DCS: A System for Distributed Computing Support

Paul K. Harter, Jr.
Paul Maybee

CU-CS-309-85

DCS: A SYSTEM FOR DISTRIBUTED COMPUTING SUPPORT

by

Paul K. Harter, Jr. and Paul Maybee

CU-CS-309-85 September, 1985

University of Colorado, Department of Computer Science, Boulder, Colorado.

DCS: A System for Distributed Computing Support

Paul K. Harter, Jr.
Paul Maybee

1. Introduction

As inexpensive workstations become increasingly available, the personal computer is rapidly replacing the simple CRT terminal connected to a large mainframe. At the same time, users continue to require the (occasional) use of expensive and/or noisy I/O peripherals or large file storage. Thus, the central system is often replaced by a communication network for resource sharing, with server nodes handling shared resources. Since each workstation is dedicated to an individual who most likely spends most of his or her time typing (eg. word processing, program editing), these workstations are grossly under-utilized most of the time. The DCS system is an experimental system designed to investigate techniques to harness this surplus power to do useful computing.

The concurrent availability of a number of under-utilized workstations opens up many new possibilities if the physical parallelism inherent in separate CPU's can be exploited. The nature of the environment, however, places constraints on the classes of algorithms and levels of parallelism that can make effective use of the available computing power. Though the argument has been made [Spector 82] that highly optimized micro-coded implementations can make remote operations on the level of individual memory references cost effective, we don't believe that this will be reasonable in most environments in the short term. Thus, we feel that the relatively high cost of communication between machines [Popek 81, Peterson 79] when compared to memory access and instruction execution within a machine, dictates that use of a local area network for distributed parallel computation be restricted to algorithms whose demand for cycles greatly exceeds their need for interprocess communication. Schnabel [Schnabel 85] is investigating a class of numerical algorithms that appear ideally suited for this environment.

There is another constraint on the use of workstations in the environment described above. As mentioned earlier, workstations are dedicated to individuals, who will typically fight fiercely to protect their recently won autonomy from the threat of encroachment from without. At long last free of having to suffer at the hands of others' **troff** jobs and huge **make**s, the workstation owner will not tolerate slowed response. If we are to make use of other workstations we must do so without incurring the wrath of their owners. Thus we have the additional constraint that the use of workstations belonging to other users should be completely non-intrusive. That is, the owner/user of a workstation should experience neither excessive performance degradation nor any loss of functionality.

This paper describes the design of an experimental system that allows the application programmer to make use of multiple workstations, subject to the constraints mentioned above. The DCS system supports this use by providing a high-level interface to the applications programmer that allows him or her to write multiprocess programs that are automatically distributed across a network. The DCS system manages the distribution of program text, arguments and return values, and supplies simulated globally shared variables. DCS is experimental in the sense that it is the laboratory or test bench for our research into practical aspects of supporting distributed computations.

Our current investigations can be classified into two areas: process and load distribution, and the application of update disciplines to the use of distributed shared variables. In our context, load distribution involves the choice of particular workstations as the sites on which to run particular computations. We have two goals to satisfy here, maximally efficient use of excess cycles, and continued autonomy of individual workstations. One aspect of the decision rule will clearly involve the *load average* statistic, though we believe we can make use of other information as well. Examples might be: owner input on acceptable load, recent history of commands execution by the owner and the type of load they represent, historical information on the prior behavior of the computation being scheduled, or "hints" from the programmer of the computation.

Distributed shared variables are a natural extension of global shared memory in multiprocess programs into the distributed domain. While variables in the former case are instantiated only once, it is much more efficient to have multiple copies of shared data in a distributed environment. This leads to the inevitable synchronization and update problems that are well known from distributed data base technology. The introduction of extensive synchronization mechanisms into distributed computations will lead inevitably to excessive coupling of the processes on the various nodes of the network, and the concomitant loss in overall efficiency.

Medusa [Ousterhout 80] addressed this problem by introducing "co-scheduling" for processes of a computation running on separate processing elements to ensure that processes would run at the same time and would not incur large communication delays. Co-scheduling across nodes is infeasible in our case, since each workstation will have its own autonomous kernel and scheduler. Thus, our approach is to investigate techniques for allowing concurrent updates and insuring that multiple copies of shared data will eventually have the same value. One example of such a use occurs quite naturally in large network optimization problems. A number of processes are put into execution on portions of the search space and share only one variable, which represents the cost of the cheapest configuration found to date. Each process checks its own progress against this value and decides to terminate if it is unlikely to be able to do better. Separate copies of this "lowest cost" variable may be modified independently and the new values broadcast asynchronously to the other nodes for subsequent installation. New values must be merged upon receipt in such a way as not to lose information. In this case, values may be merged using the rule that *a new value may only replace a higher old value.* This ensures that the desired information will not be lost. A late update will result in a little extra work by the effected process(es). This is not tragic.

The above scenario may be generalized to distributed shared variables, whose modification is governed by update disciplines. An update discipline is a "goodness" rule that determines which of two values is "better", to allow for a decision as to which may replace the other. As long as the update discipline imposes a total ordering on possible values, the values of distributed copies are guaranteed to converge eventually. Algorithms making use of such variables must be designed to tolerate this "eventual" convergence perhaps in ways similar to the example above, i.e. by having some process(es) do a little extra work. The design of such algorithms is a topic of current research [Schnabel 85].

Finally, the designs of both the user interface and the system architecture were the result of some compromise. We had two goals in building DCS. We wanted to develop a system within which we could carry out experiments on scheduling and the use of distributed shared variables, using real algorithms designed to solve real problems. This required that we develop an interface that would be quickly learned and usable by programmers with relatively little sophistication in systems programming. This would enable us to convince some of our user community to attempt to write programs using DCS. Our second goal was to develop a system that depended only on available operating system and language support. This has the advantage of easing ports to other machines. Specifically, we wanted to base our system on a network of SUN workstations running a version of Berkeley Unix 4.2 and the **C** programming language, making few or no modifications to either.

Our second goal fit in well with the resources at our disposal, as we didn't have the manpower to rewrite the **C** compiler or make major modifications to Unix. However, it also required that we introduce some complexity in both the interface and the implementation to overcome deficiencies in the underlying system. Specifically, the relative primitiveness of the macro facility supported by the standard pre-processor caused us to make several, otherwise undesirable, design choices. In the end, we did implement an extension to Unix to support shared memory [Harter 85c], since we felt it was necessary for good performance. We expand on these details in Section 3.

The remainder of this paper is organized as follows. The next section gives an overview of DCS functionality and describes the user interface in detail. Section 3 accounts for the bulk of the paper. It describes the architecture of DCS and discusses the motivation for many of the decisions made during the design phase. Section 4 contains some preliminary timing results based on a simulated computationally intensive task, while Section 5 contains a somewhat larger example with annotations demonstrating some of the capabilities of DCS. Section 6 discusses the relation of DCS to some other work in the area and

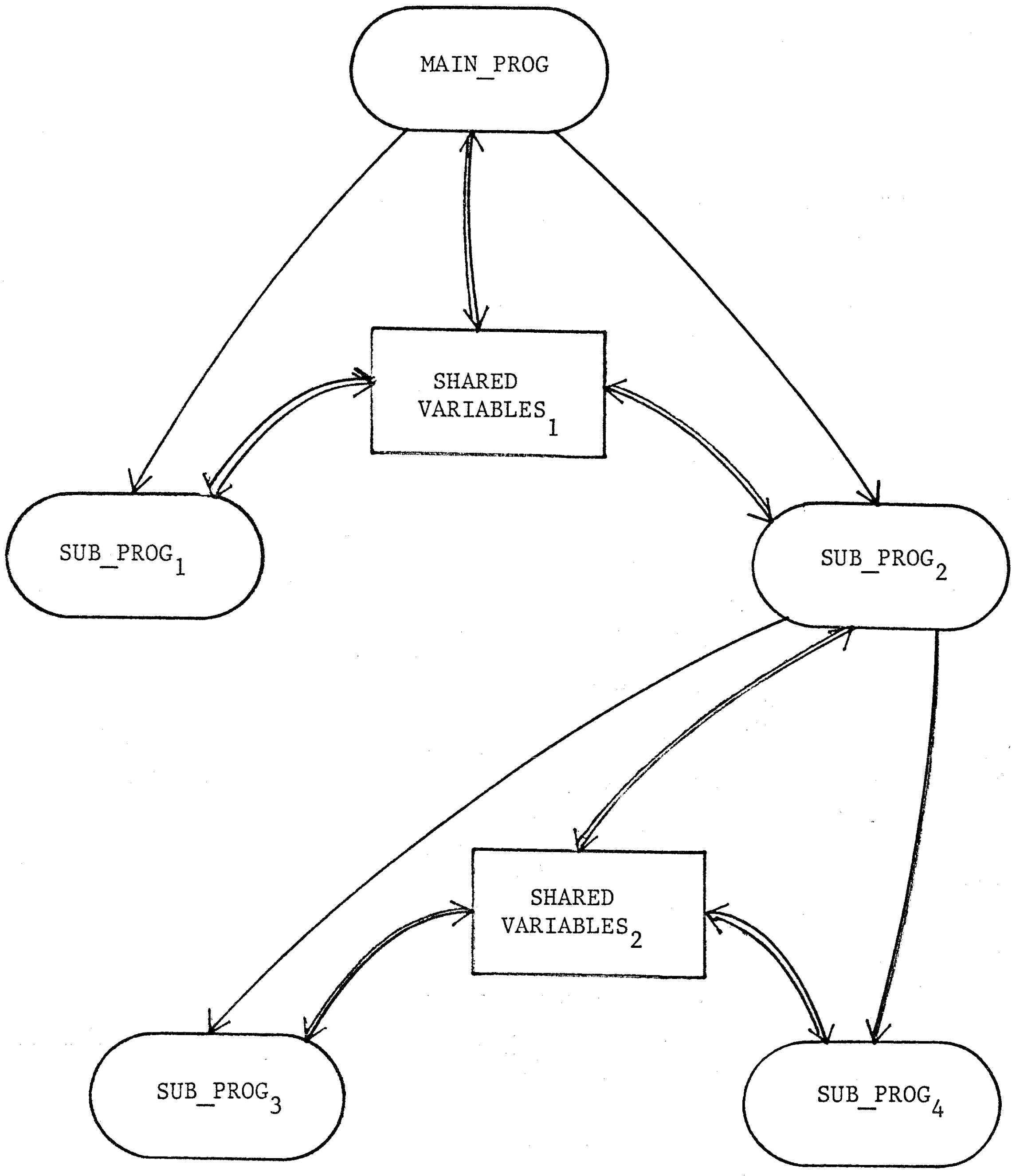

Figure 1 - Programmer View

finally, Section 7 concludes with some observations on where we are and now and what remains to be done.

2. User Interface

The DCS system implements a program environment supporting multiprocess programs executing in a network of personal workstations. The underlying model of computation (see Figure 1) used by the DCS programmer is that a computation comprises a main program that may call sub-programs, each of which executes in parallel with the main program and with each other. Sub-programs may, in turn, invoke other sub-programs or copies of themselves to execute in parallel. After invoking sub-programs to execute in parallel, a calling program may subsequently resynchronize with its subprograms by delaying itself until they terminate. The model currently contains no explicit mention of separate nodes or a communications network.

Analogous to the sequential case, where sub-programs may access variables defined in surrounding (global) contexts, the sub-programs in a DCS computation may access "global variables" contained in *shared segments*, as agreed upon by the main- and sub-programs. These shared segments are identified within the computation by *seg_id*'s which allow multiple segments of the computation to be distinguished from one another. Thus, a main program may invoke several copies of two different sub-programs and may share a different segment with each group.

2.1. Functional Overview

The model described above is based in a collection of macros and systems routines, which support two new abstractions. The user abstractions provided by DCS are the asynchronous remote procedure call and distributed shared variable.

An asynchronous remote procedure call is like a normal procedure call in that input and output arguments are passed and returned on termination of the procedure. However, whereas in the case of the synchronous call the caller is suspended during execution of the procedure, the synchronization semantics of the asynchronous call allow the caller to continue execution after making the call. The caller may then either continue to compute or make more asynchronous calls. The procedure may or may not actually execute remotely, so the programmer may make no assumptions as to the execution site, though programmer input on location may be investigated in the future.

The asynchronous execution of called procedures has implications with regard to parameter passing. If two procedures are called, where the output parameter of one overlaps the input parameter of the other, then timing could affect the arguments passed to a procedure. To prevent this we guarantee that the output arguments of a procedure are not returned to the caller until explicitly requested via resynchronization. Resynchronization of the caller and the callee occurs at the caller's discretion, when he requests that he be suspended pending completion of some subset of the currently outstanding calls. After resynchronization, output values and statuses of the waited-for procedures are available.

