j < N) || ((turn != prnum) && (flag[turn] != IDLE)));
    turn = prnum;
}

unlock (prnum)
    int prnum;
{
    int j;

    j = (turn + 1 ) % N;            /*  Give turn to next in line  */
    while (flag[j] == IDLE)
        j = (j + 1) % N;
    turn = j;
    flag[prnum] = IDLE;             /*  I am out of my critical section  */
}

counts (prnum)
    int prnum;
{
    int i;
```

```
    for (i = 0; i < N; i++)  {
        lock (prnum);
        counter++;
        printf ("Process %d Increments Counter to %d\n", prnum, counter);
        fflush (stdout);
        unlock (prnum);
    }
}

main ()
{
    int i;

    counter = 0;
    turn = 0;
    for (i = 0; i < N; i++)      /*  set everyone to idle  */
        flag[i] = IDLE;

    cobegin
        counts (0);
        counts (1);
        counts (2);
        counts (3),
        counts (4),
    coend
}
```

```
/*  prforks is called each time a new statement is found within a      */
/*  cobegin-coend block.  The routine inserts the code required to     */
/*  fork a new process and add the PID of each child process to an     */
/*  on-going list.  An array of print statements  (the code to be      */
/*  inserted) is created using the structure 'entry'.  The paramenter  */
/*  'col' holds the column to begin printing the inserted code.        */

prforks (col)
   int col;
{
   int j, i;
   static struct entry stmt[4] = { { "pid = fork ();\n\0"},
                                   {"if (pid != 0)\n\0"},
                                   {"   pidarray[jj++] = pid;\n\0"},
                                   {"if (pid == 0)  {\n\0"} };

   for (j = 0; j < 4; j++)  {
      for (i = 0; i < col; i++)     /*  move to column col  */
         putc (' ', output);
      fprintf (output, stmt[j].ln);  /*  print fork logic  */
   }
   fprintf (trout, "*** New Statement and Fork ***\n");
}
```

```
/*  The routine push will enter a new symbol on the top of the stack   */
/*  push returns a 1 if successful and a 0 if stack Overflow is found. */
/*  If Overflow is found, UPPERBOUND may be reset in the #define       */
/*  statement at the beginning of this program.                        */

push (symbol, ln)
   char symbol;          /*  symbol to be pushed on stack        */
   int ln;               /*  line number where symbol was found  */
{
   fprintf (trout, "Entered Push routine\n");
   if(stk.top >= UPPERBOUND)  {                    /*  OVERFLOW ?  */
      fprintf(stderr, "Overflow on Stack\n");      /*  Print error message  */
      return (0);
   }
   else {
      stk.top++;                                   /*  advance top  */
      fprintf (trout, "Top : %d\n", stk.top);
      fprintf (trout, "Pushed stack char: %c at line: %d\n", symbol, ln);
      stk.sym[stk.top] = symbol;                   /*  add symbol          */
      stk.lnum[stk.top] = ln;                      /*  add current line #  */
      return (1);
   }
}
```

```
/*  The empty routine returns 1 if the stack is empty and 0 otherwise.  */
/*  The character '&' indicates an empty stack.                         */

empty ()
{
   return (stk.sym[stk.top] == '&');
}


/*  The routine pop will delete the top symbol on the stack.  */
/*  pop returns a 1 if the operation is successful and a 0    */
/*  if Underflow is found.                                    */

pop ()
{
   fprintf (trout, "Entered Pop routine\n");      /*  Print statements to  */
   if (stk.top == LOWERBOUND)  {                 /*  trout  trace the     */
      fprintf(stderr, "Underflow on Stack\n");   /*  stack operations     */
      return (0);
   }
   else  {
      fprintf (trout, "Poped stack char: %c at line: %d\n", stk.sym[stk.top],
               stk.lnum[stk.top]);
      stk.top--;                              /*  delete top symbol  */
      return (1);
   }
}
```

```
#include <sys/wait.h>
#include <ctype.h>
#include <strings.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define LOWERBOUND 0       /*  lowerbound on stack  */
#define UPPERBOUND 99      /*  upperbound on stack  */

/*  FILES:                                                              */
/*  input points to the C source file                                   */
/*  output points to the C source file after precompilation             */
/*  trout points to a file containing a trace of the stack operations   */

