Process Migration in DEMOS/MP

Michael L. Powell
Barton P. Miller

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Process migration has been added to the DEMOS/MP operating system. A process can be moved during its execution, and continue on another processor, with continuous access to all its resources. Messages are correctly delivered to the process's new location, and message paths are quickly updated to take advantage of the process's new location. No centralized algorithms are necessary to move a process.

A number of characteristics of DEMOS/MP allowed process migration to be implemented efficiently and with no changes to system services. Among these characteristics are the uniform and location independent communication interface, and the fact that the kernel can participate in message send and receive operations in the same manner as a normal process.

This research was supported by National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

1. Introduction

Process migration has been discussed in the operating system literature, and has been among the design goals for a number of systems [Finkel 80][Rashid & Robertson 81]. Theoretical and modeling studies of distributed systems have suggested that performance gains are achievable using relocation of processes [Stone 77, Stone & Bokhari 78, Bokhari 79, Robinson 79, Arora & Rana 80]. Process migration has also been proposed as a tool for building fault tolerant systems [Rennels 80]. Nonetheless, process migration has proved to be a difficult feature to implement in operating systems.

As described here, ***process migration*** is the relocation of a process from the processor on which it is executing (the ***source*** processor) to another processor (the ***destination*** processor) in a distributed (loosely-coupled) system. A loosely-coupled system is one in which the same copy of a process state cannot directly be executed by both processors. Rather, a copy of the state must be moved to a processor before it can run the process. Process migration is normally an involuntary operation that may be initiated without the knowledge of the running process or any processes interacting with it. Ideally, all processes continue execution with no apparent changes in their computation or communications.

One way to improve the overall performance of a distributed system is to distribute the load as evenly as possible across the set of available resources in order to maximize the parallelism in the system. Such resource load balancing is difficult to achieve with static assignment of processes to processors. A balanced execution mix can be disturbed by a process that suddenly requires larger amounts of some resource, or by the creation of a new process with unexpected resource requirements. If it is possible to assess the system load dynamically and to redistribute processes during their lifetimes, a system has the opportunity to achieve better overall throughput, in spite of the communication and computation involved in moving a process to another processor [Stone 77, Bokhard 79]. A smaller relocation cost means that the system has more opportunities to improve performance.

System performance may also be improved by reducing inter-machine communication costs. Accesses to non-local resources require communication, possibly through intermediate processors. Moving a process closer

to the resource it is using most heavily may reduce system-wide communication traffic, if the decreased cost of accessing its favorite resource offsets the possible increased cost of accessing its less favored ones.

A static assignment to a processor may not be best even from the perspective of a single program. As a process runs, its resource reference pattern may change, making it profitable to move the process in mid-computation.

The mechanisms used in process migration can also be useful in fault recovery. Process migration provides the ability to stop a process, transport its state to another processor, and restart the process, transparently. If the information necessary to transport a process is saved in stable storage, it may be possible to "migrate" a process from a processor that has crashed to a working one. In failure modes that manifest themselves as gradual degradation of the processor or the failure of some but not all of the software, working processes may be migrated from a dying processor (like rats leaving a sinking ship) before it completely fails.

Process migration has been proposed as a feature in a number of systems [Solomon & Finkel 79, Cheriton 79, Feldman 79, Rashid & Robertson 81], but successful implementations are rare. Some of the problems encountered relate to disconnecting the process from its old environment and connecting it with its new one, not only making the new location of the process transparent to other processes, but performing the transition without affecting operations in progress. In many systems, the state of a process is distributed among a number of tables in the system making it hard to extract that information from the source processor and create corresponding entries on the destination processor. In other systems, the presence of a machine identifier as part of the process identifier used in communication makes continuous transparent interaction with other processes impossible. In most systems, the fact that some parts of the system interact with processes in a location-dependent way has meant that the system is not free to move a process at any point in time.

In the next section, we will discuss some of the structure of DEMOS/MP, which eliminates these impediments to process migration. In subsequent sections, we will describe how a process is moved, how the communication system makes the migration transparent, and the costs involved in moving a process.

2. The Environment: DEMOS/MP

Process migration was added to the DEMOS/MP [Powell, Miller, & Presotto 83] operating system. DEMOS/MP is a version of the DEMOS operating system [Baskett, Howard, & Montague 77, Powell 77] the semantics of which have been extended to operate in a distributed environment. DEMOS/MP has all of the facilities of the original uni-processor implementation, allowing users to access the multi-processor system in the same manner as the uni-processor system.