In the case of the synchronous procedure call, the called procedure may access global data declared in some surrounding scope. The utility of this type of access prompted us to attempt to provide a similar type of access in the asynchronous case. From a programmer's point of view, a distributed shared variable, independent of its size, may be read, written, or updated atomically with respect to other processes. That is, a process can read the value of even a large structure, with the assurance that no other process can change it during the read. The update supported is restricted to an atomic read--compute value--write paradigm, that is, an update has the form:

$$var := user\_function(var, arg),$$

with the constraint that *arg* may not contain a direct reference to another shared variable.

Modifications to variables are controlled by an *update discipline.* These can be thought of as part of the type of a shared variable. The update discipline is used to determine whether the value to be written into the shared variable should replace the value currently there. The update discipline is imposed whether the attempted write is from a user process attempting a modification or from a DCS support process (see Section 3) attempting to keep the value consistent with other nodes. Thus, although the programmer must be aware of the fact that the variable may change between subsequent reads by a single process, the variable is always in a consistent state, independent of its size or the locations of sharing processes.

The user of the DCS abstractions need not be concerned with networks, addresses, ports (or sockets), or the extra system interfaces for sending and receiving messages. Argument passing and the maintenance of shared variables across the network are handled automatically and invisibly in conformance to the semantics described above.

2.2. User Interface Routines

The DCS user interface is provided to standard **C** programs via an **#include** file (dcs_user.h), which supplies some manifest constants, macro definitions and procedure definitions for DCS support procedures. In addition, a user library (dcslib) must be specified to the loader to include the run-time support routines that provide the interface to the DCS system processes. This section gives an informal description of the routines involved in the user interface to DCS. The Unix manual pages for these routines are included as an Appendix.

The facilities provided by DCS may be divided into two relatively independent sets, shared memory management, and procedure invocation. Asynchronous procedure invocation and parameter passing are supported via the routines **doprog, parwait, raccept** and **rreturn.** The use of distributed shared memory is supported via **openseg, sharedvar, sharedvec,** and **closeseg** for declaring variables in shared segments, and **aread, agetelt, awrite, aputelt, aupdate,** and **aupdtelt** for reading, writing, and updating whole variables or vector elements respectively. In other words, **aread** will read the contents of an entire shared variable or entire shared vector, while **agetelt** will read the value of a specified element of a shared vector. The vector element routines differ from their whole variable counterparts only in that they reference vector elements, indeed their presence is simply an artifact of the lack of string processing capability within the **C** pre-processor. Hence we will ignore the distinction in the sequel and couch the discussion in terms of **aread, awrite** and **aupdate.**

2.2.1. Asynchronous Invocation

A distributed computation begins with the execution of a "main" program, which then requests the asynchronous execution of one or more "sub-programs" via calls to the **doprog** procedure. Each call specifies a file name containing an executable program, a place to put the subprogram's termination status, a data block of values to be passed to the subprogram, and a location to put the results of the subprogram's computation. Each call to **doprog** returns an identifier (or *prog_id*), for the subprogram started, which may later be used to refer to the subprogram.

The main program may re-synchronize with a set of subprograms executing in parallel via a call to the routine **parwait.** If a subprogram is to return values in the return block, or if the main program wishes either to wait for the termination or know the termination status of a subprogram, then a call to **parwait** is issued giving the identifier previously returned by the corresponding call to **doprog.** The termination status and any values returned from the subprogram in the return argument area specified in the call to **doprog** are available only after the call to **parwait** returns. This prevents the conflict on arguments mentioned above, and gives the programmer control over when his local data is modified. Note that a single call to **parwait** may specify that the main routine is to be delayed until the completion of any subset of the the currently executing subprograms specified by a list of their *prog_id*'s, and that the call to **parwait** does not return until all of the sub-programs mentioned in the call have terminated.

The **doprog** procedure causes the DCS system to register the request for execution and queue the request and parameter information. The DCS support system will then schedule the sub-program for execution on one or more of the nodes in the network. If the file named in the **doprog** call does not exist on the node on which the program is scheduled to execute, it will copy the file from the site on which the entire computation was started to a temporary directory and execute it from there. Any number of sub-programs (up to system limitations) may be started using repeated invocations of **doprog.**

Thus, for example, the effect of a simple **cobegin-coend**, where the statements to be executed in parallel are all procedure calls may be obtained by a series of calls to **doprog** followed by a call to **parwait** specifying a wait on all of the subprograms as follows

$\text{prog\_id}_0 = \text{doprog}(\text{fname}_0, \text{status}_0, \text{arg}_0, \text{sizeof}(\text{arg}_0), \text{ret}_0, \text{sizeof}(\text{ret}_0));$
$\text{prog\_id}_1 = \text{doprog}(\text{fname}_1, \text{status}_1, \text{arg}_1, \text{sizeof}(\text{arg}_1), \text{ret}_1, \text{sizeof}(\text{ret}_1));$
...
$\text{prog\_id}_n = \text{doprog}(\text{fname}_n, \text{status}_n, \text{arg}_n, \text{sizeof}(\text{arg}_n), \text{ret}_n, \text{sizeof}(\text{ret}_n));$
$\text{status} = \text{parwait}(\text{id}_0, \text{id}_1, ..., \text{id}_n, 0).$

Here, the names, $fname_0$, $fname_1$, ..., $fname_n$ name (possibly different) files containing an executable program. The $status_i$ arguments will eventually receive the termination status' of the respective programs. Finally, the contents of *arg* are copied by the call to **doprog** and passed to the sub-program when it executes, and the addresses of the *ret* and *status* parameters are kept as the locations into which to place the results returned by the subprogram and its completion status respectively.

Procedures that are to execute asynchronously as part of a DCS computation must be compiled into independently invokable programs. That is, the procedure must be named "main" or be explicitly mentioned as the entry point of the module, since they are invoked via the system **exec** call. No "command line" arguments are supplied to the sub-program on invocation, so parameter passing to and from the sub-program is handled via the calls **raccept** and **rreturn**. The subprogram receives initial values from the main program via the **raccept** call. The type of the parameter to the call must be the same as the type of the argument sent by the main program. The subprogram returns results to the main program and terminates using the **rreturn** call. The parameter to this routine and the return parameter on the **doprog** must also have the same type. Upon returning its results, the subprogram terminates, and the results and program termination status are then made available to the main program, when it issues a **parwait** referencing the *prog_id* of the terminating sub-program. Note that the termination status is returned whether the subprogram calls **rreturn** or not, and that at most one call each to **raccept** and **rreturn** is allowed in a subprogram.

2.2.2. Shared Variables

In addition to the ability to pass parameters in and out of the asynchronous procedures, interprocess communication is possible via a form of distributed shared variables. The programs of a distributed computation wishing to use shared variables use four macros supplied by DCS to make the necessary local declarations, lay out the storage of the shared segment, and perform the initializations required to permit subsequent access. First the program calls **openseg**, specifying a name for the shared segment, as in

openseg (SEG1);

Variables may now be placed in this shared segment using the macros **sharedvar** and **sharedvec** for declaring variables and vectors respectively. For each variable, the program specifies a name, a type and an update discipline, and for **sharedvec**, a length. The type may be any valid **C** type, with the restriction that **struct** and **union** types must be specified using their *tag*'s and can not be defined within the macro call. For example,

sharedvar (char, done, UPDT_UNREST);
sharedvec (double, x, SIZE, UPDT_UNREST);

declare a shared character variable called "done" and a vector of double precision variables of length SIZE called x, both using the "unrestricted" update discipline, which places no constraints on value replacement. This is advisable when it is known that processes will not conflict on writing the same variable. Other update disciplines are described below. Finally, the segment is ended by a call to **closeseg** as in

closeseg (SEG1)

specifying the same segment identifier (SEG1) as in the **openseg** call.

The calls describing a shared segment must appear in the declarations section of the "main" routine in the (sub-) program, i.e. before any executable statements. A subprogram declares its shared segments in exactly the same manner as a main program. To share a segment, the sharing processes must be part of the same computation and must declare the segment in exactly the same way. That is, the name of the segment, the types of all variables, and the order in which they were declared within the **openseg-closeseg** must all be identical. Typically, both would **#include** the segment declaration from the same file, thus ensuring agreement on segment identifiers and variable types and layout. Multiple shared segments may be declared by specifying different names in successive **openseg** calls. Declarations of shared segments may not be nested.

DCS supplies three procedures for accessing the variables in shared memory. The **aread** routine allows the user program to read the contents of a shared variable into local (private) storage without being interrupted by another program independent of variable size. For example,

stat = **aread** (x, &my_x);

copies the current value of the shared variable x into the location of my_x. The return code may indicate an error condition, for example, that x was never initialized. The **awrite** routine similarly provides exclusive access while writing a variable from local (private) storage, as in stat = **awrite** (x, &my_x);

which copies the current value of my_x in to x, subject to its update discipline. Here the return code may indicate whether the write was actually performed or whether the old value was retained. Sometimes exclusive access more complex than a simple read or write is needed. The **aupdate** routine provides exclusive access to a shared variable for the duration of a call to a user function that may read, compute with and modify the variable. Thus, **aupdate** provides the ability to modify a variable based on its current value without losing an update. For example,

stat = **aupdate** (updt_func, x, &updt_arg);

which passes the value of x and the address of updt_arg to updt_func. The resulting new value for x is written into the shared variable subject to its update discipline. The only safe access to shared variables is through the use of these routines, attempts at direct access will give undefined results.

2.2.3. Update Disciplines

True shared variables have semantics specified by the architecture of the underlying machine and operating system. Distributed shared variables as supported by the DCS may have certain aspects of their behavior specified at declaration time. This specification takes the form of an update discipline supplied for each variable, which allows for specific digressions from the semantics of a true shared variable. The discipline specified affects the manner in which the access associated with **aread, awrite**, and **aupdate** affect the distributed shared variable.

The following is a sample list of update disciplines. All disciplines will be available from the system. It will be easy to include additional disciplines at a later time.

1. Global Update - all copies of this variable are identical.
2. Integer Max - only update if new value is larger than old value.
3. Floating Max - same as above, with floating variables.
4. Integer Min - only update if new value is smaller than old value.
5. Floating Min - same as above, with floating variables.
6. Unrestrained Update - new values may always supercede old values.
7. Unsafe Update - no synchronization, no checking of values.

The *global update* discipline is the most restrictive, it insists that the distributed shared variable behave as a real shared variable would. That is, all copies of the variable have the same value at all times.

Disciplines 2 through 5 can all work on integer or floating variables or vectors of these variables treated component-wise. These disciplines specify some relationship between current values and allowable new values. An update is done under one of the disciplines if the value to be written and the value currently in the variable satisfy the given relation (e.g. a floating min-so-far variable would use discipline 5, and new values would only be written if they were less then the current value). No global synchronization is implied.

Discipline 6 specifies that new variables can be written into the shared memory whenever the process interested in writing them can get exclusive local access, no check on values is performed. Discipline 7 doesn't require any synchronization at all, i.e. no local or global exclusive access. All disciplines which do not need global synchronization simply forward the modifications for installation in the other copies of the shared memory.

If a discipline which requires some relationship between old and new values (e.g. 2 through 5) is in effect for a variable, there must be some way to provide that variable an initial value. This is achieved by having DCS automatically accept the first write to such a variable, using either **awrite** or **aupdate**, without applying the test implied by the update discipline. The **aread** routine will return a "no value" status if a variable to be read has not been initialized. Further, the update routine specified in an **aupdate** call will be notified if the variable to be updated has not been initialized by an additional flag parameter.

3. System Architecture

The functionality described above is supported at the user level by macros included in the compilation of a DCS computation and library routines that are loaded from dcslib. Thus, calls to the user interface routines described in the previous section all result either directly or indirectly through macros to invocations of DCS support routines in user process space. These routines are divided conceptually into three modules. The Remote Service Module (RSM) provides the **doprog** and **parwait** services allowing a program to request and wait for remote execution. The Remote Execution Module (REM) provides the **raccept** and **rreturn** routines used by a subprogram to accept and return its parameters. Lastly, the Shared Memory Module (SMM) supplies the interface to declare and reference shared variables. This includes **openseg, sharedvar, sharedvec,** and **closeseg** for declaring segments containing shared variables, and **aread, awrite,** and **aupdate** for referencing them. In addition to the routines already mentioned, these modules contain private variables and tables for implementing the services they provide.