FILE *input, *output, *trout;
int lcount;                /*  count number of lines in source file  */
char line[80];             /*  buffer to read one line at a time     */

struct stack {             /*  The stack is used to search for the   */
   char sym[100];          /*  end of a statement.  When the stack   */
   int lnum[100];          /*  is empty the end of the statement has */
   int top;                /*  been found.  top points to the top of */
} stk;                     /*  the stack, sym holds the next symbol  */
                           /*  to locate, and lnum tells on which    */
                           /*  line the search began.                */

struct entry {             /*  entry allows the creation of an array */
   char ln[80];            /*  of print statements. Each statement   */
};                         /*  must be less than 80 characters.      */
```

APPENDIX A
PRECOMPILER CODE

1. Andrews, G. R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming", *Computing Surveys*, Vol. 15, No. 1, March 1983.

2. Peterson and Silberschatz, *Operating Systems Concepts*, Chapter 9 and 10, Addison–Wesley Publishing Co., Reading, Mass. 1983.

3. Rochkind, Marc J., *Advanced Unix Programming*, Prentice–Hall Inc., Englewood Cliffs. New Jersey 1985.

4. *The Balance 8000 Guide to Parallel Programming*, Sequent Computer Systems Inc., 1985.

5. *DYNIX Programmers Manual*, Sequent Computer Systems Inc., 1985.

6. *The Balance 8000 Technical Summary*, Sequent Computer systems Inc., 1985.

7. *The Balance 8000 C Compiler Users Manual*, Sequent Computer Systems Inc., 1985.

The Balance 8000 provides a complete concurrent programming environment. The programmer has all the required primitives for process creation, synchronization, termination. The programmer has the responsibility to ensure that every use of these primitives is consistent and correct. A complete investigation into the memory aspects of the Balance 8000 is needed. The creation of a message sending mechanism would also be an interesting research topic. This mechanism would, of course, be based on shared memory. The concurrent programming routines in the Parallel Programming Library are also available for Pascal. Appendix B discusses the implementation of the cobegin-coend construct in Pascal. The software community has a fair amount of research and implementation work to accomplish in order to meet the multiprocessing capabilities provided by the computer hardware community.

6 Conclusion

The objective of this paper was to investigate and document the concurrent environment of the Sequent Balance 8000 Multiprocessing System. The paper concentrates on the process creation and control mechanisms provided by the DYNIX Parallel Programming Library. A precompiler is also introduced to implement the parallel programming construct "cobegin–coend". The **fork** and **m_fork** routines are DYNIX routines for process creation. The precompiler implemented the cobegin–coend construct using the **fork**, **exit**, and **wait** routines. The **m_fork** routine may seem very limited in its capabilities, however, when used for data partitioning applications, its serves its purpose. The **m_fork** routine, like cobegin–coend, is based on the **fork** routine. The **fork** routine is a simple and flexible routine for process creation. However, the coding of **forks**, **exits**, and **waits** can become confusing. It is not a clear mechanism for denoting process creation. Each call to **fork** also requires approximately 50 milliseconds. This suggests that process creation should be limited to only those applications whose individual process run times exceed the time to execute each **fork**. Although many applications benefit from the fact that a **fork** operation copies the parents total environment to the new child process, this should not be the default and serves only to waste time and memory. What is needed is a mechanism which copies only the section of code which is to be executed by the new process and any other explicitly referenced information. The cobegin–coend construct is a very easy and clear mechanism for process creation, but loses some of the flexibility of the **fork** operation. An extension to the cobegin–coend construct is needed to add the dynamic features of the **fork** operation. A type of optional statement guard is suggested for the cobegin–coend construct. This guard could be used to determine process creation and carry a parameter which determines the number of processes to be created.

The DYNIX Parallel Programming Library also provides many routines for process synchronization. These mechanisms can be used to solve both the mutual exclusion and conditional synchronization problems. However, all these primitives are built upon the Balance's locking mechanism. This paper has already shown that this mechanism is only weakly fair. This means that no strict order is maintained on which process is next to obtain a lock, but that each process will eventually obtain the lock. The locking routines of DYNIX are very dangerous. The programmer has the responsibility for explicitly and consistently coding every **lock** and **unlock** command. This can also lead to difficulty in maintenance. A Monitor capability is suggested in which any shared resource can be encapsulated in one location along with any valid operations on that resource.

```
/*  prwaits is called at the end of a cobegin - coend block.          */
/*  The rountine inserts the code  needed to perform the wait system  */
/*  call into the updated C source file.  A loop is created in the    */
/*  new source file to perform a wait operation for each child        */
/*  process.  Within the loop, the parent receives the PID and the    */
/*  status of each child process.  If the status is nonzero, then an  */
/*  error has occurred and the array pidarray is searched to find the */
/*  number of the statement which returned with the error.  A message */
/*  is printed giving the statement number relative to the cobegin    */
/*  block.  An array of print statements (the inserted code) is       */
/*  created using the structure 'entry'.  The paramenter 'n' holds    */
/*  the number of waits (loop iterations) to perform.  'col' holds    */
/*  the column number to start the code for a structured appearance.  */