DEMOS/MP is currently in operation on a network of Z8000 microprocessors, as well as in simulation mode on a DEC VAX running UNIX. Though the processor, I/O, and memory hardware of these two implementations are quite different, essentially the same software runs on both systems. Software can be built and tested using UNIX and subsequently compiled and run in native mode on the microprocessors.

2.1. DEMOS/MP Communications

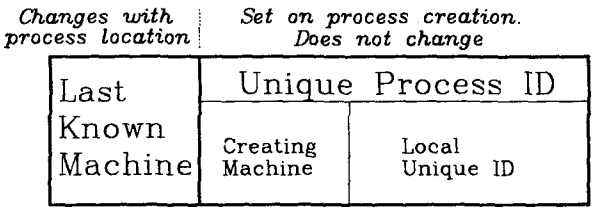
DEMOS/MP is a message-based operating system, with communication as the most basic mechanism. A kernel implements the primitive objects of the system: executing processes, messages, including inter-processor messages, and message paths, called links. Most of the system functions are implemented in server processes, which are accessed through the communication mechanism.

All interactions between one process and another or between a process and the system are via communication-oriented kernel calls. Most system services are provided by system processes that are accessed by message communication. The kernel implements the message operations and a few special services. Messages are sent to the kernel to access all services except message communication itself.

A copy of the kernel resides on each processor. Although each kernel independently maintains its own resources (CPU, real memory, and I/O ports), all kernels cooperate in providing a location-transparent, reliable, interprocess message facility. In fact, different modules of the kernel on the same processor, as well as kernels on different processors, use the message mechanism to communicate with each other.

In DEMOS/MP, messages are sent using *links* to specify the receiver of the message. Links can be thought of as buffered, one-way message channels, but are essentially protected global process addresses accessed via a local name space. Links may be created, duplicated, passed to other processes, or destroyed. Links are manipulated much like capabilities; that is, the kernel participates in all link operations, but the conceptual control of a link is vested in the process that the link addresses (which is always the process that created it). Addresses in links are context-independent; if a link is passed to a different process, it will still point to the same destination process. A link may also point to a kernel. Messages may be sent to or by a kernel in the same manner as a process.

The most important part of a link is the message process address (see Figure 2-1). This is the field that specifies to which process messages sent over that link are delivered. The address has two components. The first is a system-wide, unique, process identifier. It consists of the identifier of the processor on which the process was created, and a unique local identifier generated by that machine. The second is the last known location of the process. During the lifetime of a link, the first component of its address never changes; the second, however, may.



Structure of a process address
Figure 2-1

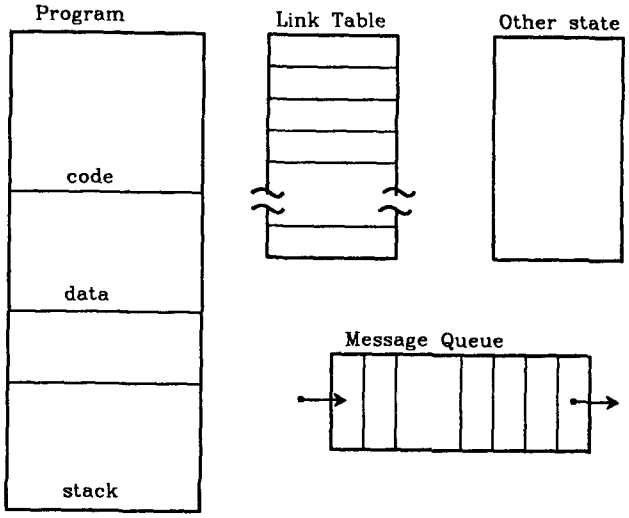
2.2. Special Kernel Communications

A link attribute, called *DELIVERTOKERNEL*, causes the link to reference the kernel of the processor on which a particular process resides. Except for the DELIVERTOKERNEL flag, a link with this attribute looks the same as a link to the process to which it points. Links with the DELIVERTOKERNEL attribute used to cause the kernel to manipulate the process in ways that system processes cannot do directly.

A message sent over a DELIVERTOKERNEL link follows the normal routing to the process. However, on arrival at the destination process's message queue, the message is received by the kernel on that processor. A link with the DELIVERTOKERNEL attribute allows the system to address control functions to a process without worrying about which processor the process is on (or is moving to).

This mechanism has simplified a number of problems associated with moving a process. It is often the case that some part of the system needs to manipulate the state of a process, for example, the process manager may wish to suspend a process. Using a link with the DELIVERTOKERNEL attribute, the process manager can send a message to the process's kernel asking that the process be stopped. If the process is temporarily unavailable to receive the message (for instance, it is in transit during process migration), the message is held and forwarded for delivery when normal message receiving can continue.