These routines in turn are based on services provided by two DCS processes running in the background (i.e. system "daemon" processes) on each node of the network (see Figure 2). These services are requested and received via messages exchanged between the support modules and the DCS daemon processes. The Remote Scheduling Processes (RSP's) provide for local and remote queuing and execution as well as parameter exchange. The Distributed

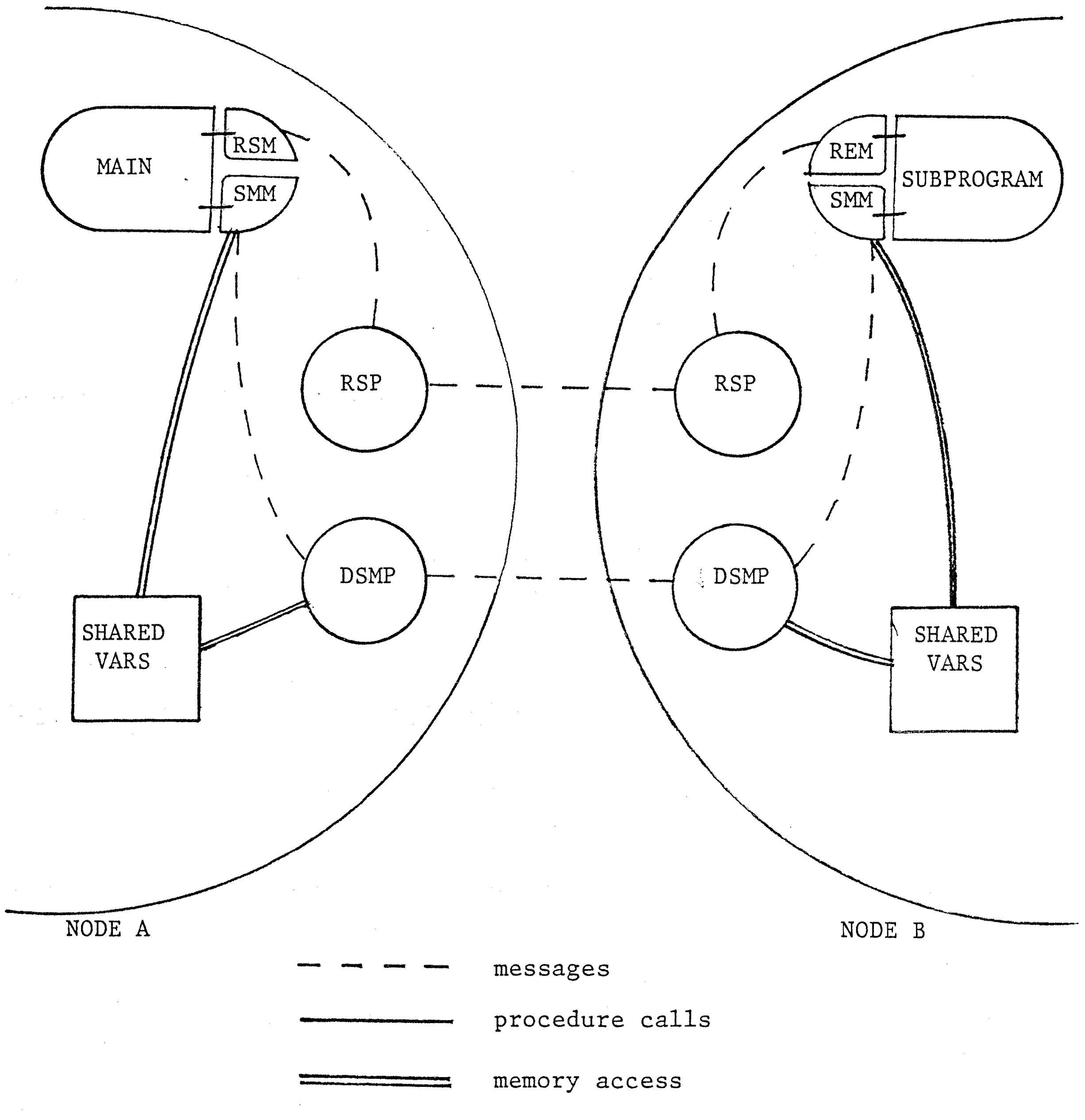

Figure 2 - DCS Architecture

Shared Memory Processes (DSMP's) maintain consistency of the various copies of shared variables across the network nodes. The DCS processes on the various nodes exchange messages for internal coordination.

There are several levels of communication involved in a DCS computation. User programs communicate with the DCS service modules (RSM, REM, SMM) via normal procedure calls. The DCS routines in these modules execute in user space and have access to user locations directly or indirectly through parameters. The service module routines also communicate with the DCS daemon processes (RSP, DSMP) using messages passed via sockets. The DCS support processes communicate as well, the RSP's to transfer invocation requests and arguments, the DSMP's to create and maintain segments of shared variables.

There is a fourth, implicit, channel of communication that exists between user programs and other user programs as well as between user programs and the DSMP's. This channel results from the sharing of physical memory pages within a node. This additional channel was added to increase the efficiency of use of our shared variables.

The distributed shared variables associated with a computation must be equally accessible to all processes of that computation independent of their physical site of execution. This desire for equal, efficient access excluded an implementation where processes obtained and assigned values by sending messages to a special storage or caretaker node. Thus, a copy of each variable is kept at each node. Further, the various copies of a shared variable must be kept consistent across nodes. This is ensured in DCS by the DSMP's, one on each node. The DSMP's on the various nodes communicate update information to each other when values are changed by user programs and guarantee adherence to the update discipline specified for each variable. Thus, the DSMP must also be able to access the shared data.

Again, our desire for efficiency caused us to reject a solution where the DSMP "owned" the copy of the shared variable on each node. This implementation would have the DSMP responding to requests for the current values of variables as well as requests for updates. Particularly in the case of multiple concurrent computations or multiple processes of a single computation on one node, the DSMP could present a performance bottleneck. Thus, to avoid the cost of sending messages within a node and the bottleneck effect of having one process do so much work, it is necessary for processes to be able to access the same set of memory locations.

As this feature was not available in Berkeley Unix and we believed it might be useful to others as well, we designed and implemented the a shared memory extension to Unix. The design and implementation of the our extension are discussed in detail elsewhere [Harter 85c]. We give here only a brief overview, in order to make the following discussion of the design of DCS understandable. Our extension for shared memory involved the addition of 5 new system calls to Unix, **vshare, vrlse, getsem, Psem** and **Vsem.** In order for two processes to share physical memory, both must call the routine **vshare** to attach to a shared segment. The first caller specifies a kernel segment identifier (*kern_seg_id*) value of 0, and receives the actual identifier for the new segment allocated by the kernel. The second caller, must specify the *kern_seg_id* returned from the first call to attach to the same segment. Each call must specify the segment size and a starting address for mapping the shared segment into the callers address space. The shared segment may be subsequently released by calling **vrlse** specifying the *kern_seg_id.*

The DCS interface guarantees that any access to shared variables is atomic, so that variables are always in a consistent state. There is no problem with this in communication between nodes to keep variables up to date, since the protocols have been designed to send and install shared variables in their entirety. Synchronization becomes important in preserving atomicity within a node, where the DSMP and one or more user processes all share a single copy of a shared variable. Thus, we needed to implement some form of mutual exclusion, so that access by both user processes and the DSMP would not conflict.

The routines **getsem, Psem** and **Vsem** implement a semaphore-like synchronization facility for processes sharing memory. Semaphores are created and initialized by calls to **getsem**, which returns the *sem_id* for the newly created semaphore. This *sem_id* is used to refer to the semaphore in subsequent calls to **Psem**, and **Vsem**, used to decrement or increment its value atomically.

The standard semantics of semaphores [Dijkstra 68a], however, is inadequate for synchronizing processes within DCS. Since the DSMP must acquire exclusive access to install changes from other nodes in the system, it would have to execute a **P** on a semaphore that could be held by a user process. Since the user process may fail or terminate while holding the semaphore, the DSMP would be stuck forever. Even barring this error case, a user process could hold the semaphore over a page fault or disk read. Since the DSMP may be serving many processes and many shared variables, this wasted time could lead to significant performance degradation. This led us to the design of a semaphore that could be used in the face of performance constraints. The semaphores we included provide a timeout period so that a **Psem** operation will return after either the semaphore has been decremented or the timeout period has passed. The timeout period can have zero (for polling), infinite or some finite length.

3.1. User Support Modules

As mentioned above, the user support modules form the interface between the user program and the DCS daemon processes. When the user requests a procedure invocation by calling the **doprog** procedure in the Remote Support Module RSM, **doprog** registers the request in a local table, where it also stores the locations into which to place the termination status and return values. It then communicates with the Remote Scheduling Process (RSP) on the same node to request that a subprogram be scheduled for execution. The RSP receives the file name containing the program text, and the values of any input arguments. The RSP will queue the request, and in cooperation with its cohorts on other nodes, will eventually schedule the program for execution on some node in the network. Upon receipt of an acknowledgement from the RSP, the **doprog** call returns a success value. If there is no acknowledgement, then an error code is returned.

Once a site is chosen for the execution of the sub-program, the RSP at that site starts the program executing as a user process. The sub-program receives its argument (*arg*) from the main program, using a call to the **raccept** procedure in the Remote Execution Module (REM) which has been linked into the sub-program. The **raccept** procedure actually gets the argument by sending a **getarg** message to the local RSP who responds with the values in a **sendarg** message. Thus, the argument value is received by the **raccept** routine in the REM and returned to the user when **raccept** returns.

When the subprogram has finished its computation, it may call the **rreturn** procedure of its REM to return its results (*ret*). This procedure returns the results to the local RSP in a **retarg** message, and then terminates the subprogram. Since the local RSP is the parent of the subprogram, it will receive the termination status of the sub-program. The RSP then notifies the caller's RSP of the remote termination in a **remterm** message containing the return values and termination status. The main program requests this information from its RSP in a call to the **parwait** routine in the RSM specifying the *prog_id* of the subprogram as returned from **doprog**. The **parwait** routine reads its parameters and sends the identities of the referenced sub-programs to the RSP in a **waiting_on** message. It then waits until it has received a **subterm** message from the RSP for each referenced sub-program. These messages contain the termination status and return values of the corresponding sub-program. As each **subterm** is received, the return information is copied into the locations in the caller as specified in the corresponding **doprog** call.

Programs in DCS computations make use of shared variables by creating and interacting with shared segments via the interface supplied by the Shared Memory Module (SMM). The segment is created via the macros **openseg, sharedvar, sharedvec,** and **closeseg**, which must be used in the declarations section of the "main" routine of the (sub-) program. Each of these macros creates a local variable in the stack space of the main routine, which is initialized by calling a segment mapping routine in the SMM. The local variables are initialized with pointers into the shared segment, and have the names specified by the programmer for the shared variables. These names are subsequently used by the programmer in the macros for referencing the shared data and appear to the user to contain the actual data. The mapping from these local variables to the shared variables they represent is described below.

To create the segment, the program first executes the **openseg** macro with the computation specific name for that segment (seg_id). The macro invokes the **open_seg** routine in the SMM, which creates and initializes an entry in its table for storing such information as the size and starting address of the new segment in user space. It also expands the data segment of the program by one page, which will eventually become be the first page of the shared segment. The user program then calls the declaration macros **sharedvar** and **sharedvec** to place variables in the segment. These invoke the routine **smm_vmap** in the SMM, which allocates and initializes a variable descriptor for the new variable in the portion of the data segment that will become shared, expanding the data segment by another page as necessary. It then returns a pointer to this variable descriptor which is used to initialize the user's local variable. When the **closeseg** macro is called to complete the declaration of the current segment, it calls the **close_seg** procedure in the SMM to finish processing and make the segment sharable. This involves a number of steps. First, additional space is added to the data segment and allocated to hold the variables that will occupy the segment. For shared vectors, space is also allocated for run-time dope vectors, whose use will be described below. The next step is to make the segment sharable by calling **vshare.** If the segment being created already exists on the current node due to another (sub-) program of the current computation, then the call to **vshare** must specify the correct identifier (*kern_seg_id*) to insure that both programs will share the same memory. Thus, the **close_seg** procedure sends a message to the DSMP that includes the process id (pid) of the declaring process and the *seg_id* for the segment being declared. The DSMP will return a *kern_seg_id* for **vshare.** If the value returned is 0, then the segment did not exist previously and this value will make the segment sharable. The SMM must then then inform the DSMP of the *kern_seg_id* returned from **vshare** so that it may then rendezvous on the newly sharable segment.

The SMM controls user access to the variables in a shared segment, whereas the DSMP maintains the consistency of the different copies. Therefore some communication between SMM and DSMP is necessary to coordinate updates to the shared memory. Each variable declared in a shared segment is described by a small control block or descriptor in the shared segment (see Figure 3). This descriptor contains the offset into the shared segment of the shared variable, its size, update discipline, and the *sem_id* of the semaphore used for coordination of processes on the same node and flags indicating whether the variable is a vector, has been modified, initialized, or is globally locked. For vectors, the descriptor also contains the offset into the segment of a dope vector. The descriptor is used by the SMM and the DSMP to coordinate and govern access to the variable.

