User-Level Shared Variables (in a Hierarchical Control Environment) Don Libes National Bureau of Standards Factory Automation Systems Division Gaithersburg, MD 20899 We have implemented a shared variable system for 4.2BSD [Joy] that emulates a shared or common memory. This system is all user level code and requires no kernel modifications. It is accessible from a variety of languages. This shared memory system provides its service transparently to remote hosts across a local-area network. This system is being used in a real application - an automated factory. Shared variables are an appropriate communication paradigm in a real-time environment (in comparison to message passing or pipes). Background The National Bureau of Standards has been performing research in the area of robot control [Albus] for over a decade. Robot control is typified by tightly-coupled multiprocessor systems often implemented as multiple single-board computers residing in a common bus. These systems require frequent and fast transfer of small amounts of data. For example, a robot arm may require new joint angles 100 times a second in order to move smoothly. These joint angles are very small pieces of data and further, the number of joints and joint angles is constant, hence they can be stored in "well-known" locations. To support this type of application, multiported memory is used so that several different computers can access the same locations. In the robot example, the controlling computer and the servo computers each need access to the information. Common memory has the further advantage that as new processes are added that need information already present, extant processes do not have to be modified to deliver that information. For example, a process was added that displays the robot's actions on a graphics monitor. The display process was added without modification to any other part of the system, since it uses the joint angles which are stored in well-known common memory locations. Recently a "safety" process was added to guarantee that the robot never departs its working envelope. The safety process also obtains its information from common memory. In 1981, work [Furlani] began on an automated factory. This is an extension of the robot control system in many ways. It is expected that the factory model will grow over time, and become richer in the number of processes used and the amount of information shared between processes. A major difference in the automated factory project is that systems which have to communicate are often in separate backplanes and use different operating systems. Common memory is seen as a consistent communications methodology for this disparate collection of computers. Today, as computers grow larger and become more complex, more and more layers of protocol prevent ready access to common memory. Indeed, we find that in most of our computer-to-computer interfaces, common memory is now simply a way of thinking, rather than an underlying implementation. Such is the case with 4.2BSD. The following section describes an implementation of the common memory paradigm in the 4.2BSD environment. I. User View A server, the Common Memory Manager (CMM), acts on requests to access shared variables. Typical requests are read, write, and declare. Variables are structured and have attributes such as type, protection, lifetime, and ownership. Variable type checking is performed at runtime startup, since processes are loosely connected. Some of the more interesting attributes of these shared variables are: lifetime and "shelf life". In a real time environment, the system must survive a process dying or getting bogged down temporarily. When the useful lifetime of a variable's value has expired, other processes are free to manipulate the variable, for instance, by redeclaring or writing it. This attribute allows a "god" process to notice unexpected process deaths and restart processes or transfer control to other processes. For less critical variables (such as an infrequently updated sensor value), processes can note that a value is "stale", and make a projection that will carry them over until the variable writer catches up to its duties. write count, timestamp, authorstamp. By marking each variable with how many times or when it has been written, handshaking can be performed between processes. The archetypal example from the automation environment has a supervisor sending the command "hit nail with hammer" to a subordinate. If the supervisor runs more quickly than the subordinate, without handshaking, the subordinate may never see the command (i.e. nail is not hit). If the subordinate runs more quickly than the supervisor, without handshaking, the subordinate may execute the same command twice (i.e. nail is hit more than once). The same problem exists when the supervisor reads the feedback from the subordinate. In particular, because there is no synchrony between processes, the supervisor has no way of knowing what command the subordinate is responding to (without handshaking). Actual code for such a sequence would use higher-level routines to read the shared variable. For example:

```
struct shared_variable *command_out, *feedback_in;
struct shared_variable_value *expected_feedback;
.
.
if (status_equal(command_out, feedback_in, expected_feedback)) . . .
```