In addition to providing a message path, a link may also provide access to a memory area in another process. When a process creates a link, it may specify in the link read or write access to some part of its address space. The process holding the link may use kernel calls to transfer data to or from the data area defined by the link. This is the mechanism for large data transfers, such as file accesses or data transfer in process migration. The kernel implements the data move operation by sending a sequence of messages containing the data to be transferred. These messages are sent over a DELIVERTOKERNEL link to the kernel of process containing the data area. Using DELIVERTOKERNEL links allows the data to be read from or written to the kernel of the remote process without the kernel that instigated the operation being aware of the process's location.



Components of a DEMOS/MP process
Figure 2-2

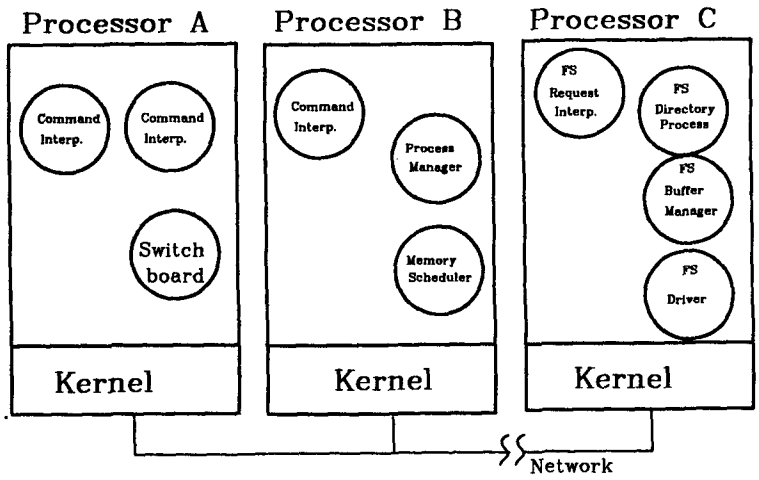
The inter-machine communication of DEMOS/MP provides reliable delivery of messages. The fundamental guarantee is that any message sent will eventually be delivered.

A DEMOS/MP process is shown in Figure 2-2. A process consists of the program being executed, along with the program's data, stack, and state. The state consists of the execution status, dispatch information, incoming message queue, memory tables, and the process's link table. Links are the only connections a process has to the operating system, system resources, and other processes. Thus, a process's link table provides a complete encapsulation of the execution of the process.

2.3. DEMOS/MP System Processes

DEMOS/MP *system* processes are those processes assumed to be present at all times. *User* processes are created dynamically to perform computation, usually at the request of some user. A system process will often be a *server* process, that is, most other processes will be able to ask it to perform some functions on their behalf. The system processes being used in DEMOS/MP are the switchboard, process manager, memory scheduler, file system (actually, four processes), and command interpreter. The switchboard is a server that distributes links by name. It is used by the system and user processes to connect arbitrary processes together. An example of the system process structure is shown in Figure 2-3.

The process and memory managers handle all the high-level scheduling decisions for processes. These processes allocate and keep track of usage for system resources such as the CPU, real memory, etc. They control processes by sending messages to kernels to manipulate process states. For example, although the kernel implements the mechanisms of migrating a process, the process manager makes the decision of when and to where to migrate a process.



Example of system process lay-out
Figure 2-3

The file system is the same as that implemented for the uni-processor DEMOS [Powell 77], with the added freedom that the file system processes can be located on different processors. The command interpreter allows interactive access to DEMOS/MP programs.

One of our test examples of process migration runs the above processes. It migrates a file system process while several user processes are performing I/O. This is more difficult than moving a user process would be, as we shall see below.

2.4. The Long and Short of Links

It is important to consider all the places where links to a process might be stored when that process is moved, since they contain information specifying the location of the process. Although a link is not useful after the process that it addresses terminates, some links last for relatively long periods of time. For example, a *request* link, which represents a service such as process management, or a *resource* link, which represents an object such as an open file, may exist for as long as the system is up, if they are held by a system process. Other links, such as *reply* links, have short lifetimes, since they are used only once to respond to requests.

Links may be either in some process's link table or in a message that is enroute to a process. Once a link is given out, it may be passed to other processes without the knowledge of the process that created the link (the process to which the link points). There is no way short of a complete system search of finding all links that point to a process. The mechanism for handling messages during and after a process is migrated must provide a way for messages to be directed to the new location, despite out-of-date links. Moreover, for performance reasons, it should eventually bring these links up-to-date.