As seen from the figure, the variable allocated in private user space for a shared variable contains a pointer to either the descriptor or a dope vector. The dope vector contains a pointer to the descriptor and one element for each

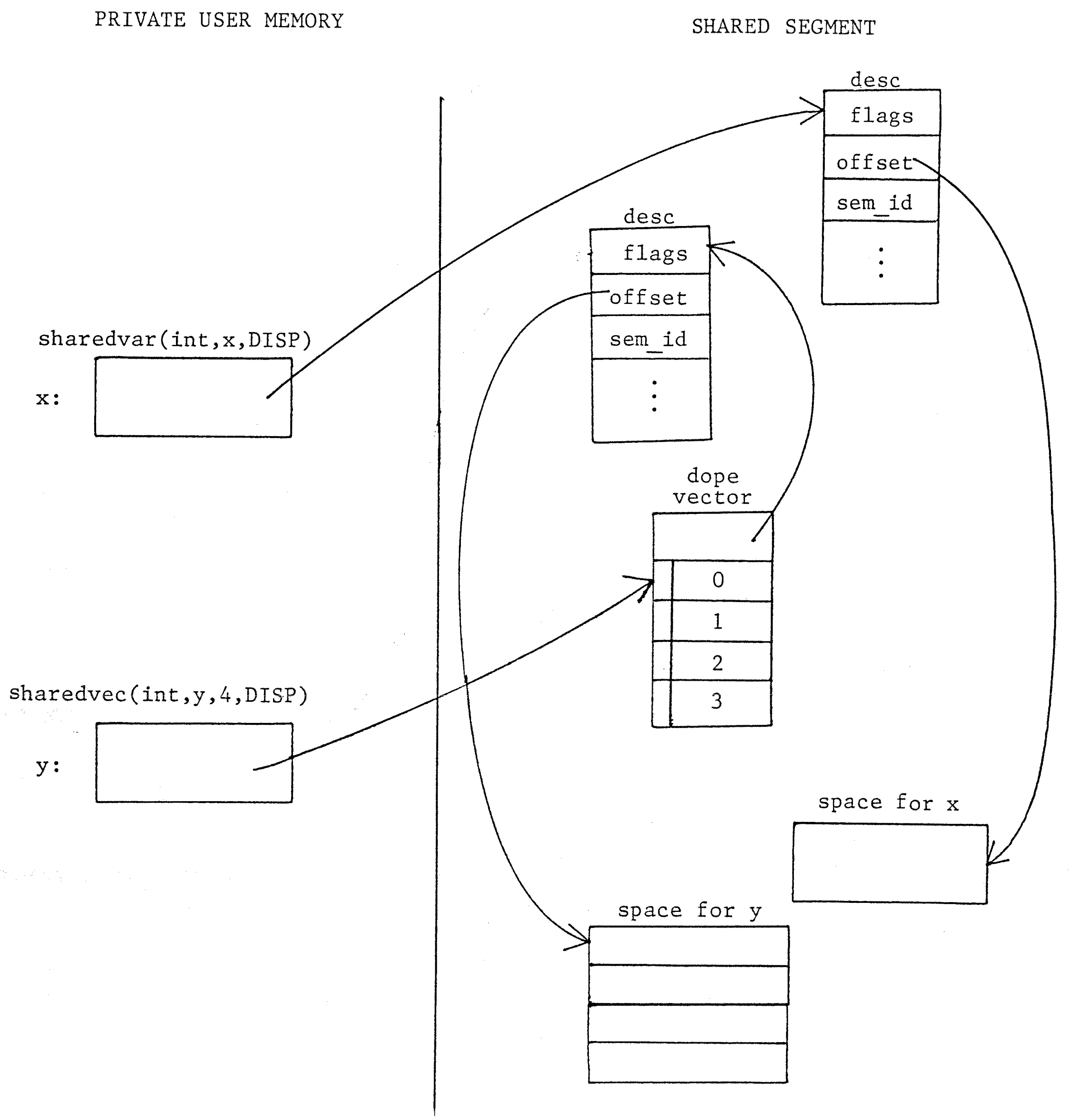

Figure 3 - Shared Segment Structure

element of the variable vector, containing the index of the corresponding element and a modified bit. This somewhat bizarre arrangement may be traced to two factors. First, to make the user interface seem as natural and "**C**-like" as possible, we wanted users to be able to access variables using the same syntax as they would in **C**. Thus,

stat = **aread** (x, &my_x);

and

stat = **agetelt** (A[6], &my_A_elt);

will atomically read the shared variable x, and the sixth element of the shared vector A. Since the user variables x and A must contain pointers into the shared segment, the pointer contained in A must allow indexing. The reason for this is that there was no way to determine within a macro that one of its arguments has the form of a reference to an array element. If it were, then the macro could call the necessary SMM access routine passing the pair (A,6), where A would just point to the descriptor whence we could locate A[6]. Indeed, in this case, there would be no need for two macros. This effect could have been obtained by requiring the user to write

stat = **agetelt** (A, 6, &my_A_elt);

but we felt strongly about the user interface and felt the situation could be improved in the future without modifying user code. Upon receiving the address "&A[6]" when called from the **agetelt** macro, the SMM reads the vector index stored at that address, indexes backwards to the front of the dope vector whence it can find the descriptor. The SMM now has a pointer to the descriptor and the index of the element to be accessed. An earlier design did not require the dope vector space but necessitated a search through the descriptors. We opted for speed over space in this case, furthermore the dope vector elements are relatively small.

Both the SMM and the DSMP have access to the semaphores associated with shared variables, hence they can insure that they do not interrupt each others access to the shared memory. Depending on the synchronization constraints imposed by the update discipline for the variable, the SMM may update the variable or may have to request in a message that the DSMP synchronize with all other DSMP's for a "globally atomic" update. In the global case, all copies of the variable in the system are modified at once.

Generally, the SMM routines called by **aread, awrite** and **aupdate** access shared variables by reading the descriptor to find the *sem_id* of the semaphore associated with the variable, performing a **Psem** on the semaphore, accessing the variable, and finally releasing the semaphore via **Vsem.** If a modification is performed by the SMM, the affected variable's modified indicator is marked before the modification routine (e.g. **awrite** or **aupdate**) returns to the calling program. When the SMM attempts to write a shared variable which has its initialized flag clear, the write is always performed and the flag is set regardless of the update discipline. Thus all variables, of all disciplines are initialized at the first write. If the SMM makes an attempt to read a shared variable which has not been initialized, the attempt returns an error status.

The **aupdate**, call provides for exclusive access to a shared variable by a user program for an extended period, i.e. for the duration of the execution of the user procedure passed as a parameter. In order to prevent the DSMP from being delayed for an arbitrary time, it must not be required to wait when it desires exclusive access to a variable. Since it is nonetheless necessary to use the semaphore primitives provided by the kernel to guarantee disciplined access, the DSMP makes use of the time-out facility of the **Psem** call, specifying a timeout of 0 (to poll). If the call is successful the DSMP can access the variable, otherwise the DSMP will consider the variable again later.

3.2. Remote Scheduling Process

The RSP's control and schedule the execution of distributed computations. They schedule processes for execution across the net, transfer data between processes, move files from machine to machine, and manage computation and program naming issues. As a group, the RSP's cooperate to determine the best location for queuing and executing processes. All processes that are queued for execution or are executing on a node are monitored by that node's RSP. Parameters are passed between main programs and subprograms by the RSP. Executable files that are not available on remote machines are transferred under control of the RSP. Each DCS computation has a network wide, unique computation name or *comp_id*, which is constructed by the RSP on the node on which it originates. The *comp_id* is the pair *<node_addr, pid>* containing the network address of the node and the process id (*pid*) of the root process. When sub-programs of the computation are scheduled on other nodes, the *comp_id* is forwarded to those nodes so that each program can be correctly associated with a computation for parameter passing and shared memory management. The RSP's act as the distributed manager of the DCS.

DCS uses a distributed scheduling algorithm in order to select the execution site for a process. Each RSP on the network periodicly broadcasts information to the other RSP's. The individual RSP's use this information to make a site placement decision for the processes queued on their own machines as the result of **doprog** calls. The only status information broadcast in the initial version is the set of load averages for the node. The current scheduling algorithm is the naive one: schedule the request on the node with the lowest load average. All requests for execution at a given site are honored. As mentioned above, one of the important research issues to be investigated using DCS is the development of other (better) heuristics for the scheduling decision.

After a site has been selected the local RSP sends a message to the remote RSP containing the information necessary to run the subprogram. This information includes the computation name, the name of the executable file, and the arguments, as well as the node address to which to return any results. The

RSP at the chosen site stores this information in a local table. To run the program, the RSP **fork**'s, the new child changes its user id into that of the requesting user, and then **exec**'s the subprogram. As discussed in the previous section, the subprogram communicates with the RSP for parameter passing. When the RSP receives a **getarg** message as the result of a call to **raccept**, it returns the arguments from its local table in a **sendarg** message. Upon completion, the sub-program may either return values via **rreturn** and terminate or simply terminate. When the subprogram terminates, the RSP receives a signal from the system (since the subprogram is a child of the RSP) and then does a **wait** to determine the identity and status of the terminating process. If an **rreturn** is executed, the SMM sends the return values to the RSP in a **retarg** message. The status, along with any return value is sent back to the invoking program's RSP. This RSP buffers the data and status returned until the main program asks for the results in a **parwait** call.

The design of the distributed shared memory requires that created segments remain in existance until the entire computation has terminated. Thus, the RSP's must detect the termination of a distributed computation. Each RSP keeps track of those processes that it starts or queues in its local tables. The table entries indicate, for each process, whether it is executing or has terminated, and if terminated whether there are surviving descendents. Table entries for processes are updated upon receipt of termination messages or (SIGCHLD) signals from the kernel. Thus, the local tables of the RSP's define a tree for each computation containing the processes that are alive or have living descendants. Using this information, an RSP can detect which processes are still executing on this node, and which children of these processes are still executing.

The progress towards termination of a computation is charted as follows. When a process terminates, the local RSP is notified and a message is sent to the RSP of the parent as described above. If this process has any live descendents (i.e. executing subprograms created by previous **doprog** calls) then the message sent to the parent indicates that there are live progeny for the terminating process. In this case, the table entry for the terminating process is not deleted, but is changed to indicate that it terminated with progeny still living. When the local RSP receives notification that these progeny have finished (i.e. that there are no more living descendents of this process), a final termination message is sent to the RSP of its parent and the local table entry is deleted.

Thus, when an RSP detects that a process and all of its descendents have terminated, it knows that that subtree has completed, and may be pruned. When the subtree corresponding to the root process of the computation has completed, then the entire computation is complete. At this time, all other RSP's are notified of the termination of the computation and they in turn inform their local DSMP's so that shared memory segments used by this computation may be deleted.

3.3. Distributed Shared Memory Process

The DSMP's are responsible for the creation and maintenance of all shared memories on all nodes. Since a distributed computation may have programs running on more than one machine in the network, there will be multiple copies of what is conceptually one shared memory segment, one for each node. As a result of separate parallel access, the individual copies may occasionally differ. These changes to the separate copies must be propagated to the other machines in order to resolve the differences. The DSMP's are responsible for keeping the various copies of a shared segment up to date. To accomplish this, the DSMP's communicate with each other, exchanging updates to modified shared variables. When a DSMP receives a new value for one of the variables it manages, it obtains control of the variable and updates its value. For each distributed shared segment of a computation, one node is designated as the "owner" or controlling site. This distinction is necessary for some of our protocols.

To coordinate updates to the various copies of a shared segment in the network the DSMP's use a special *seg_name* to refer to (all copies of) that segment. The seg_name is actually just the pair *<comp_id, seg_id>*, combining the network-wide name for the computation and the name given to the segment within the computation. Thus all segments on various machines that are instantiations of a particular computation's segment are referred to by the same name. There are thus three names for a shared memory segment. The *seg_id* used by the programs within a computation to declare and rendezvous on the segment (in an **openseg**). The *kern_seg_id* is the identifier by which the kernel memory mapping routines know the segment (from **vshare**).

Except for global update variables (which are described in the next section), the DSMP periodically checks all variables, in all segments, to see if any have been modified. When a modified variable is found (by checking the modified flags in the variable descriptors), its value is sent to the other DSMP's. When values are received, they are installed in the proper segment, subject to their respective update disciplines.

The creation and initialization process for a shared segment is somewhat complex and involves the local RSP, the SMM creating the segment and all the DSMP's, particularly the controlling (owner) DSMP. If a segment declared by a user process causes the DSMP to allocate a new instance of a segment previously existing in the network on another node, then this new segment's variables must be initialized to the current values from the copy on the node of the controlling DSMP. Thus, as part of the creation process, the creating DSMP executes the find-owner protocol, which either returns the identity of the controlling DSMP or informs the creator that it is the controlling DSMP implying that the segment did not previously exist. If the segment existed previously, a request for initial values is sent to the controlling DSMP, who cycles through the variables in the segment and sends messages with their values to the requesting DSMP, acquiring semaphores where necessary. After all initialized variables have been sent, the controlling DSMP sends a message to indicate that it has finished the initialization. The requesting DSMP installs these updates just as if they were the result of modifications from a user program. It also installs other update for variables in the segment that originate on other nodes. Thus this new copy is up to date when the DSMP receives word that the controller is done.