prwaits (n, col)
   int n, col;
{
   int j, i;
   static struct entry stmt[9] = { {"for (ii = 0; ii < %d; ii++)  {\n\0"},
                                   {"    pid = wait (&status);\n\0"},
                                   {"    if (status)  {\n\0"},
                                   {"       jj = 0;\n\0"},
                                   {"       while (pid != pidarray[jj])\n\0"},
                                   {"          jj++;\n\0"},
       {"       printf (\"Error on Stmt %%d in cobegin block\\n\",jj);\n\0"},
                                   {"    }\n\0"},
                                   {"}\n\0"} };

   fprintf (trout, "Prwaits is called number of Waits : %d\n", n);

   for (i = 0; i < col; i++)      /*  move to correct column to insert code  */
      putc (' ', output);

   fprintf (output, stmt[0].ln, n);    /*  insert 'for loop'  */
   for (j = 1; j < 9; j++)  {
      for (i = 0; i < col; i++)        /*  move to column 'col'  */
         putc (' ', output);
      fprintf (output, stmt[j].ln);    /*  insert wait logic  */
   }
}
```

```
/*  The routine forkstmts is called when the beginning of a */
/*  cobegin - coend block is found.  The overall function   */
/*  of this routine is to separate the statements within    */
/*  the cobegin - coend block and to call the appropriate   */
/*  routines to insert the fork, exit and wait code.  This  */
/*  routine is recursive, so that if it finds the beginning */
/*  of a cobegin - coend block, it calls itself.  forkstmts */
/*  uses the stack to find the end of a statement.  It      */
/*  initially assumes the end to be a ';' and thus places   */
/*  that character on the stack.  If a '}' is found before  */
/*  the end of the statement is found, then a '}' replaces  */
/*  ';' and will now indicate the end of the statement.     */
/*  every time a character is read that matches the top of  */
/*  the stack, then the top of the stack is deleted.  If    */
/*  the stack is empty, then the end of the statement is    */
/*  found.  A switch statement is used to check each        */
/*  character that is read. The parameter 'col' holds the   */
/*  column number where the cobegin was found. This is used */
/*  to create a structured appearance when inserting code.  */
/*  forkstmts will call the push and pop routines inside    */
/*  if statements.  This is to check for errors.  If an     */
/*  error is found, forkstmts returns a 0, otherwise it     */
/*  returns a 1.                                            */

forkstmts (col)
   int col;
{
   int nocoend;      /*  True when no coend has been found  */
   int dostmt;       /*  True when a do statement is found  */
   int notstmtend;   /*  False when the end of a statement is found  */
   int numwaits;     /*  The number of iterations for the final wait logic */
   int cont;         /*  Used to find the begining of the next statement   */
   int nxtchr;       /*  The index into the current line for the next char. */
   int j;

   numwaits = 0;
   nocoend = TRUE;

   if ((fgets (line, 80, input)) == NULL)  {     /*  read new line  */
      fprintf(stderr, "EOF found, Missing coend\n");
      return (0);
   }
   else {
      nxtchr = 0;       /*  next character is index 0  */
      lcount++;         /*  increment line count       */
   }

   while (nocoend)  {          /*  do while no coend statement is found  */
      dostmt = FALSE;
      notstmtend = TRUE;

      /*  Loop until a new statement or a coend is found     */
      /*  This loop is for skipping over blank lines and     */
      /*  finding coend statements.  The coend must be on    */
      /*  line by itself and not within a statement.  So     */
      /*  look for coend before entering end of stmt logic.  */
      cont = TRUE;
      while (cont)  {
         while ((isspace (line[nxtchr])) && (line[nxtchr] != '\n'))
            nxtchr++;          /* find first nonspace char  */

         fprintf (trout, "First char on new stmt. %c\n", line[nxtchr]);

         switch (line[nxtchr]) {
```

```
        /*  if a coend is found then insert code for wait loop  */
        /*  and leave the forkstmts routine. If a cobegin is    */
        /*  found print an error message.  Placing a cobegin    */
        /*  block between statements of a cobegin block will    */
        /*  accomplish nothing.                                 */
        case 'c':
           if (! strncmp ("coend", &line[nxtchr], 5)) {
              fprintf (trout, "coend found at line %d\n", lcount);
              nocoend = FALSE;
              prwaits (numwaits, col);  /*  insert wait logic      */
              notstmtend = FALSE;       /*  no more stmts to find  */
              if (! pop ())             /*  pop stack separater    */
                 return (0);
           }
           else if (! strncmp ("cobegin", &line[nxtchr], 7))  {
              fprintf (stderr, "Improper placement of cobegin");
              return (0);
           }
           cont = FALSE;      /*  exit loop  */
           break;