Moving a *user* process will usually be simple. The only processes likely to have links to a user process are system processes. Such links may be used to send only one message, so the out-of-date link will no longer exist after forwarding the reply message to the new location of the user process.

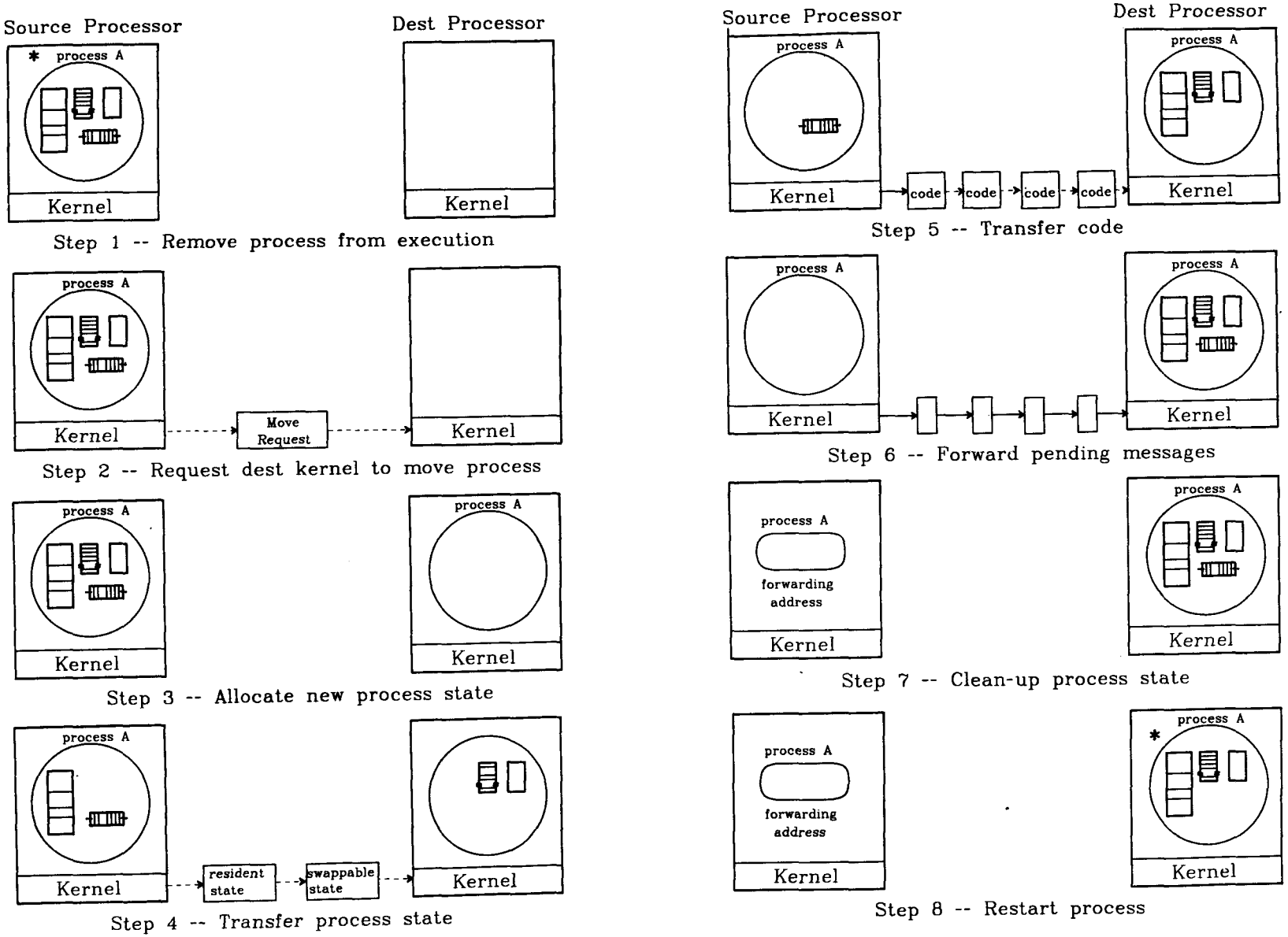
Moving a *system* process (or, more precisely, a *server* process), is more difficult, since many processes may have links to it, and such links may last a long time, being duplicated and passed to other processes. In fact, the server process may not know how many copies of links there are to it (it is possible, but optional, for a process to keep track of how many, but not where they are). Since such links may be used for many messages, performance considerations will require a method for updating these links.