The entire procedure is as follows.

(1) The **close_seg** procedure in the SMM sends the process id (pid) of the declaring process and the seg_id for the segment being declared. (In return it will get a kern_seg_id to pass to **vshare**.)

(2) The DSMP requests the *comp_id* for the computation of which the declaring process is a member by sending its pid to the RSP. The RSP finds the *comp_id* in its local table unless the creating process is the root of a computation and has not previously created a segment. In the latter case, the RSP constructs a new *comp_id* for the computation.

(3) The RSP returns the comp_id to the DSMP who then constructs the *seg_name* of the segment being created and checks whether a segment by that *seg_name* already exists on its node. If it does, the DSMP returns its *kern_seg_id* to the SMM otherwise it returns 0.

(4) Upon receiving the kern_seg_id from the DSMP, the **close_seg** routine calls **vshare** with the *kern_seg_id* from the DSMP. If the *kern_seg_id* received from the DSMP is non-zero, then the segment already exists and will be mapped into the space of the caller by **vshare**. In this case, **close_seg** can then exit because the mapping is complete. For the remainder, we assume that the returned value was zero, i.e. that the SMM is creating the intial copy. In this case, the call to **vshare**, returns the *kern_seg_id* for the newly created shared segment.

(5) **Close_seg** now sends the new kern_seg_id to the DSMP, who enters it in its segment table along with the *seg_name*.

(6) The DSMP then calls **vshare** to map the new segment into its address space (expanding it if necessary).

(7) The DSMP executes the find_owner protocol to determine whether it needs to initialize the segment from elsewhere. If so, it requests initial values from the owner as described above. If not, then the DSMP makes itself the "owner" of the segment and will in the future act as a controller of the segment.

(8) When the initial values have been installed, the DSMP sends a message to **close_seg**, indicating that the segment is ready for use and **close_seg** then returns to the user program.

The DSMP may delete a shared segment when it is no longer being used. A shared segment is still in use as long as any part of the computation is still executing, on any machine in the net. The global termination of a computation is detected by the RSP's which then informs the DSMP's of the name of the computation (see section 3.2). The DSMP's then delete all shared segments with this same computation name.

3.4. Implementation of Update Disciplines

As described in the section on the user interface, a shared variable may be updated in one of several ways. This section discusses some of the implementation considerations as they affect the DSMP.

The *global update* discipline requires that the distributed shared variable behave as a real shared variable would, thus requiring global synchronization. This is performed within a node through the use of a semaphore attached to the local copy of the shared variable. Global synchronization is achieved through a global locking protocol implemented by the DSMP's. The process works as follows.

(1) The user wishes to modify a global update variable and calls **aupdate** or **awrite.** We will assume an **aupdate** call.

(2) The SMM requests a global lock for the variable from the local DSMP.

(3) The DSMP sends a request for the global semaphore to the segment's controlling DSMP. When it has been granted, the DSMP acquires the local semaphore for the variable. (Note, a fully distributed global locking mechanism such as described by Schneider [Schneider 82b] was considered but discarded. The reason is that the global update discipline already has such potential for delaying a distributed computation that we wanted to make locking as fast as possible.)

(4) After acquiring the semaphores, the DSMP broadcasts a message to the other DSMP's telling them to acquire their local semaphores for this variable. The DSMP waits for for acknowledgement from all other DSMP's and then replies to the SMM telling it to go ahead with the **aupdate.**

(5) The SMM receives this message and performs the update. When the operation is complete it changes the initialized flag if necessary and sends a message informing the DSMP that the update is done.

(6) The DSMP broadcasts the change to the other DSMP's. It again waits for acknowledgement from all the other DSMP's. The other DSMP's install the change, release their local semaphores, and acknowledge the message. Once the local semaphore on a node has been released, the new value may be read by processes on that node. When all acknowledgements have arrived the DSMP releases its local semaphore and notifies the controlling DSMP that it is releasing the global semaphore.

(7) The SMM is notified that the update is done, and **aupdate** returns.

Several disciplines specify some relationship between current values and allowable new values for integer or floating point variables, either for scalars or, component-wise, for vectors. The SMM must wait on the semaphore for the shared variable before it is allowed to update that variable. When an SMM updates a variable, it sets the modified bit in the descriptor to notify its DSMP that the values has changed. The DSMP must also wait on this local semaphore before writing or reading the shared variable. The global synchronization protocol described above applies to the "Global-update" discipline, and is not required for the other disciplines.

The unrestrained update discipline specifies that new variables can be written into the shared memory whenever the process interested in writing them can get the associated semaphore. The unsafe update discipline doesn't require a wait on the semaphore in order to update the variable.

4. Example

This example shows an implementation of the chaotic relaxation method for solving diagonal dominating systems of linear equations, Ax=b [Chazan 69]. The algorithm involves running separate processes in parallel, one for each row of the system. Each process repeatedly solves for the component of the solution vector corresponding to its row. Each time it does so, it uses the most recent value for the other components calculated by the other processes. The calculation of component i is given by

$$x_i = \frac{\sum_{j=1,j\neq i}^{n} a_{ij}x_j}{b_i} .$$

A controlling process monitors the solution vector, checking for convergence. This example was chosen, not because it represents the type of computation that would reap maximum benefit from distribution (the computation of each component is trivial), but because of its simplicity and its illustrative use of the facilities provided by DCS. There are two parts to the example, a driver program and a subprogram. The driver reads in the system to be solved, initializes variables, starts up a copy of the subprogram for each row of the system, and then goes into a loop checking for convergence of the solution. The solution vector is kept in global shared memory, so that each process can read the vector and modify its component visibly. When convergence is detected, the driver lets the subprograms know he is satisfied so they can quit, and then waits for their completion before printing the solution and terminating. Each time a subprogram solves its row of the system and updates its component of the solution in shared memory, it checks to see whether the driver has signalled convergence. If so, the subprogram returns its final solution component and terminates, otherwise it solves its row again.

The example program given below solves linear systems with dimension up to SIZE (defined by an option on the cc command). The driver program (see Figure 4) begins the declaration of a shared segment with an **openseg** statement. The seg_id is specified in the **openseg** with the constant SEG1. There are two variables placed in the segment; a simple variable, *done*, used to signal convergence and a vector, *x*, which holds the current approximate solution to the system. The convergence flag is declared in the **sharedvar** statement to be a character variable using the global update discipline (discipline 1 above). This discipline is used because it is the only one which will guarantee that all the subprograms will receive a new value after a single modification. Generally, the use of "global-update" should be avoided because of the high cost of updates. In this case, the variable *done* is written only once per computation, and reads to global-update variables cost no more than any other shared variable. The solution vector declared in the **sharedvec** statement contains double precision floating point numbers, is of length SIZE, and uses unrestrained update (discipline 6). The individual subprograms will be updating the components of this vector independently and repeatedly, thus this discipline is sufficient to make sure that current values get distributed. The declaration of the segment is then completed with the **closeseg** statement using the same seg_id as specified in the **openseg**.

The variables used for subprogram bookkeeping are *prog_id* and *status*; prog_id holds the identifiers returned by **doprog**, *status* holds the subprogram termination status'. The system to solve is defined by the variables *A*, *b*, and *n*. The matrix *A* holds the coefficients, *b* contains the right hand side, and *n* is the size of the system. When checking for convergence the program uses a non shared copy, *x1*, of the shared vector *x*. The convergence check requires past values of the solution vector, these are stored in vector *oldx*, and the error tolerance is in *error*. Finally, the structure *arg* is the argument block to be passed to the subprograms. Although the address of this structure is passed to **doprog**, the argument is actually passed by value. Therefore rather that a vector of these structures, only one is needed.

The driver first calls the input_data routine to read in *n*, *A*, *b*, and *error*. The vectors *x* and *oldx* are zeroed and *done* is assigned false. Note that the single **awrite** call is sufficient to initialize the entire *x* vector in shared memory. Now all variables, local and shared, have been initialized.

The driver then loops for each row of the system, initializing the argument block and starting a subprogram. The argument block gets assigned the size of the system, the subprograms row number, the corresponding coefficient matrix row, and rhs component. A **doprog** call it executed to request the scheduling of the subprogram. The identifier for this subprogram is placed in the *prog_id* vector, its termination status will be placed in the corresponding element of the

```
#include              "dcs/h/dcs_user.h"
#define         TRUE            1
#define         FALSE           0
#define         INTERVAL        3
#define         SEG1            1

main()          /* chaotic relaxation driver */
{
        openseg(SEG1);    /* place the solution vector in shared memory */

        sharedvar(char, done, UPDT_GLOBAL);                /* termination indicator */
        sharedvec(double, x, SIZE, UPDT_UNREST);           /* solution vector */

        closeseg(SEG1);

        short   prog_id[SIZE];              /* holds the ids returned by doprog */
        int     status[SIZE];               /* subprogram termination status */
        double  A[SIZE] [SIZE];             /* coefficient matrix */
        double  b[SIZE],                    /* rhs of system */
                oldx[SIZE],                 /* comparison vector */
                x1[SIZE];                   /* solution vector - local copy */
        int     n;                          /* size of the system */
        int     i,j;                        /* loop control */
        int     pstatus;                    /* parwait status */
        char    converged,                  /* termination indicator - local copy */
                false = FALSE,              /* pointer to false - for awrite */
                true = TRUE;                /* pointer to true - for awrite */
        double  error;                      /* relative error tolerance */
        struct  {                           /* subprogram argument block */
                double  a[SIZE];            /* ..coefficient matrix column */
                double  b;                  /* ..b vector entry */
                int     n,                  /* ..number of variables */
                pos;                        /* ..equation position */
        }       arg;


        input_data(A,b,&n,SIZE,&error);     /* read the array and RHS */
        arg.n = n;

        /* initialize the shared variables */
        for (i=0; i<n; i++)
                oldx[i] = 0;                /* comparison vector is zero ...*/
        awrite(x,oldx);                     /* ... and so is the initial approximation */
        awrite(done, &false);               /* indicate we have not converged */

        /* start a subprogram for each row */
        for(i=0; i<n; i++) {
                for (j=0; j<n; j++)         /* init the argument */
                        arg.a[j] = A[i][j];
                arg.b = b[i];
                arg.pos = i;
                if (                        /* request execution of a subprogram */
                   (prog_id[i] = doprog("/student/cs.paul/paulm/chaos_sub",
                                    &status[i],
                                    &arg,sizeof(arg),
                                    &oldx[i],sizeof(oldx[i])))
                    <= 0)
                        (void) printf(" Error %d in starting subprogram %d.0,
                                                                        prog_id[i], i);
        }

        /* loop until the solution vector has converged */
        do {
                sleep(INTERVAL);            /* pause here to allow sub_programs to work */
                aread(x,x1);                /* read the solution vector */
                converged = TRUE;
                for (i=0; i<n; i++) {                    /* check accuracy */
                        if (fabs(x1[i] - oldx[i]) > error)
                                converged = FALSE;
                        oldx[i] = x1[i];                 /* update previous value */
                }
        } while (!converged);
        awrite(done, &true);                             /* signal convergence */

        pstatus = parwait_v(prog_id, n);                 /* wait for completion */
        if (pstatus) {                                   /* all completed ok? */
                (void) printf("Parwait status = %d.0,pstatus);
                for (i=0; i<n; i++)
                        if (status[i]) (void) printf("Error %d in computing row %d.0,
                                                        status[i],i);
        }
        for (i=0; i<n; i++)                              /* print the solution */
                (void) printf(" x[%d] = %f0,i,oldx[i]);
}

input_data(a,b,n,size,error)
double  a[][SIZE];
double  b[];
int     *n,size;
double  *error;
{
        int     i,j;

        (void) scanf("%d",n);                                   /* get array size */
        if (*n < 1 || *n > size)        (void) printf("bad dimension0);
        for(i=0; i<*n; i++) {
                for (j=0; j<*n; j++)
                        if(scanf("%lf",&a[i][j]) < 1)           /* A values */
                                (void)printf("data format error0);
                if (scanf("%lf",&b[i]) < 1)                     /* b values */
                                (void)printf("data format error0);
        }
        if (scanf("%lf",error) < 1) (void)printf("data format error0);

        return;
}
```

Figure 4 -- Chaotic Relaxation Driver

status vector.