        /*  If a new line is found, read in the next line  */
        /*  if EOF is found, then print error message      */
        case '\n':
           fprintf (output, "%s", line);
           if ((fgets (line, 80, input)) == NULL) {
              fprintf (stderr, "EOF found, Missing coend\n");
              return (0);
           }
           else {
              nxtchr = 0;      /*  reset index and increment line count  */
              lcount++;
              fprintf (trout, "line %d read \n", lcount);
           }
           break;

        default:
           cont = FALSE;     /*  next statement is found  */
           break;
     }
  }

  if (nocoend) {                    /*  if no coend stmt has been found,  */
     if (! push (';', lcount))      /*  then fork off the next stmt and   */
        return (0);                 /*  increment numwaits                */
     numwaits++;
     fprintf (trout, "Numwaits is now %d\n", numwaits);
     prforks (col);
  }

  /*  This next loop finds the end of each statement by using a stack.  */
  /*  A switch statement is used to evaluate each character.  The top   */
  /*  of the stack and the next character read determines the action     */
  /*  to be taken.  The following logic is applied:                      */
  /*                                                                     */
  /*  The stack top is a ';'.  Whenever a symbol is found which          */
  /*  matches the top of the stack, pop the stack.  If the stack is      */
  /*  empty, then the end of the statement has been found.  If a '}'     */
  /*  is found and the top of the stack is a ';' then a block            */
  /*  statement is found and the end of the statement will be a '}'.     */
  /*  Therefore, pop the ';' off the stack and push the '{'.  However,   */
  /*  if the block is a do statment, then do not pop the ';'.  If a      */
  /*  newline char is found, then read the next line.  If a '"' is       */
  /*  found, push it on the stack and ignore all else until another      */
  /*  '"' is found. If a '}' is found ignore all characters except a     */
  /*  '"' until a '{' is found.  If a '(' is found and the top of the    */
  /*  stack is a ';', then ignore all other characters except '"'.       */
```

```
/*  until a ')' is found.  If another cobegin is found, then call     */
/*  forkstmts recursively to separate and fork the statments.         */

while (notstmtend)  {          /* statement end has not been found  */
   switch (line[nxtchr])  {

      /* If a d is found, check for a do loop.  The do loop */
      /* is a special case.  A do loop will be enclosed by  */
      /* the { and } symbols, but will end in a ; symbol.   */
      /* Do not pop the ; symbol off the stack.             */
      case 'd':
         fprintf (trout, "Case d at %d\n", lcount);
         if (stk.sym[stk.top] == ';')
            if ((! strncmp ("do ", &line[nxtchr], 3)) ||
               (! strncmp ("do{", &line[nxtchr], 3)))  {
               fprintf (trout, "do while found on line %d\n", lcount);
               dostmt = TRUE;
            }
         nxtchr++;   /*  get next character  */
         break,

      /*  If a c is found, then check for a new cobegin block.  */
      /*  If a new cobegin block is found, recursively call     */
      /*  forkstmts routine to process the block.               */
      case 'c':
         if (! strncmp ("cobegin", &line[nxtchr], 7))  {
            fprintf (trout, "cobegin found at line %d\n", lcount);

            if (! push ('&', lcount))   /*  push on stack separater  */
               return (0);

            if (forkstmts (nxtchr))  {  /*  call forkstmts recursively  */
               fprintf (trout, "***Returned from coend***\n");
               fprintf (trout, "Top: %d\n", stk.top);
               fprintf (trout, "Symbol: %c \n", stk.sym[stk.top]);

               /*  Returned OK so read next line  */
               if ((fgets (line, 80, input)) == NULL)  {
                  fprintf (trout, "Missing %c from line %d on stack\n",
                           stk.sym[stk.top], stk.lnum[stk.top]);
                  fprintf (stderr, "EOF found, Missing coend\n");
                  return (0);
               }
               else {
                  nxtchr = 0;  /*  reset index and increment line count */
                  lcount++;
                  fprintf (trout, "line %d read \n", lcount);
               }
            }
            else {   /*  Bad return from forkstmts routine  */
               fprintf (stderr, "Bad cobegin block \n");
               return (0);
            }
         }
         else if (! strncmp ("coend", &line[nxtchr], 5))  {
            fprintf (stderr, "coend found within statement\n");
            fprintf (stderr, "Missing End of Statement\n");
            return (0);
         }
         else nxtchr++;  /*  get next character  */
         break;

      /*  If a new line is found, read in the next line  */
      case '\n':
         fprintf (trout, "Case newline at %d\n", lcount);
         fprintf (output, "%s", line);
         if ((fgets (line, 80, input)) == NULL   }        /*  EOF? */
```