Here, the command to the subordinate is The actual feedback received is and we are looking for The expected feedback is thus compared to the feedback actually received with respect to the command which generated it. These handshaking and synchronization problems are thoroughly discussed in [Libes]. wakeup. Variables can be marked with a list of processes to be notified upon value change. This was originally added for efficiency, so that clients, for example, would not have to loop just waiting for new values. It has turned out to be extremely useful for debugging. Rather than inserting print statements in existing code and recompiling, it is possible to create a debug module that simply prints out variables when they change. It has also been used for a graphics monitor that continuously displays the state of the entire system. The wakeup attribute can be used to force a queuing discipline on a variable. For example, a process performing as a resource server may take requests from a well-known common memory variable. As clients write to the variable, the CMM wakes up the resource server with each new variable value. The resource server typically responds to each variable value in turn; however it can ignore all but the most recent values if it desires. non-exclusive write. Variables can be declared read or write, exclusively or nonexclusively. By default, writes are exclusive, meaning that only one process can write a variable during its lifetime. Non-exclusive write has no such restriction, allowing multiple processes to write the same variable. One use of this might be a server listening at a well-known location ("socket") for a request for service. Any client can request service by writing the variable associated with the socket. Another use is to send information to a log file. In our application, we can ask all processes to log their states a given number of clock ticks. By having them all write their states to one variable, a logging process can record all the states of the system through time. This has proven to be invaluable during system testing. II. 4.2BSD Implementation The Shared Variables System (SVS) is written in C and uses the 4.2 interprocess communication system [Leffler]. An interface exists for Franz Lisp. The SVS consists of three layers. The lowest level simulates a common memory facility such as is available in System V. On top of this are the procedures that transform common memory into common variables. This level provides users with routines allowing access and control of the common variables. Specific to our use of the SVS is a higher level that customizes it for communication in a hierarchical control environment. This level includes interprocess handshaking. Building a server on 4.2 4.2BSD has no common memory facility. Our system provides one via a server that stores variables in its own memory space which is private to it. (See Figure 1) The server runs as a user-level process which needs no special privileges and can be run from an unrelated user-id.

Figure 1 goes here

Communications between the common memory server and clients have properties of both streams and datagrams. Communication resembles virtual circuits in that the server and clients pass many messages between them in a conversation (like TCP). These messages are also record-oriented (like UDP). In implementation, UDP was rejected because it would have forced us to handle unreliability and fragmentation (datagrams were limited to 1K which was smaller than the typical message). In this case, we chose TCP as the lesser of two evils. TCP had the drawback (for us) of removing record boundaries (i.e. the receiver could not tell where one write ended and the next began). To handle this, new versions of read and write were constructed to do the actual I/O to the sockets. These packetized and depacketized the TCP streams back into records. Since TCP/IP is capable of communicating across the network, clients may be distributed (figure 2) while accessing the common memory. However, the small number of concurrent TCP connections (viz. the number of open files) is a severe limit. Another system [Mitchell] glues distributed common memories together throughout the factory. This design allows multiple common memories to exist but appear to the user as one. This gluing process bypasses the TCP restriction and allows multiple instances of the 4.2 (as well as others') common memory service to be run.

Figure 2 goes here

Simulating common memory on 4.2 The server's own private memory serves as common memory. Access to variables, therefore, is strictly through the valid requests to the CMM server. The server accesses variables as requested and maintains extra information such as variable type, owner, etc. Whenever a client wants to access common memory, it does so by sending a message to the CMM. Such messages are sent asynchronously but received synchronously. In particular, the server is never interrupted. Typically, the server is waiting for service requests. When a request is received, service is performed. This service may or may not cause sending of a message in return to the client. For example, "read" operations always cause information to be returned from the server while "write" operations don't (unless an error is encountered). It is up to the client whether it wishes to wait (even after a "read") for a response from the common memory server. Client processes keep their own local version of shared variables of interest until it is convenient for them to synchronize with the CMM. By calling the user and the common memory server become synchronized. An argument to specifies whether the client waits for a response or not. Whether the client is waiting or not, an error can occur at the server end, in which case an error message is returned. For example, a client may write a value but the variable may be an incompatible type. When this happens, a message is sent back to the client. The client is not interrupted, but at the next operation it is notified of the error. Interestingly, the problem of being able to provide atomic updates to common memory vanishes in this system. Since the server is never interrupted, client updates functionally prevent all other clients access to common memory while their request is being serviced. This removes the classic synchronization problem which exists when multiple processes are reading and writing real common memory with no synchronization. On the other hand, this can cause relatively long waits for service. Construction of higher-level synchronization such as semaphores is easy. Potentially stalling manipulations of semaphores and monitors can call and then wait for a response from the server to continue. The server acknowledges the client if and only if the client semaphore action is satisfied. III. An application - manufacturing automation While the SVS is a general tool, it is currently being used at the National Bureau of Standards in the Automated Manufacturing Research Facility (AMRF), a testbed for research in the automation of small batch manufacturing [Simpson]. The project is funded by NBS and the Navy Manufacturing Technology Program and significantly supported by industry through donations or loans of major components and through cooperative research programs. The objective of the project is to identify, implement and exercise potential standard interfaces between existing and future components of small-batch manufacturing systems and to provide a laboratory for the development of factory-floor metrology in an automated environment, delivering proven measurement techniques and standard reference materials to American industry. Commercially available products are being used to construct the facility wherever possible in order to expedite transfer of research results into the private sector. To provide a real testbed for interface standards, the AMRF is intentionally composed of manufacturing and computer equipment from many vendors, thereby making its construction a major integration effort. The types of systems that must be integrated and communicate with each other include stationary and mobile robots, machine tools, their controllers, and higher-level cell and factory management systems. Actual computers and operating systems run the gamut, from micros to mainframes. Processes in the AMRF are organized in a control hierarchy, much like the organization of a conventional factory. At each level, tasks are broken down into simpler tasks that can be performed by subordinate processes. Control is hierarchical. To support such control communication, we imagine that processes in the system communicate along hierarchical channels. Each process runs asynchronously, continually "sending" commands to subordinate processes and feedback to superior processes. Logically however, communication is performed using the common memory model. For example, a process sending commands to a subordinate writes the information in a well-known common memory variable. The subordinate reads the information out of the common memory variable, noting whether or not it contains a new value. In reality, there are small groups of processes and processors that directly communicate with each other, some using physical common memory and some simulating common memory. Each cluster representing a piece of common memory can choose to replicate any other portion of another cluster's common memory. An underlying network (figure 3) invisibly supplies this service of keeping all the common memories synchronized. The result is effectively one of a consolidated common memory, with a very consistent and easy means of communication between processes.