After starting all the subprograms the driver begins the convergence check. First it calls *sleep* to allow the subprograms to get some work done and modify their components. This step also gives up the cpu so that subprocesses on the same node as the driver will have a chance to compute (rather than the driver merely wasting the remainder of his time slice). Next the solution vector is read from shared memory in a single **aread** call. The difference between the new solution and the previous solution must be less than the error tolerance for each component in order for convergence to be reached. When the convergence condition is met, the convergence flag is assigned TRUE.

Parwait_v is now called with the vector of prog_id's. When all the subprograms have terminated **parwait_v** returns, the final solution is printed out, and the driver exits. Note that the final solution is returned as the values from the subprograms (this was set up in the **doprog** call) and are only available after **parwait_v** returns. By the time the driver calls **parwait_v**, it already has the solution in shared memory and could terminate without waiting. However our method has the advantage of getting the results of any work the subprograms did after the driver read its last set of values and before the subprogram terminated. Furthermore it demonstrates the use of the **parwait_v** call.

The subprogram is extremely simple (see Figure 5). The same shared segment is declared as was described above. The argument sent in the **doprog** call is accepted. The computation for this row is executed using the current solution read from shared memory. The result is written back to shared memory. The subprogram then pauses to allow other subprograms to work. If the computation loop is executed twice with the same values, the same solution will result, so with this pause the subprogram gives up the cpu in order that other subprograms on the same machine can get a chance to modify the solution. This is a practical consideration, its absence would not effect the correctness of the solution, but merely waste time. The convergence flag is then read, if it is true the subprogram returns its last computed component and exits, otherwise the computation is started again.

5. Some Preliminary Timings

The first stage of the implementation of DCS to be completed was the RSP and its associated user modules, RSM and REM. At this point we ran a simple test to check on the ease and cost of performing some distributed computations. Since our system is intended for cpu intensive applications our test was cpu intensive, consisting of simply the execution of one million floating point multiplications. We ran two programs, one sequential and the other distributed. Our test was designed to discover the speedup achievable when perfect parallelism is possible. Therefore the sequential program ran the million multiplies serially and the distributed program used ten subprograms, each executing 100,000 operations, to achieve the same goal.

The distributed program has two parts, a driver and a subprogram. The driver program starts up 10 copies of the subprogram, each with an argument vector containing 10 floating point numbers, and then waits for them to complete. Each subprogram performs 100,000 floating point multiplies and then

```
#include "dcs/h/dcs_user.h"
#define         TRUE            1
#define         FALSE   0
#define         SUB_INTERVAL    1
#define         SEG1            1

main()          /* chaotic relaxation subprogram */
{
        openseg(SEG1);      /* place the solution vector in shared memory */

        sharedvar(char, done, UPDT_GLOBAL);
        sharedvec(double, x, SIZE, UPDT_UNREST);

        closeseg(SEG1);

        int     j;                              /* loop control */
        char    converged = FALSE;              /* local copy of done */
        double  y[SIZE],                        /* local copy of x */
                sum;
        struct  {                               /* argument block */
                double  a[SIZE];                /* ..coefficient matrix column */
                double  b;                      /* ..b vector entry */
                int     n,                      /* ..number of variables */
                        pos;                    /* ..equation position */
        }       arg;


        raccept(&arg, sizeof(arg));             /* get the arg values */

        /* loop until the solution converges */
        do {
                aread(x, y);                    /* read up the current x values */

                /* calculate our component of the solution */
                sum = 0;
                for (j=0; j< arg.pos; j++)
                        sum += arg.a[j] * y[j];
                for (j=arg.pos+1; j<arg.n; j++)
                        sum += arg.a[j] * y[j];
                sum = (arg.b - sum) / arg.a[arg.pos];

                aputelt(x[arg.pos],&sum);       /* write a new x value */
                sleep(SUB_INTERVAL);            /* pause to allow others to do work */
                aread(done, &converged);        /* read the convergence flag */
        } while (!converged);

        rreturn(&y[arg.pos],sizeof(y[arg.pos])); /* return the final value */
}
```

Figure 5 -- Chaotic Relaxation Subroutine

returns the argument vector to the driver. Thus we have 1,000,000 floating point multiplies, 10 process creations, and the associated communications involved in the computation. The sequential program merely loops one million times performing multiplications.

The distributed test was run on 5 SUN workstations running our modified kernel supporting shared memory. The test was repeated six times. The sequential test was run on a single SUN workstation, and was repeated four times (the variation in run times was so small with this test that further repetition was considered pointless).

When comparing the running times of these two programs we compare the process execution time of the sequential program to the real elapsed time for the distributed program. The process time for the program is a lower bound for the real time in which the sequential program could execute. The real time measured for the distributed program can then be compared to this to get an indication of the speedup achievable using DCS (on unloaded systems). The results are listed below.

	Distributed	Sequential
	real time seconds	process time seconds
	196.1	560.4
	244.7	554.6
	159.0	553.8
	221.0	554.0
	155.5	
	164.7	
average	190.2	555.7

These results show that even after all overhead costs are included (because we measure real time), we get a distributed program which runs approximately 3 times faster (2.9) than the best the sequential program can do.

6. Relation to Other Work

There has not been a great deal of work done specifically in this area, although there have been a number of projects addressing problems in related areas. The Worm program [Schoch 82] is the first of which we're aware. The program expanded onto more workstations as they became available, however it was based on the "off hours" use of workstations that were assumed idle, and required rebooting the workstation. Grapevine [Schroeder 84, Birrell 82] and Medusa [Ousterhout 80] are both examples of distributed programs i.e. programs with autonomous, cooperating segments working on different sections of the same problem. The first is a mail-server and name database running on a network while the second is an operating system running on the network computer Cm* (actually a shared and distributed memory, multiprocessor system). LOCUS [Popek 81, Walker 83, Mueller 83] is a Unix-based network operating system in which a file system supporting nested transactions and replicated files is distributed over the entire network, while the operating system code is replicated on each node. LOCUS also comes close to addressing many of the issues we will face, since it has protocols supporting remote command execution to handle the case where the program implementing the command is located on another node of the network.

LOCUS does not address the issue of load balancing and automatic site determination, the latter being determined by a user setable "advice list". DEMOS/MP [Powell 83] contains a mechanism supporting process migration in networks, but doesn't address the problem of when to migrate a process. Powell and Miller maintain that designing "an efficient and effective decision rule [for process migration] is still an open research topic." As can be seen from the above discussion, we have some good advice available on related issues that we will need to consider, but none that deals with the same problem.

The closest work to ours is the work being done by Finkel and Manber in the context of the Crystal Project at the University of Wisconsin. Their system DIB (for Distributed Implementation of Backtracking) [Finkel 85], provides support for the writing of backtracking recursive treesearch algorithms. The DIB driver calls a user supplied problem generator to get sub-problems of a given problem. These sub-problems are then distributed to the nodes of a network based on minimal load average. The only communication between processes working on sub-problems possible in their system is via input parameters and output values. We differ from their work by providing a more flexible interface that allows arbitrary invocations under programmer control, and also in the provision of shared variables for communication amongst running processes.

7. Status and Future Work

As of this writing, DCS is approximately 95% functional. Procedure invocation with input and output parameters has been working since early June, and we've included the results of some early measurements. The design of the protocols supporting shared variables has turned out to be far more difficult than we originally anticipated largely for two reasons. First, the shared memory support protocols typically involve several nodes at once whereas the procedure invocation protocols generally operate between two nodes. Second, we decided very early in the design to use the datagram protocol for efficiency.

The justifications for the use of datagrams were first, that datagrams have fairly high reliability on local area networks, and second that the general philosophy of our shared variables requires programs to be somewhat tolerant where shared variables are concerned. In an iterative algorithm, an update broadcast may be lost, but as the variable is modified repeatedly more update broadcasts will be sent, and most of these will be received. If the last update is critical, then steps can be taken to make sure it arrives, such as designing the sub-program to return the final value, or declaring the variable to have the global update discipline as we did in our example in Section 4 for the variables x and *done* respectively.

At the same time, there are instance where lost messages could not be tolerated or messages had to be processed in order. Examples are segment creation and the implementation of global updates. In these cases, we were forced to design tolerance for the unreliable service into our protocols, which made them more complicated. If a reliable datagram service had been available, we would have paid the price for the channels used for these functions, and stayed with the cheaper service for the normal update processing, which will make up the greatest bulk of the message traffic anyway.

In the near future, DCS will be completely functional, and we can begin the research for which it was initially planned. That is, we will write and encourage others to write, programs based on the DCS abstractions. We will then experiment with some of the heuristics mentioned in the Introduction with various distributed algorithms and under varying ambient user load to find some good scheduling algorithms.

We will also be carrying out investigations into the usefulness of update disciplines for writing distributed programs. Further, the fact that our distributed shared variables may temporarily have inaccurate values will affect the proof obligations for formal verification. Though approches exist for demonstrating the correctness of parallel programs with shared memory [Owicki 76], the known methods assume that the shared variables are all in physicaly shared memory. We wish to develop a proof methodology or at minimum a set of programmer guidelines for the use of distributed shared variables.

8. Acknowledgments

A good portion of the low level design and coding of this system was done as a semester project by members of an advanced course in networking taught by the first author during the Spring Semester, 1985. We would like to thank the students of that course, including especially Dave Wood and Greg Bollendonk who have continued working on the project and have helped a great deal in getting it to its present state. Dave contributed extensively to the design of the DSMP communication protocols, and to the implementation of the DSMP. Greg was instrumental in the implementation of shared memory. We also owe thanks to Muhammad Benten for his help in coding the DSMP.

References

[Birrell 82] A. D. Birrell, R. Levin, R. M. Needham, M. D. Schroeder. Grapevine: An Exercise in Distributed Computing. ***Communications of the ACM 25*** (4):260-273, (April 1982).

[Chazan 69] D. Chazan, W. Miranker.
Chaotic Relaxation.
Linear Algebra and Applications 2, pages 199-222, (1969).

[Dijkstra 68a] E. W. Dijkstra.
Cooperating Sequential Processes.
In ***Programming Languages,*** (F. Genuys, Editor), Academic Press, New York, 1968.

[Finkel 85] R. Finkel, U. Manber.
DIB--A Distributed Implementation of Backtracking.
Proceedings of the 5th International Conference on Distributed Computing Systems, (Denver, Colorado), pp 446-452, May 1985.

[Harter 85c] P. K. Harter, Jr. G. R. Bollendonk.
An Implementation of Shared Memory for Unix with Real-Time Synchronization.
University of Colorado, Computer Science TR #CU-CS-310-85.

[Mueller 83] E. T. Mueller, J. D. Moore, G. J. Popek.
A Nested Transaction Mechanism for LOCUS.
Proceedings of the Ninth Symposium on Operating Systems Principles, (Bretton Woods, 1983), published as ***Operating Systems Review 17*** (5):71-89, (December 1983).

[Ousterhout 80] J. K. Ousterhout, D. A. Scelza, P. S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23 (2):92-104, (February 1980).

[Owicki 76] S. S. Owicki, D. Gries.
An Axiomatic Proof Technique for Parallel Programs I.
Acta Informatica 6 :319-340, Springer Verlag, 1976.

[Peterson 79] J. L. Peterson.
Notes on a workshop on distributed computing.
Operating Systems Review 13 (3):18-27, (July 1979).

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel.
LOCUS: A Network Transparent, High Reliability, Distributed System.
Proceedings of the Eighth Symposium on Operating Systems Principles, (Asilomar, 1981), published as ***Operating Systems Review 15*** (5):169-177, (December 1981).

[Powell 83] M. L. Powell, B. P. Miller.
Process Migration in DEMOS/MP.
Proceedings of the Ninth Symposium on Operating Systems Principles, (Bretton Woods, 1983), published as ***Operating Systems Review 17*** (5):110-118, (December 1983).

[Schnabel 85] R. B. Schnabel.
Parallel Computing in Optimization.
Proceedings of the NATO Advanced Study Institute on Computational Mathematical Programming, (Klaus Schittkowski, Editor), Springer-Verlag, 1985.
Also Available as University of Colorado Technical Report CU-CS-282-84.

[Schneider 82b] F. B. Schneider.
Synchronization in Distributed Programs.
ACM Transactions on Programming Languages and Systems 4 (2):179-195, April 1982.

[Schoch 82] J. F. Schoch and J. A. Hupp.
The 'Worm' Program - Early Experience with a Distributed Program.
Communications of the ACM 25 (3):172-180 (March 1982).

[Schroeder 84] M. D. Schroeder, A. D. Birrell, R. M. Needham.
Experience With Grapevine: The Growth of a Distributed System.
ACM Transactions on Computer Systems 2 (1):3-23, (February 1984).

[Spector 82] A. Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Proceedings of the Eighth Symposium on Operating Systems Principles, (Asilomar, 1981), full paper published in: ***Communications of the ACM 25*** (4):246-259, (April 1982).