```
            fprintf (trout, "Missing %c from line %d\n",
                     stk.sym[stk.top], stk.lnum[stk.top]);
            fprintf (stderr, "EOF found, Missing coend\n");
            return (0);
         }
         else {
            nxtchr = 0;  /*  Next line read, index is 0, */
            lcount++;    /*  increment line count        */
            fprintf (trout, "line %d read \n", lcount);
         }
         break;

      /*  If ( is found and top of stack is either ; or ),  */
      /*  push the symbol ) on the stack                    */
      case '(':
         fprintf (trout, "Case ( at line %d\n", lcount);
         if ((stk.sym[stk.top] == ')') || (stk.sym[stk.top] == ';'))
            if (! push (')', lcount))
               return (0);
         nxtchr++;           /*  get next character  */
         break;

      /*  If a { is found and the top of the stack is either a ;  */
      /*  or a }, then push the { on the stack.  Pop the stack if */
      /*  the top is a ; and dostmt is FALSE.                     */
      case '{':
         fprintf (trout, "Case { at line %d\n", lcount);
         if ((stk.sym[stk.top] == ';') && (dostmt))  {
            if (! push ('}', lcount))
               return (0);
         }
         else if (stk.sym[stk.top] == ';')  {
            if (! pop ())
               return (0);
            if (! push ('}', lcount))
               return (0);
         }
         else if (stk.sym[stk.top] == '{')
            if (! push ('}', lcount))
               return (0);
         nxtchr++;      /*  get next character  */
         break;

      /*  If ) is found and top of stack is ), then pop it  */
      /*  off the stack.                                    */
      case ')':
         fprintf (trout, "Case ) at line %d\n", lcount);
         if (stk.sym[stk.top] == ')')
            if (! pop ())
               return (0);
         nxtchr++;           /*  get next character  */
         break;

      /*  If a } is found and top of stack is {, then pop the  */
      /*  stack and check if stack is empty.  If the stack is  */
      /*  empty, then insert exit code and statement end has   */
      /*  been found                                           */
      case '}':
         fprintf (trout, "Case } at line %d\n", lcount);
         if (stk.sym[stk.top] == '{')  {
            if (! pop ())
               return (0);
            if (empty ())  {      /*  empty stack?  */
               nxtchr = printexit (++nxtchr, col);    /*  insert exit */
               if (nxtchr == -1)       /* bad return */
                  return (0);
               notstmtend = FALSE;
```

```
                }
                else nxtchr++;    /*  get next character  */
            }
            else nxtchr++;        /*  get next character  */
            break;

        /*  If a ; is found and the top of the stack is a ;,  */
        /*  then pop stack and check if stack is empty.  If   */
        /*  stack is empty, insert exit code and statement    */
        /*  end is found.                                     */
        case ';':
            fprintf (trout, "Case ; at line %d\n", lcount);
            if (stk.sym[stk.top] == ';')  {
                if (! pop ())
                    return (0);
                if (empty ()) {   /*  empty stack?  */
                    nxtchr = printexit (++nxtchr, col);  /*  insert exit  */
                    if (nxtchr == -1)     /* bad return */
                        return (0);
                    notstmtend = FALSE;
                }
                else nxtchr++;  /*  get next character  */
            }
            else nxtchr++;     /*  get next character  */
            break;

        /*  If a " is found and the top of the stack is also a ",  */
        /*  then pop the stack, otherwise push the " on the stack. */
        case '\"':
            fprintf (trout, "Case \" at line %d\n", lcount);
            if (stk.sym[stk.top] == '\"')  {
                if (! pop ())
                    return (0);
            }
            else
                if (! push ('\"', lcount))
                    return (0);
            nxtchr++;      /*  get next character  */
            break;

        default:
            nxtchr++;      /*  get next character  */
            break;
        }
    }
}
return (1);
}
```

```
/*  The routine find_block is called by the main routine for every  */
/*  file entered by the user.  The main purpose of find_block is to */
/*  search the input file for a cobegin block and to call the       */
/*  routine forkstmts to process the block.  find_block opens three */
/*  files; the input source file, the output source file, and a     */
/*  a trace file. The output file contains the new C source code.   */
/*  The trace file contains a trace of the stock operations         */
/*  performed.  Given an input file name of XXX.c, find_block will  */
/*  create the output file with the name XXXp.c and the trace file  */
/*  with the name XXXt.d.  The trace file will be retained only if  */
/*  an error occurrs while processing the input file.               */

find_block (filenum, argv, argc)
   int filenum;                    /*  The number of the file in argv  */
   char *argv[];
   int argc;
{
      FILE  *fopen();
      static char trace[15] = {"    "};      /*  The name of the trace file  */
      static char temp[15] = {"  "};         /*  The name of the output file */
      int i, noerror;

      noerror = TRUE;   /*  noerror indicates an error in the input file */
      stk.top = 0;