3. Moving a Process

Most of the low-level mechanisms required to manipulate the process state and move data between kernels were available in the version of DEMOS/MP that existed when this effort began. Process migration was implemented by using those facilities to move the process, and adding the mechanisms for forwarding messages and updating links.

3.1. The Mechanism

A process is moved between two processors called the *source processor* and the *destination processor*. A request to the kernel to move a process is made by the process manager system process. In the absence of an authentic workload for our test cases, the decision to move a particular process and the choice of destination were arbitrary. However, adding a decision rule for when and to where to move a process will be easy. The process manager and memory scheduler already monitor system



Steps in moving a process
Figure 3-1

activity for memory and cpu scheduling, and can use the same information to make process migration decisions. Information on the communications load is also available. It is of course possible for a process to request its own migration. This request can be thought of as one more piece of information that the process manager can use in making migration decisions. Designing an efficient and effective decision rule is still an open research topic.

There are several features of a decision rule that we have considered in our implementation. The migration scheme depends on the ability to evaluate the resource use patterns of processes. This function is normally available in the accounting or performance monitoring part of the system. There must also be a way to assess the load on individual processors. This function is often available in systems with load-limiting schedulers, which activate or deactivate processes based on overall system load. The three features not usually available are the means to collect the above information in one place, an strategy for improving the operation of the system considering the appropriate costs, and a hysteresis mechanism to keep from incurring the cost of migration more often than justified by the gains.

Information used to determine when and where to move a process involves the state of machine on which the process currently resides, and machines to where the process could move. Processor loading and memory demand for each machine is required.

More difficult is integrating the communications cost incurred by a process. Processes cooperating in a computation may exhibit a great deal of parallelism, and therefore should be on different machines. However, separating them could increase the latency of communication beyond the savings accrued by parallel execution. Collection of the communication data is beyond the ability of most current systems.

Once the decision has been made to migrate a process, the following steps are performed (shown in figure 3-1).

1. Remove the process from execution:

   The process is marked as "in migration". If it had been ready, it is removed from the run queue. No change is made to the recorded state of the process (whether it is suspended, running, waiting for message, etc.), since the process will (at least initially) be in the same state when it reaches its the destination processor. Messages arriving for the migrating process, including DELIVERTOKERNEL messages, will be placed on its message queue.

2. Ask destination kernel to move process:

   A message is sent to the kernel on the destination processor, asking it to migrate the process to its machine. This message contains information about the size and location of the the process's resident state, swappable state, and code. The next part of the migration, up to the forwarding of messages (Step 6), will be controlled by the destination processor kernel.

3. Allocate a process state on the destination processor:

   An empty process state is created on the destination processor. This process state is similar to that allocated during process creation, except that *the newly allocated process state has the same process identifier as the the migrating process.* Resources such as virtual memory swap space are reserved at this time.

4. Transfer the process state:

   Using the move data facility, the destination kernel copies the migrating process's state into the empty process state.

5. Transfer the program:

   Using the move data facility, the destination kernel copies the memory (code, data, and stack) of the process into the destination process. Since the kernel move data operation handles reading or writing of swapped out memory and allocation of new virtual memory, this step will cause definition of memory to take place, if necessary. Control is returned to the source kernel.

6. Forward pending messages:

   Upon being notified that the process is established on the new processor, the source kernel resends all messages that were in the queue when the migration started, or that have arrived since the migration started. Before giving them back to the communication system, the source kernel changes the location part of the process address to reflect the new location of the process.

7. Clean-up process's state:

   On the source processor, all state for the process is removed and space for memory and tables is reclaimed. A *forwarding address* is left on the source processor to forward messages to the process at its new location. The forwarding address is a degenerate process state, whose only contents are the (last known) machine to which the process was migrated. The normal message delivery system tries to find a process when a message arrives for it. When it encounters a forwarding address, it takes the actions described in the next section. The source kernel has completed its work and control is returned to the destination kernel.

8. Restart the process:

   The process is restarted in whatever state it was in before being migrated. Messages may now arrive for the process, although the only part of the system that knows the new location of the process is the source processor kernel. The destination kernel has completed its work.

At this point, the process has been migrated. The links from the migrated process to the rest of the system are all still valid, since links are context-independent. Links created by the process after it has moved will point to the process at its new location. The only problem is what to do with messages sent on links that still point to the old location.

3.2. A Note on Autonomy and Interdomain Migration

The DEMOS/MP kernels trust each other, and thus are not completely autonomous. Moreover, for practical purposes, all DEMOS/MP processors are identical and provide the same services. This makes process migration particularly useful in our environment. However, the process migration mechanism could work even if the kernels were autonomous and had different resources.