[Walker 83] B. Walker, G. Popek, R. English, C. Kline, G. Thiel.
The LOCUS Distributed Operating System.
Proceedings of the Ninth Symposium on Operating Systems Principles, (Bretton Woods, 1983), published as ***Operating Systems Review 17*** (5):49-70, (December 1983).

Appendix

The following pages contain the Unix manual entries for the routines and macros supporting the DCS user interface.

NAME

doprog - queue a program for (remote) execution.

SYNOPSIS

prog_id = doprog(fname, &status, &arg, sizeof(arg), &ret, sizeof(ret))
int prog_id, status;
char *fname;
x arg;
y ret;

DESCRIPTION

Doprog causes the program on the file named by *fname* to be queued for remote execution by the DCS system. *Status* is the address of an integer to contain the termination status of the program (i.e. the main program can determine if a subprogram has crashed by examining this value). The format of the status is described in <sys/wait.h>, the *Fill1* field of the status is used to return DCS system error status codes. *Arg* contains the value of type *x* to be passed as an argument to the program. *Ret* is the address of a data block of type *y* to receive the values returned by the program. The *ret* value is not returned until a *parwait* call has been executed for the *prog_id* .

RETURN VALUE

Upon successful completion the remote program identifier, *prog_id* , for the scheduled program is returned, this value may be used in subsequent calls to *parwait.* If a negative number is returned an error has occured.

DIAGNOSTICS

A negative return value indicates one of the following error conditions:

-1	the global variable *errno* should be checked.
-2	the DCS system has not responded to the *doprog*
-3	this program has exceeded the DCS system limit for the number of outstanding *doprog* calls.
-4,-5	DCS system installation errors.

SEE ALSO

parwait(3)

FILES

dcs/lib/dcslib	library containing this routine.

AUTHORS

Paul Harter, Paul Maybee

NAME

parwait - wait for the completion of a set of remote programs.

SYNOPSIS

status = parwait(prog_id0, prog_id1, ..., prog_idn,0)
int status, prog_id0, prog_id1, ..., prog_idn

status = parwait_v(prog_id_v, n)
int prog_id_v[],n

DESCRIPTION

Parwait returns when all subprograms identified by *prog_id0, prog_id1, ..., prog_idn* have terminated. These *prog_id s* must be values returned by previous *doprog* calls. The values computed by the programs (i.e. the *doprog ret* parameters) are available after the *parwait* has returned.

Parwait_v is an alternate calling mechanism, allowing the *prog_id s* to be elements of a vector, *prog_id_v* , with *n* specifying the length of the vector.

RETURN VALUE

Upon completion a 0 is returned if all subprograms completed without errors. Any other value returned indicates an error condition.

DIAGNOSTICS

A non-zero, positive return value indicates the number of subprograms that did not terminate normally (e.g. because of machine crashes or communications failures). A negative return values indicates a failure in the processing of the *parwait* call.

-1	refer to the global variable *errno*.
-2	the DCS system has not reponded to the *parwait* call.
-3	indicates that no *doprog* proceeded this *parwait* call.
-4	indicates an internal DCS system error.

SEE ALSO

doprog(3), rreturn(3)

FILES

dcs/lib/dcslib	library containing this routine.

AUTHORS

Paul Harter, Paul Maybee

NAME

raccept - remote accept.

SYNOPSIS

raccept((char*) &arg, sizeof(arg))
x arg

DESCRIPTION

This routine is called by a subprogram of a distributed computation to accept the parameter passed to this subprogram by the main program. *Arg* is the variable to receive the argument specified as *arg* on the *doprog* call that caused the execution of this subprogram. In order to insure proper alignment and addressing in the user program, the type, *x* , of *arg* must match the type of the corresponding parameter on the *doprog* call.

DIAGNOSTICS

If the routine is not successful in obtaining the argument then *raccept* will cause a process exit. The exit status portion of the status (see <sys/wait.h>) returned after a *parwait* has been called for this subprogram will contain a negative value indicating the failure.

SEE ALSO

doprog(3), parwait(3)

FILES

dcs/lib/dcslib	library containing this routine.

AUTHORS

Paul Harter, Paul Maybee

NAME

rreturn - remote return.

SYNOPSIS

rreturn((char∗) &ret, sizeof(ret))
y ret

DESCRIPTION

This routine is called by a subprogram of a distributed computation to return the output variable to the main program. The main program receives *ret* in the variable specified by *ret* on the *doprog* call which caused the execution of this subprogram. The type, *y* , of *ret* must match the type of the corresponding variable on the *doprog* call to insure proper addressing and alignment.

SEE ALSO

doprog(3), parwait(3)

FILES

dcs/lib/dcslib	library containing this routine.

AUTHORS

Paul Harter, Paul Maybee

NAME

openseg, sharedvar, sharedvec, closeseg - declare shared variables.

SYNOPSIS

#include "dcs_user.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

DESCRIPTION

This set of declarations is an example layout of the macros to define a shared memory segment in a program of a distributed computation.

Closeseg begins the definition of the segment and uses the integer constant *seg_id* to identify the segment. This declaration must appear before any shared memory declarations (i.e. *sharedvar* and *sharedvec)* for this segment.

Sharedvar declares the non-vector variable *x* to be of type *x_type* and allocates space for it in the shared memory segment specified by the previous *openseg* . *Sharedvec* declares the variable *y* to be a vector with length *length* and of type *y_type* and allocates space for it in the shared memory segment specified by the previous *openseg* . *X_type* (*y_type*) may be any valid C type with the exception that if it is a structure or union type then only the tag field is used. The *upt* parameter is an integer which specifies the updating discipline to use for this variable. The possible values for *upt* are defined in the include file dcs_user.h. The only access to this variable is through the *aread, awrite,* and *aupdate* routines, all other references have undefined results.

The *closeseg* macro terminates the mapping of a shared memory segment. The *seg_id* must be identical to the *seg_id* specified in the previous *openseg* declaration. None of the variables in this segment (declared with *sharedvar,* or *sharedvec)* can be accessed until this declaration has appeared.

There may be multiple occurances of *sharedvar* and *sharedvec* in between an *openseg* and its corresponding *closeseg* . A single program may declare multiple shared segments, each with a unique *seg_id* .

SEE ALSO

aread(3), awrite(3), aupdate(3)

FILES

dcs/lib/dcslib	library containing code to process this declaration.

AUTHORS

Paul Harter, Paul Maybee, David C. M. Wood

NAME

aread, agetelt - atomicly read a variable from shared memory.

SYNOPSIS

```
#include "dcs_user.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

x_type dest1;
y_type dest2;
 .
 .
 .
aread( x, &dest1 );

agetelt( y[i], &dest2 );
```

DESCRIPTION

The synopsis above is a segment of an example program of a distributed computation. *Aread* and *agetelt* are used to atomicly read variables stored in shared memory. These functions reads the contents of a shared variable into nonshared storage.

The *aread* routine reads an entire variable, whether vector or non-vector. *Agetelt* reads an element of a shared vector variable. *X* (*y[i]*) is the variable whose contents is desired. *Dest1* (*dest2*) is the variable to hold the value read from shared storage. *X* and *y* must be variables defined with *sharedvar,* or *sharedvec* declarations. *Dest1* and *dest2* must not have been defined in this way.

RETURN VALUE

Upon successful completion a 0 is returned. If an error has occured then a -1 is returned. Possible errors are: the destination address is in shared memory, the variable is not in a shared segment, and the variable has not been initialized.

SEE ALSO

openseg(3), awrite(3), aupdate(3)

FILES

dcs/lib/dcslib	library containing code to process this declaration.

AUTHORS

Paul Harter, Paul Maybee, David C. M. Wood

NAME

awrite, aputelt - atomicly write to a shared variable.

SYNOPSIS

```
#include "dcs_user.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

x_type dest1;
y_type dest2;
 .
 .
 .
awrite( x, &source1 );

aputelt( y[i], &source2 );
```

DESCRIPTION

The synopsis above is a segment of an example program of a distributed computation. *Awrite* and *aputelt* are used to atomicly write variables stored in shared memory, subject to the update descipline, *upt* , with which they were declared. These functions write the contents of a non-shared variable into a shared variable.

The *awrite* routine modifies an entire variable, whether vector or non-vector. *Aputelt* modifies an element of a shared vector variable. *X* (*y[i]*) is the variable which is being modified. *Source1* (*source2*) is the variable which contains the value to be written. *X* and *y* must be variables defined with *sharedvar,* or *sharedvec* declarations. *Source1* and *source2* must not have been defined in this way.

RETURN VALUE

Upon successful completion a 0 is returned. If an error has occured then a -1 is returned. Possible errors are the source address is in shared memory or the variable is not in a shared segment.

SEE ALSO

openseg(3), aread(3), aupdate(3)

FILES

dcs/lib/dcslib	library containing this routine.

AUTHORS

Paul Harter, Paul Maybee, David C. M. Wood

NAME

aupdate, aupdelt - atomicly update a shared variable.

SYNOPSIS

```
#include "dcs_user.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

char *param;
int iflag;
int proc1(param, x, iflag), proc2(param, y, iflag);
 .
 .
 .
aupdate( proc1, x, param )

aupdelt( proc2, y[i], param )
```

DESCRIPTION

The synopsis above is a segment of an example program of a distributed computation *Aupdate* and *aupdtelt* are used to atomicly update variables stored in shared memory. These functions read and modify the contents of a shared variable subject to the update discipline, *upt,* with which they were declared.

The *aupdate* routine works on an entire variable, whether vector or non-vector. *Agetelt* works on an element of a shared vector variable. *X* (*y[i]*) is the variable whose contents is being updated. *Proc1* (*proc2*) is the user routine which will be called to do the modification. *Param* is an additional parameter to be passed to the user routine. *Iflag* is a third parameter which indicates whether *x* (*y*) has been initialized yet (*Iflag* is 1 if initialized). *Proc1* (*proc2*) has exclusive access to *x* (*y*) during its execution. *X* and *y* must be variables defined with *sharedvar,* or *sharedvec* declarations. *Param* must not have been defined in this way.

RETURN VALUE

Upon successful completion a 0 is returned. If an error has occured then a -1 is returned. A possible error is the variable is not in a shared segment.

SEE ALSO

openseg(3), aread(3), awrite(3)

FILES

dcs/lib/dcslib	library that contains this routine.

AUTHORS

Paul Harter, Paul Maybee, David C. M. Wood

NAME

intro - introduction to DCS library functions

DESCRIPTION

DCS provides the C programmer with a library of functions to support parallel distributed programs on a network of SUN workstations. To access this library, compile your program with the dcs library. The link editor searches this library when the "-ldcs" option is supplied on the C compilation commmand. Declarations and macros used with DCS are in the include file *"/usr/local/include/dcs.h"*. The functions are listed below.

LIST OF FUNCTIONS

Function	*See*	*Description*
agetelt	aread	atomicly read a variable from shared memory
aputelt	awrite	atomicly write to a shared vector element
aread	aread	atomicly read a variable from shared memory
aupdate	aupdate	atomicly update a shared variable
aupdelt	aupdate	atomicly update a shared vector element
awrite	awrite	atomicly write to a shared variable
closeseg	openseg	end declaration of shared variables
contin	contin	continue execution of synchronized programs
doprog	doprog	queue a program for (remote) execution
openseg	openseg	begin declaration shared variables
parsync	parsync	synchronize with a set of subprograms
parwait	parwait	wait for the completion of remote programs
raccept	raccept	accept remote parameter
rprintf	rprintf	print messages to the controlling terminal
rreturn	rreturn	remote parameter return
sharedvar	openseg	declare single shared variable
sharedvec	openseg	declare single shared vector
syncp	syncp	synchronize with the parent program

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

aread, agetelt - atomicly read a variable from shared memory.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

x_type dest1;
y_type dest2;
 .
 .
 .
aread( x, &dest1 );

agetelt( y[i], &dest2 );
```

DESCRIPTION

The synopsis above is a segment of a program in a distributed computation. *aread* and *agetelt* are used to atomicly read variables stored in shared memory. These functions read the contents of a shared variable into nonshared storage.

The *aread* routine reads an entire variable, whether vector or non-vector. *agetelt* reads an element of a shared vector variable. *aread* reads the value of the shared variable *x* and copies it to the local variable *dest1*, likewise *agetelt* reads *y[i]* into *dest2*. *x* and *y* must be variables defined with *sharedvar*, or *sharedvec* declarations. *dest1* and *dest2* must be regular variables, and not have been declared using either of these.

RETURN VALUE

Upon successful completion 0 is returned. If an error has occurred then -1 is returned.

SEE ALSO

openseg(3), awrite(3), aupdate(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

aupdate, aupdelt - atomicly update a shared variable.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

char *param;
int iflag;
int proc1(param, x, iflag), proc2(param, y, iflag);
 .
 .
 .
aupdate( proc1, x, param );

aupdelt( proc2, y[i], param );
```

DESCRIPTION

The synopsis above is a segment of a program in a distributed computation. *aupdate* and *aupdtelt* are used to atomicly update variables stored in shared memory. These functions read and modify the contents of a shared variable subject to the update discipline, *upt*, with which they were declared.