Figure 3 goes here

Summary Common memory is a communications methodology that is easy to use and can be efficiently implemented. It can use physical common memory or be simulated by other communications techniques such as message passing. In turn, it can be used to implement any other communications techniques. Common memory has several important advantages and disadvantages. The primary advantage is the ease with which information may be shared. In our application, we have required this richness of data sharing between processes. Using common memory also has the benefit of being able to add, remove or modify processes without disturbing other processes, even if they communicate with each other. The primary disadvantage is that sharing large amounts of information in this manner can become hard to keep track of. Since there is no scoping in the common memory, naming conventions must be followed (which in effect, provides scoping). Common memory is not particularly apt for all types of communications. Picking the most appropriate model (pipes, databases, message passing) will invariably lead to a simpler process in the final analysis. We, of course, use these other techniques in our application when appropriate. At the lowest level of our application, efficient process-to-process communication is necessary for real-time behavior. This is implemented using real common memory with low overhead. At higher levels where common memory is simulated and the high-speed demand does not exist, the methodology extends upwards suitably. In the future, we expect to move the ideas of hierarchical control and common memory to new applications. Currently, we are investigating the potential for control of autonomous aircraft. References Albus, J., Barbera, T., Fitzgerald, M.L., "Hierarchical Control for Sensory Interactive Robots", Proceedings of 31st General Assembly, International Institution for Production Engineering Research (CIRP), Toronto, Canada, September, 1981. Furlani, C., Kent, E., Bloom, H., McLean, C., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Proceedings of the Summer Simulation Conference, Vancouver, B.C., Canada, July 1983. Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K., Mosher, D., "4.2BSD System Interface Overview", Computer Systems Research Group, U.C. Berkeley, July, 1983. Leffler, S., Fabry, R., Joy, W., "4.2BSD Interprocess Communications Primer", Computer Systems Research Group, U.C. Berkeley, July, 1983. Libes, D., "Handshaking in a Hierarchical Control Environment", Internal Memorandum, Center for Manufacturing Engineering, National Bureau of Standards, Washington, DC 20234. Mitchell, M., Barkmeyer, E., "Data Distribution in the NBS Automated Manufacturing Research Facility", Center for Manufacturing Engineering, National Bureau of Standards, Washington, DC 20234, 1984. Simpson, J., Hocken, R., Albus, J., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Journal of Manufacturing Systems, Vol. 1, #1. Further Reading Bloom, H., Furlani, C., Barbera, A., "Emulation as a Design Tool in the Development of Real-Time Control Systems", 1984 Winter Simulation Conference, Dallas, Texas, November 1984. Johnson, T., Milligan, S., Fortmann, T., Bloom, H., McLean, C., Furlani, C. "Emulation/Simulation of a Modular Hierarchical Feedback System", The 21st IEEE Conference on Decision and Control, Orlando, FL, December 1982.