      /*  open input file */
      if ((input = fopen (argv[filenum], "r+w")) == NULL)  {
         fprintf (stderr, "Could not open file: %s\n", argv[filenum]);
         exit (2);
      }
      else
         fprintf (stderr, "Opened input file: %s\n", argv[filenum]);

      strcpy (trace, argv[filenum]);  /*  copy name of input file  */
      strcpy (temp, argv[filenum]);   /*  to both trace and output */
      i = 0;
      while (temp[i] != '.')  i++;
      trace[i] = 't';
      temp[i] = 'p';
      trace[++i] = '.';                  /*  These instructions complete the */
      temp[i] = '.';                     /*  names of the trace and output   */
      trace[++i] = 'd';                  /*  files.                          */
      temp[i] = 'c';
      trace[++i] = '\0';
      temp[i] = '\0';

      /*  open output file  */
      if ((output = fopen (temp, "w")) == NULL)  {
         fprintf (stderr, "Could not open file: %s\n", temp);
         exit (2);
      }
      else
         fprintf (stderr, "Opened output file: %s\n", temp);

      /*  open trace file  */
      if ((trout = fopen (trace, "w")) == NULL)  {
         fprintf (stderr, "Could not open file: %s\n", trace);
         exit (2);
      }
      else {
         fprintf (stderr, "Opened Trace File  %s\n", trace);
         fprintf (trout, "Trace of Cobegin - Coend Block\n\n");
      }

      lcount = 0;
      /*  This loop will read the input file and write to the   */
```

```
    /*  output file until a cobegin block is found.  At that  */
    /*  time, the routine forkstmts is called to process the  */
    /*  cobegin block.  If the routine 'main' is found, then  */
    /*  the array 'pidarray' and the variables pid, ii, jj,   */
    /*  and status are inserted into the output file.  These  */
    /*  are used by the fork and wait code.                   */
    /*  If an error is found in the file, return a 1.         */

    while (((fgets (line, 80, input)) != NULL) && (noerror)) {
        lcount++;
        i = 0;
        while (isspace (line[i]))  i++;
        if (! strncmp ("main", &line[i], 4)) {          /*  found 'main'?  */
            fprintf (output, "int pidarray[25];\n");
            fprintf (output, "int status, pid, ii;\n");
            fprintf (output, "static int jj = {1};\n");
        }
        if (strncmp ("cobegin", &line[i], 7))          /*  found 'cobegin'?  */
            fprintf (output, "%s", line);
        else  {
            fprintf (trout, "cobegin found at line %d\n", lcount);
            push ('&', lcount);
            if (! forkstmts (i))  {       /*  process cobegin block  */
                noerror = FALSE;
                fprintf (stderr, "Bad cobegin block\n");
            }
        }
    }
    if (noerror)  {
        fprintf (trout, "EOF: %s\n", argv[filenum]);
        fprintf (stderr, "Number of lines in file %s: %d\n",
                 argv[filenum], lcount);
    }

    /*  close files  */
    if ((fclose (input)) == EOF)
        fprintf (stderr, "Could not close file: %s\n", argv[filenum]);
    if ((fclose (output)) == EOF)
        fprintf (stderr, "Could not close file: %s\n", temp);
    if ((fclose (trout)) == EOF)
        fprintf (stderr, "Could not close file: %s\n", trace);

    /*  check for errors  */
    if (noerror)   {
        unlink (trace);
        return (0);
    }
    else
        return (1);
}
```

```
/*  The main routine simply checks to see how many files were entered  */
/*  by the user and to fork a separate child to process each file.      */
/*  If the user doesn't enter a file, an error message is printed.      */
/*  The user may enter only 6 files.  This is because a user can only   */
/*  have 20 files opened and each file entered requires three opened    */
/*  files (input, output, and trace).  If only one file is entered,     */
/*  the parent will process the file and no children will be created.   */
/*  If a child has an error, then it returns the file number of the     */
/*  file it was processing and the parent will print an error message.  */

main (argc, argv)
   int argc;
   char *argv[];
{
   int i, err, pid, filenum, nochilds;
   union wait status;

   if (argc < 2)  {      /*  Did the user enter a filename?  */
      fprintf (stderr, "Must Give A File Name!\n");
      exit (1);
   }
   if (argc > 7)  {     /*  Did the user enter too many files?  */
      fprintf (stderr, "User May Only Input 6 Files!\n");
      exit (1);
   }

   nochilds = 0;      /* number of children is 0    */
   filenum = 1;       /* file numbers start with 1  */