The *aupdate* routine updates an entire variable, whether vector or non-vector; *agetelt* a single element of a shared vector variable. *aupdate* reads the value of the shared variable *x* and calls the routine *proc1* supplying *x* as the second actual parameter.
aupdelt reads the value of the shared vector element *y[i]* and calls the routine *proc2* supplying *y[i]* as the second actual parameter. The first actual parameter in each case is an arbitrary pointer variable, *param*, and the third parameter is *iflag*, which is 0 or 1 depending on whether the variable to be updated was initialized previous to the *aupdate* (*aupdelt*) call. *proc1* and *proc2* may access their second parameter directly (i.e. without using DCS shared variable access functions). *x* and *y* must have been declared using *sharedvar* or *sharedvec* declarations, *param* must be a regular variable, and not have been declared using these calls.

Nested *aupdate* and *aupdelt* calls are not allowed.

RETURN VALUE

Upon successful completion 0 is returned. If an error has occurred then -1 is returned.

SEE ALSO

openseg(3), aread(3), awrite(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

awrite, aputelt - atomicly write to a shared variable.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);

x_type dest1;
y_type dest2;
 .
 .
 .
awrite( x, &source1 );

aputelt( y[i], &source2 );
```

DESCRIPTION

The synopsis above is a segment of a program in a distributed computation. *awrite* and *aputelt* are used to atomicly write variables stored in shared memory, subject to the update discipline, *upt*, with which they were declared. These functions write the contents of a non-shared variable into a shared variable.

The *awrite* routine modifies an entire shared variable, whether vector or non-vector; *aputelt* a single element of a shared vector variable. *awrite* copies the value of the local variable *source1* into the shared variable *x*, likewise *aputelt* copies *source2* into *y[i]*. *x* and *y* must be variables defined with *sharedvar*, or *sharedvec* declarations. *Source1* and *source2* must be regular variables, and not have been declared using these declarations.

RETURN VALUE

Upon successful completion 0 is returned. If an error has occurred then -1 is returned.

SEE ALSO

openseg(3), aread(3), aupdate(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

BUGS

awrite fails when the variable is larger than 1024 bytes. If the variable is a vector then multiple *aupdelt* calls can be used to share larger amounts of data.

NAME

contin, contin_v - allow synchronized subprograms to continue execution.

SYNOPSIS

#include "/usr/local/include/dcs.h"

status = contin(prog_id0, prog_id1, ..., prog_idn,0)
int status, prog_id0, prog_id1, ..., prog_idn

status = contin_v(prog_id_v, n)
int prog_id_v[],n

DESCRIPTION

Contin informs all the subprograms identified by *prog_id0, prog_id1, ..., prog_idn* that they may continue executing from a *syncp*. These *prog_id*'s must be values specified in a previous *parsync* call with no intervening *contin* or *parwait* calls for the same ids.

Contin_v is an alternate calling mechanism, allowing the *prog_id*'s to be elements of a vector, *prog_id_v*, with *n* specifying the length of the vector.

RETURN VALUE

0 is returned for normal completion, otherwise -1 is returned. The type of error can be determined from the system variable *errno*.

SEE ALSO

doprog(3), parsync(3), parwait(3), syncp(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

doprog - queue a program for (remote) execution.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

prog_id = doprog(fname, &status, &arg, sizeof(arg), &ret, sizeof(ret),rrs)
int prog_id, status;
char *fname;
type1 arg;
type2 ret;
struct res_req_spec rrs[ ];
```

DESCRIPTION

Doprog causes the program in the file named by *fname* to be queued for remote execution by the DCS system. *Status* is of type **union** wait (see <sys/wait.h>), it will contain the termination status of the remote program after it has exited. The format of the status is described in <sys/wait.h>, the *Fill1* field of the status is used to return DCS system error status codes. *arg*, a variable of type *type1*, is passed as an argument to the remote program. *ret*, a variable of type *type2*, is used to contain the return values from the remote program. The *ret* value is not available until a **parwait** or **parsync** call has been executed for the *prog_id* (see the descriptions of these functions). *rrs* is a list of resource requirements for the subprogram. This parameter is used by DCS to determine a location for the remote execution of the subprogram. An execution requirement has a *type* and an *instance* .

```
        struct rec_req_spec {
                int     type;           /* type are defined in dcs_user.h */
                char    *instance       /* instances are type dependent */
        } rrs[ ];
```

The *rrs* list is a vector terminated by an entry with a *type* of 0. Thus to insist that a subprogram be started on a machine named **molson** the following *rec_req_spec* would be used.

```
        rrs[0].type = MACH_NAME;
        rrs[0].instance = "molson";
        rrs[1].type = 0;
```

If no special resources are needed (the default is to place the program on the SUN3 with the least load), then a 0 may be specified for the *rrs* parameter.

RETURN VALUE

Upon successful completion the remote program identifier, *prog_id* , for the scheduled program is returned, this value may be used in subsequent calls to **parwait, parsync,** and **contin.** If a negative number is returned an error has occurred, consult *errno.*

SEE ALSO

parwait(3), parsync(3), contin(3), raccept(3), rreturn(3), syncp(3)

FILES

/usr/etc/cohorts	static requirements file for network machines.

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

openseg, sharedvar, sharedvec, closeseg - declare shared variables.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

openseg(seg_id);

sharedvar(x_type, x, upt);
sharedvec(y_type, y, length, upt);

closeseg(seg_id);
```

DESCRIPTION

Declarations of shared variables appear as extensions to C; they are implemented as macros to the C preprocessor.

Openseg begins the definition of the segment and uses the integer constant *seg_id* to identify the segment. This declaration must appear before any shared memory declarations (i.e. *sharedvar* and *sharedvec)* for this segment. This *seg_id* is used by each program in the distributed computation to identify the same segment.

Sharedvar declares the non-vector variable *x* to be of type *x_type* and allocates space for it in the shared memory segment specified by the previous *openseg* . *Sharedvec* declares the variable *y* to be a vector with length *length* and of type *y_type* and allocates space for it in the shared memory segment specified by the previous *openseg*. *x_type* and *y_type* may be any valid C types. If that type is **struct** (or **union**) then only **struct** (or **union**) followed by the tag field appears in the macro. The *upt* parameter is an integer that specifies the update discipline to use for this variable. The possible values for *upt* are defined in the include file *"/usr/local/include/dcs_user.h"*. Shared memory is replicated, one copy of a shared segment to each machine running a subprogram in the distributed computation. The update discipline governs how the DCS system will perform updates to the different copies of a variable.

UPDT_UNSAFE - all new values are written without any synchronization.
UPDT_UNREST - all new values are written using only local synchronization.
UPDT_GLOBAL - all new values are written using global synchronization.

The UPDT_UNSAFE discipline allows very efficient access to shared storage, however provides no protection against two processes simultaneously accessing the same piece of storage. This discipline is appropriate for situations in which processes will only be reading the segment once it has an initial value. The UPDT_UNREST discipline insures that only one process will access a particular copy of the shared variable. However since the variable is replicated on each machine, simultaneous updates may occur and thus the various copies may differ. This discipline is appropriate when only one process is writing the variable (or vector element) and other processes are reading the result. The UPDT_GLOBAL discipline insures that only one modification to the variable occurs at a time throughout the computation. This discipline makes the shared variables behave exactly as if physical shared memory protected by semaphores were being used.

The only access to shared variables is through the *aread, awrite,* and *aupdate* routines, all other references have undefined results.

The *closeseg* macro terminates the mapping of a shared memory segment. The *seg_id* must be identical to the *seg_id* specified in the previous *openseg* declaration. The declaration of shared variables is not complete until matching *openseg/closeseg* statements have been encountered.

There may be multiple occurrences of *sharedvar* and *sharedvec* between an *openseg* and its corresponding *closeseg* . A single program may declare multiple shared segments, each with a unique *seg_id* .

SEE ALSO

aread(3), awrite(3), aupdate(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

parsync, parsync_v - synchronize with a set of subprograms.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

status = parsync(prog_id0, prog_id1, ..., prog_idn,0)
int status, prog_id0, prog_id1, ..., prog_idn

status = parsync_v(prog_id_v, n)
int prog_id_v[],n
```

DESCRIPTION

Parsync returns when all subprograms identified by *prog_id0, prog_id1, ..., prog_idn* have executed *syncp* . These *prog_ids* must be values returned by previous **doprog** calls. The values computed by the programs (i.e. the **doprog** *ret* parameter) are available after the *parsync* has returned, if the programs executing *syncp* provided return parameters.

Parsync_v is an alternate calling mechanism, allowing the *prog_id s* to be elements of a vector, *prog_id_v* , with *n* specifying the length of the vector.

RETURN VALUE

Upon completion a 0 is returned. A non-zero, positive return value indicates the number of subprograms that did not synchronize normally (e.g. because of early termination). A negative return values indicates a failure in the processing of the *parsync* call, consult *errno*.

SEE ALSO

doprog(3), parwait(3), contin(3), syncp(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

parwait, parwait_v - wait for the completion of a set of remote programs.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

status = parwait(prog_id0, prog_id1, ..., prog_idn,0)
int status, prog_id0, prog_id1, ..., prog_idn

status = parwait_v(prog_id_v, n)
int prog_id_v[],n
```

DESCRIPTION

Parwait returns when all subprograms identified by *prog_id0, prog_id1, ..., prog_idn* have tc These *prog_id*'s must be values returned by previous **doprog** calls. The values computed by (i.e. the **doprog** *ret* parameter) are only available after the *parwait* has returned, provided thc ing subprogram executed an **rreturn.**

Parwait_v is an alternate calling mechanism, allowing the *prog_id*'s to be elements of a vectc with *n* specifying the length of the vector.

RETURN VALUE

Upon completion 0 is returned if all subprograms completed without errors. Any other value cates an error condition. A non-zero, positive return value indicates the number of subprogra not terminate normally. A negative return values indicates a failure in the processing of the pc Consult *errno* for further information.

SEE ALSO

doprog(3), rreturn(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

raccept - accept remote parameter.

SYNOPSIS

#include "/usr/local/include/dcs.h"

raccept((char*) &arg, sizeof(arg))
type1 arg

DESCRIPTION

This routine is called by a subprogram of a distributed computation to accept the parameter passed by the calling program. *arg* is the variable to receive the argument specified as *arg* on the **doprog** call that started the execution of this subprogram. In order to insure proper alignment and addressing in the user program, the type, *type1*, of *arg* must match the type of the corresponding parameter on the *doprog* call.

DIAGNOSTICS

If the routine is not successful in obtaining the argument then *raccept* will cause the process to exit. This failure will be communicated to the calling program via the *status* variable of the **doprog** call.

SEE ALSO

doprog(3), parwait(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

rprintf - print messages from a remote process to the controlling terminal.

SYNOPSIS

```
#include <stdio.h>
#include  "/usr/local/include/dcs.h"

int rprintf(format[,arg] ... )
char *format;
```

DESCRIPTION

Rprintf prints to the standard output stream, **stdout**, of the main program of a distributed computation. The standard output stream must be a terminal or the output is ignored. The message may be no longer than 132 characters. Otherwise, this function behaves exactly as *printf*.

SEE ALSO

printf(3S)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

rreturn - remote return.

SYNOPSIS

#include "/usr/local/include/dcs.h"

rreturn((char*) &ret, sizeof(ret))
type2 ret

DESCRIPTION

This routine is called by a subprogram of a distributed computation to return the output variable to the calling program. The calling program receives *ret* in the variable specified by *ret* on the *doprog* call which caused the execution of this subprogram. The type, *type2*, of *ret* must match the type of the corresponding variable on the *doprog* call to insure proper addressing and alignment.

This function exits, it does not return control to the caller.

SEE ALSO

doprog(3), parwait(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO

NAME

syncp - synchronize with the parent program.

SYNOPSIS

```
#include "/usr/local/include/dcs.h"

syncp((char*) &ret, sizeof(ret))
type2 ret
```

DESCRIPTION

This routine is called by a subprogram of a distributed computation to synchronize with, and return the output variable to, the main program. The calling subprogram is suspended at the *syncp* until the parent program executes a *parsync* call followed by a *contin* call, both specifying this subprogram. The parent program receives *ret* in the variable specified by *ret* on the *doprog* call which caused the execution of this subprogram. The type, *type2*, of *ret* must match the type of the corresponding variable on the *doprog* call to insure proper addressing and alignment.

RETURN VALUE

0 is returned for successful completion, otherwise -1 is returned.

SEE ALSO

doprog(3), parsync(3), contin(3)

AUTHORS

Paul Maybee
Paul K. Harter
David C. M. Wood
University of Colorado, Boulder CO