The crucial questions for autonomous processors are "Is the process willing to be moved?" and "Will the destination machine accept it?" Any policy to decide which process to migrate could take into account the former question. The second question can be addressed during the migration. Note that the destination machine actually performs most of the steps. In particular, in Step 2, the source machine asks the destination machine to accept the process. If the destination machine refuses, the process cannot be migrated.

It is also possible to migrate processes between domains. By domain, we mean that the destination processor belongs to a collection of machines under a different administrative control than the source processor, and may be suspicious of the source processor and the incoming process. The destination processor may simply refuse to accept any migrations not fitting its criteria. The source processor, once rebuffed, has the option of looking elsewhere.

The source and destination kernels must, of course, be able to communicate with each other in order to accomplish the migration, and the destination machine must be able to handle messages sent over the links held by the process. Since the ability to send and receive messages over links is all a DEMOS process expects of its environment, so long as that continues to be provided, the process can continue to run.

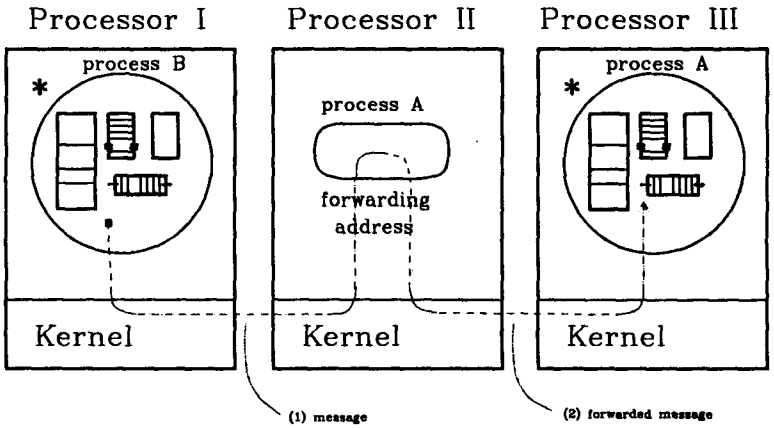
4. Message Forwarding

Since DEMOS/MP guarantees message delivery, in moving a process it must be ensured that all pending, enroute, and future messages arrive at the process's new location. There are three cases to consider: messages sent but not received before the process finished moving, messages sent after the process is moved using an old link, and messages sent using a link created after the process moved.

Messages in the first category were in process's message queue on the source machine, waiting for the process to receive them, when the process restarted on the destination machine. These messages are forwarded immediately as part of the migration procedure.

Messages in the middle category are forwarded as they arrive. After the process has been moved, a forwarding address is left at the source processor pointing toward the destination processor. When a message is received at a given machine, if the receiver is a forwarding address, then the machine address of the message is updated and the message is resubmitted to the message delivery system (see Figure 4-1). As a byproduct of forwarding, an attempt may be made to fix up the link of the sending process (See next section).

The last case, messages sent using links created after the process has moved, is trivial. Links created after the process is moved will contain the same process identifier, and the last known machine identifier in the process address will be that of the new machine.



Message sent through a forwarding address
Figure 4-1

Simply forwarding messages is a sufficient mechanism to insure correct operation of the process and processes communicating with it after it has moved. However, the motivation for process migration is often to improve message performance. Routing messages through another processor (with the forwarding address) can defeat possible performance gains and, in many cases, degrade performance. The next section discusses methods for updating links to reduce the cost of forwarding.

An alternative to message forwarding is to return messages to their senders as not deliverable. This method does not require any process state to be left behind on the source processor. The kernel sending the message will receive a response that indicates that the process does not exist on the destination machine. Normally this means that the process the link points to has terminated; in this case, it may mean the process has migrated. The sending kernel can attempt to find the new location of the process, perhaps by notifying the process manager or some system-wide name service, or can notify the sending process that the link is no longer usable, forcing it to take recovery action. The disadvantage of this scheme is that, even if the kernel could redirect the message without impacting the sending process, more of the system would be involved in message forwarding and would have to be aware of process migration. This method also violates the transparency of communications fundamental to DEMOS/MP.

When the forwarding address is no longer needed, it would be desirable to remove it. The optimum time to remove it is when all links that point to the migrated process's old location have been updated. This typically would require a mechanism that makes use of reference counts. An alternative is to remove the forwarding address when the process dies. This can be accomplished by means of pointers backwards along the path of migration.

The forwarding address is compact. In the current implementation, it uses 8 bytes of storage. As a result of the negligible impact on system resources, we have not found it necessary to remove forwarding addresses. Given a long running system, however, some form of garbage collection will eventually have to be used.