   setbuf (stderr, NULL);  /* don't buffer output to standard error device */

   if (argc == 2)  {      /*  only one file, don't fork any children.  */
      err = find_block (filenum, argv, argc);
      if (err)
         fprintf (stderr, "Error on File %s\n", argv[filenum]);
   }
   else  {                    /*  fork a child for each file  */
      while (argc != 1) {
         if (fork () == 0) {
            err = find_block (filenum, argv, argc);
            if (err)
               exit (filenum);
            else
               exit (0);
         }
         argc--;
         filenum++;      /*  increment file number and number of children  */
         nochilds++;
      }

      for (i = 0; i < nochilds; i++) {   /*  wait for the children to finish */
         pid = wait (&status);
         if (status.w_status != 0)  {    /*  the child returned an error  */
            if (! status.w_termsig)
               fprintf (stderr, "Error on File %s\n", argv[status.w_retcode]);
            else  {
               fprintf (stderr, "Terminotec by System Error: %u\n",
                        status.w_termsig);
               if ( status.w_coredump)
                  fprintf (stderr, "Core Dump Taken\n");
            }
         }
      }
   }
}
```

APPENDIX B
PARALLEL PASCAL

Any Pascal program can be linked to the Parallel Programming Library by including the -mp option on the Pascal compiler command, **pascal**. A user must declare the routines in the Parallel Programming Library as C external procedures or functions within their Pascal program by using the keyword **cexternal**. The functionality of these routines is the same for Pascal as for C. Reference the "Balance 8000 Guide to Parallel Programming" for further information on using these routines in Pascal. However, the routines **fork**, **exit**, *and* **wait** *are not part of the Parallel Programming Library and can not be directly referenced from a Pascal program. A user must first write a function in C which performs the actual* **fork**, **exit**, or **wait** and then link this function to their Pascal program. This is very simple to accomplish. Again, the C functions are linked to the Pascal program by declaring them as **cexternal** functions or procedures. Any C function which returns an integer must be declared in Pascal as returning a long integer (**longint**). Place the C functions in a separate file and compile them using the Pascal compiler. This file must have the ".c" file extension. The **pascal** command is smart enough to call in the C compiler for this file. Also, when the C compiler compiles the C functions, it places an underscore before the name. The functions must be declared and referenced in the Pascal program using this underscore. Pascal also passes parameters in reverse order to C. Thus, if you call a C function and pass parameters A, B, and C in that order, the C function will receive the parameters in the order of C, B, and A. If any C function references a Pascal procedure, you must also include the -e option when compiling the files. When a C function returns, the calling function releases the stack. When a Pascal function returns, the called function releases the stack. The -e option ensures that the calling C function releases the stack not the called Pascal function.

The following two files demonstrate the ability to **fork** procedures in Pascal, terminate the processes (**exit**), and synchronize (**wait**). The Pascal procedure "add" adds a value to a counter in an array of counters and prints its value. This program adds different amounts to five different counters, concurrently. Before each call to the procedure "add", the procedure "_fk" is called. This procedure is a **cexternal** function which performs a **fork** and returns the process ID. The parent process will receive the new PID of the child process and the child process will receive a zero. The child process performs the "add" operation and then calls the procedure "_ext". This procedure is a **cexternal** function which performs an **exit**. At the end of the program, the parent process calls the function

"_wt" for every child process. This function is also an **cexternal** function which performs a **wait** and returns the PID of the exiting process. The first file is the Pascal program. The second file is the C functions which call **fork**, **exit**, and **wait**. The array of counters is placed in shared memory since all global variables in Pascal are shared. This example implies that every routine a C program can reference can also be used by a Pascal program. However, the user is cautioned when performing I/O. If the main program is written in Pascal, use Pascal for the I/O. This example also implies that the cobegin–coend construct can be easily extended to Pascal programs.

```
                                /**************/
                                /*  FILE 1  */
                                /**************/

/*  This program increments an array of five counters.  The counters  */
/*  are global (shared memory). The function "add" adds a value to    */
/*  a counter.  Each counter is incremented concurrently by calling   */
/*  the functions _fk, _wt, and the procedure _ext.  These are        */
/*  external C functions which perform the routines fork, exit, and   */
/*  wait.                                                             */

program counters (input, output);

   const
      size = 5;
   var
      counter : array[1..size] of integer;
      i : integer;
      result : longint;

   procedure add (num, value : integer);
      begin
         counter[num] := counter[num] + value;
         writeln ('counter ', num, ' is ', counter[num], '.')
      end.

   procedure _ext; cexternal;
   function _fk : longint; cexternal;
   function _wt : longint; cexternal;

   begin
      for i := 1 to size do
         counter[i] := 0;

      result := _fk;
      if result = 0  then
         begin
            add (1, 10);
            _ext
         end;
      result := _fk;
      if result = 0  then
         begin
            add (2, 15);
            _ext
         end;
      result := _fk;
      if result = 0  then
         begin
            add (3, 20);
            _ext
         end.
      result := _fk;
      if result = 0  then
         begin
            add (4, 25);
            _ext
         end.
      result := _fk;
      if result = 0  then
         begin
            add (5, 30);
            _ext
         end;
      for i := 1 to size do
         begin
            result := _wt.
```

```
            writeln ('Child ', result, ' Returned.')
        end;
    end.
```

```
                        /************/
                        /*  FILE 1  */
                        /************/

/*  These functions are written in C to perform calls to */
/*  the routines fork, exit, and wait.                   */

int fk ()
{
    return (fork ());
}

int ext ()
{
    exit (0);
}

int wt ()
{
    int status;
    return (wait (&status));
}
```

APPENDIX C
INTRODUCTION TO MAN 3P

NAME

intro - introduction to Parallel Programming Library

DESCRIPTION

These routines constitute the Parallel Programming Library, which supports microtasking and multitasking in C, Pascal, and FORTRAN programs. (For information on microtasking and multitasking programming models, refer to the Balance Guide to Parallel Programming.) The Parallel Programming Library is not supported under System V (att universe).

The routines decribed here include the current Parallel Programming library, /usr/lib/libpps.a, and the previous version, /usr/lib/libpp.a. The older version is retained for compatibility with earlier DYNIX releases. The routines from the current library are linked into a program by including the -lpps option in the cc or ld command line, or by including the -lpps or -mp option in the fortran or pascal command line. The routines from the old library are linked by including the -lpp option. You must not link both libraries with the same program.

For an overview of how the current Parallel Programming Library routines are used, and for sample programs and related information, refer to the Balance Guide to Parallel Programming.

LIST OF FUNCTIONS

The following routines support microtasking:

	Name	Appears on Page	Description
9	m_fork	m_fork.3p	execute a subprogram in parallel
	m_get_myid	m_get_myid.3p	return process identification
	m_get_numprocs	m_get_numprocs.3p	get number of child processes
	m_kill_procs	m_kill_procs.3p	kill child processes
	m_lock	m_lock.3p	initialize and lock a lock
	m_multi	m_single.3p	end single-process section
	m_next	m_next.3p	increment global counter
	m_park_procs	m_park_procs.3p	suspend child process execution
	m_rele_procs	m_park_procs.3p	resume child process execution
	m_set_procs	m_set_procs.3p	set number of child processes
	m_single	m_single.3p	start single-process section
	m_sync	m_sync.3p	check in at barrier
	m_unlock	m_lock.3p	unlock a lock

The following routines support multitasking:

	Name	Appears on Page	Description
9	cpus_online	cpus_online.3p	return number of CPUs on-line
	s_clock	s_lock.3p	lock a lock, return if unsuccessfu
l			
	s_init_barrier	s_wait_barrier.3p	initialize a barrier

s_init_lock	s_lock.3p	initialize a lock
s_lock	s_lock.3p	lock a lock
S_LOCK	s_lock.3p	lock a lock (C macro)
s_unlock	s_lock.3p	unlock a lock
S_UNLOCK	s_lock.3p	unlock a lock (C macro)
s_wait_barrier	s_wait_barrier.3p	wait at a barrier

The following routines support memory allocation for parallel programming. The *brk* and *sbrk* routines are available without loading the Parallel Programming library (see brk(2)), but the versions in the Parallel Programming library are necessary for compatibility with the rest of the library.

Name	Appears on Page	Description
brk	brk.3p	change private data segment size
sbrk	brk.3p	change private data segment size
shbrk	shbrk.3p	change shared data segment size
shfree	shmalloc.3p	deallocate shared data memory
shmalloc	shmalloc.3p	allocate shared data memory
shsbrk	shbrk.3p	change shared data segment size

The following routines constitute the previous version of the Parallel Programming library, /usr/lib/libpp.a, and are retained for compatibility with earlier releases:

Name	Appears on Page	Description
p_cpus_online	p_cpus_online.3p	get number of processors in system
p_finit_barrier	p_wait_barrier.3p	initialize a barrier (FORTRAN)
p_init	p_init.3p	initialize shared memory and Atomic Lock Memory
p_init_barrier	p_wait_barrier.3p	initialize a barrier
p_init_lock	p_lock.3p	initialize a lock
p_lock	p_lock.3p	lock a lock
p_shmalloc	p_shmalloc.3p	allocate shared memory
p_unlock	p_lock.3p	unlock a lock
p_wait_barrier	p_wait_barrier.3p	wait at a barrier

The following routines are retained in the old Parallel Programming Library for compatibility with earlier releases, but are not described elsewhere in these man pages:

p_exit is equivalent to exit(3).

p_fexit is equivalent to the standard FORTRAN routine fhalt.

p_finit has no effect.