It is possible for the processor that is holding forwarding address to crash. Since forwarding addresses are (degenerate) processes, the same recovery mechanism that works for processes works for forwarding addresses. Process migration assumes that reliable message delivery is provided by some lower level mechanism, for example, ***published communications*** [Powell & Presotto 83].

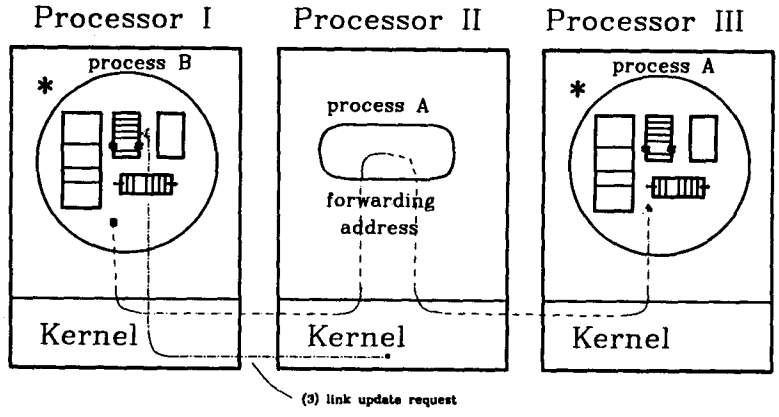
5. Updating Links

By ***updating links***, we mean updating the process address of the link. Recall that a process address contains both a unique identifier and a machine location. The unique identifier is not changed, but the machine location is updated to specify the new machine.

As mentioned above, for performance reasons, it is important to update links that address a process that has been migrated. These links may belong to processes that are resident on the same or different processors, and processes may have more than one link to a given process (including to themselves). Links may also be contained in messages in transit. It is therefore impractical to search the whole system for links that may point to a particular process.

Since race conditions might allow some messages to be in transit while the process is being moved, the message forwarding mechanism is required. As long as it is available, it can also be used for forwarding messages that are sent using links that have not yet been updated to reflect the new location of the process.

The following scheme allows links to be updated as they are used, rather than all at once: As it forwards the



Updating a link after a message forward
Figure 5-1

message, the forwarding machine sends another special message to the kernel of the process that sent the original message (see Figure 5-1). This special message contains the process identifier of the sender of the message, the process identifier of the intended receiver (the migrated process), and the new location of the receiver. All links in the sending process's link table that point to the migrated process are then updated to point to the new location.

Movement of a process should cause only a small perturbation to message communication performance. If the process that has moved is a user process, there will usually be few links that point to it. The links will tend to be either reply links, which will generate only one message and thus not need to be fixed up, or links from other user processes with which it is communicating, which will quickly be updated during the first few message exchanges. As a general rule, system processes do not retain non-reply links to user processes.

The worst case will be when the moving process is a server process. In this case, there may be many links to the process that need to be fixed up. Generally, links to servers are used for more than a few message exchanges, so the overhead of fixing up such a link is traded off against the savings of the cost to forward many messages. Moreover, the likelihood of server processes migrating is lower than for user processes. Servers are often tied to unmovable resources and usually present predictable loads that allow them to be properly located, reducing the need to move them.

6. Cost of Migration

The cost of moving a process dictates how frequently we are willing to move the process. These costs manifest themselves in two areas; the actual cost in moving the process and its related state, and the incremental costs incurred in updating message paths.

The cost of the actual transfer of the process and its state can be separated into *state transfer cost* and *administrative cost*. The state transfer cost includes the messages that contain the process's code, data, state, and message queue. DEMOS/MP uses the data move facility to transfer large blocks of data. This facility is designed to minimize network overhead by sending larger packets (and increasing effective network throughput). The packets are sent to the receiving kernel in a continuous stream. The receiving kernel acknowledges each packet (but the sending kernel does not have to wait for the acknowledgement to send the next packet). Three data moves are involved in moving a process. These are for the program (code and data), the non-swappable (resident) state, and the swappable state. The non-swappable state uses about 250 bytes, and the swappable state uses about 600 bytes (depending on the size of the link table). For non-trivial processes, the size of the program and data overshadow the size of the system information.

In addition, each message that is pending in the queue for the migrating process must be forwarded to the destination machine. The cost for each of these messages is the same as for any other inter-machine message.

The administrative cost includes the message exchanges that are used to initiate and orchestrate the task of moving a process. These costs depend on the internal structures of the system on which it is being implemented. The current DEMOS/MP implementation uses 9 such messages, each message being in the 6-12 byte range. These messages use the standard inter-machine message facility.

The incremental costs for process migration are incurred when a link needs to be updated. Each message that goes through a forwarding address generates two additional messages. The first is the actual message being forwarded to its new destination, and the second is the update message back to the sender. This will occur for each message sent on a given link until the update message reaches the sending process. In current examples, the worst case observed was two messages sent over a link before it was updated. Typically, the link is updated after the first message.

The movement of a process involves a small number of short, control messages, and a large number of block data transfers. The cost of migrating a process depends on the efficiency of both of these types of communications.

7. Conclusion

Process migration has proven to be a reasonable facility to implement in a communication-based distributed operating system. Less than one person-month of time was required to implement and test the mechanism in the current version of DEMOS/MP.

A number of DEMOS/MP design features have made the implementation of process migration possible. DEMOS/MP provides a complete encapsulation of a process, with the only method of access to services and resources being through links. There is no uncontrolled sharing of memory and all contact with the operating system, I/O, and other processes is made through a process's links. DEMOS/MP has a concise process state representation. There is no *process* state hidden in the various functional modules of the operating system. On the other hand, the system servers each maintain their own states, thus no *resource* state (except for links) is in the process state. Once a process is taken out of execution, it is a simple matter to copy its state to another processor. The location transparency and context independence of links make it possible for both the moved process and processes communicating with it to be isolated from the change in venue.

The DELIVERTOKERNEL link attribute allows control operations to be performed without concern for where the process is located. Thus control can follow a process through disturbances in its execution.

The mechanism for moving a process has been implemented, but there is not yet a strategy routine that actually decides when to move a process. The literature contains a few studies of metrics to use for processor and message traffic load optimization. Our continuing work

involves implementing process load balancing algorithms, and developing facilities for the measurement and analysis of the performance of communications in distributed programs.

References

Arora & Rana 80
Arora, R.K. & Rana, S.P., "Heuristic Algorithms for Process Assignment in Distributed Computing Systems", *Information Processing Letters* **11**, 4-5, December, 1980, pp. 199-203.

Baskett, Howard, & Montague 77
Baskett, F., Howard, J.H., Montague, J.T., "Task Communication in DEMOS", *Proc. of the Sixth Symp. on Operating Sys. Principles*, Purdue, November 1975, pp. 23-32.

Bokhard 79
Bokhari, S.H., "Dual Processor Scheduling with Dynamic Reassignment", *IEEE Trans. on Software Engineering* **SE-5**, 4, July, 1979.

Cheriton 79
Cheriton, D.R., "Process Identification in Thoth", *Technical Report* 79-10, University of British Columbia, October 1979.

Feldman 79
Feldman, J.A., "High-level Programming for Distributive Computing", *CACM* **15**, 4 (April), 1972, pp. 221-230.

Finkel 80
Finkel, R., "The Arachne Kernel", *Technical Report* TR-380, University of Wisconsin, April, 1980.

Powell 77
Powell, M.L., "The DEMOS File System", *Proc. of the Sixth Symp. on Operating Sys. Principles*, Purdue, November 1975, pp. 33-42.

Powell & Presotto 83
Powell, M.L., Presotto, D.L., "Publishing: A Reliable Broadcast Communication Mechanism", *Proc. of the Ninth Symp. on Operating Sys. Principles*, Bretton Woods N.H., October 1983.

Powell, Miller, & Presotto 83
Powell, M.L., Miller, B.P., Presotto, D.L., "DEMOS/MP: A Distributed Operating System", *in preparation*.

Rashid & Robertson 81
Rashid, R.F., Robertson, G.G., "Accent: A Communication Oriented Network Operating System Kernel", *Proc. of the Eighth Symp. on Operating Sys. Principles*, Asilomar, Calif., December 1981, pp. 64-75.

Rennels 80
Rennels, D.A., "Distributed Fault-Tolerant Computer Systems", *Computer*, **13**, 3, March, 1980, pp. 39-46.

Robinson 79
Robinson, J.T., "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms", *IEEE Trans. on Software Engineering* **SE-5**, 1, January, 1979.

Solomon & Finkel 79
Solomon, M.H., Finkel, R.A., "The Roscoe Operating System", *Proc. of the 7th Symp. on Operating Sys. Principles*, Asilomar, Calif., 1979, pp. 108-114.

Stone 77
Stone, H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", *IEEE Trans. on Software Engineering* **SE-3**, 1, January, 1977, pp. 85-93.

Stone & Bokhari 78
Stone, H.S. & Bokhari, S.H., "Control of Distributed Processes", *Computer*, July, 1978, pp. 97-106.