

A New Process Migration Algorithm

Michael Richmond

Michael Hitchens

{mar,michaelh}@cs.usyd.edu.au

Basser Department of Computer Science

University of Sydney

Australia

Abstract

Process migration has been advocated as a means of improving the load balancing and reliability of distributed systems. This paper reviews the major design issues for process migration algorithms, such as the amount of state information to be transferred and times at which information should be transferred. This examination demonstrates the existence of a process migration algorithm which has not previously been documented¹. After describing the existing algorithms, the new algorithm is given and compared to the other algorithms. The new algorithm promises better load balancing results while avoiding residual dependencies

1. Introduction

Distributed systems allow the movement of information between the various nodes of the system. In many distributed systems only data (such as files, database records, etc) may be transferred between the nodes. Other systems allow the site of execution of a process to be changed by transferring the relevant information from one host to another. This movement of the execution site of a process is known as process migration [19]. Various terminology is used in connection with process migration. In this paper the initial site of execution of a process will be referred to as the

source host and the new site of execution as the destination host, following Powell et al [14].

The migration of a process requires the transfer of (at least part of) the kernel data structures relating to that process and the process' address space. Once a sufficient amount of information about the process has arrived at the destination host execution of the process may resume. The exact information which must be transferred, and the order in which it is transferred, varies between systems and the process migration algorithms which have been proposed.

Process migration may be used to improve the performance of a system in a number of areas. For example:

- **Dynamic load balancing**
In a distributed system the processing load of the various hosts will often vary significantly. Processes may be migrated from hosts which have a relatively higher load to hosts which have a relatively lower load so as to spread the work load more equitably across the system.
- **Improved Reliability**
Sometimes the impending shutdown of a host may be known in advance (such as for periodic maintenance). A process may be migrated to another host so that it will survive the shutdown of its current host processor.
- **Reduced network traffic**
A small process may be migrated to the site of a resource which cannot be moved (such as a particular device) or a resource whose movement would cause more traffic than migrating the process (such as a large database). When the process has used the resource, it may be migrated back to its original host.

¹After an extensive literature review, we have been unable to find any previous description of the algorithm presented in this paper. One of the purposes of presenting the algorithm in this forum is to allow any who may have previously described this algorithm in the literature to inform us of their work.

The decisions as to when it would be useful to migrate a particular process and which host should be the destination of the migration are system policy decisions and are outside the scope of this paper.

Process migration algorithms all have a number of basic tasks they must carry out:

- the process must be suspended on the source host,
- its state must be transferred to the destination host,
- its execution must be resumed on the destination host.

The issues in the design of process migration algorithms are:

- the detail of carrying out each task,
- the order in which the tasks are carried out.

By examining the possible alternatives for dealing with these issues the nature and relationship of the fundamental process migration algorithms will be clarified. This process will also demonstrate the existence of a fundamental process migration algorithm, which we call post-copy, which we believe has not previously been described in the literature.

Existing process migration algorithms suffer from one or both of the following problems:

- a considerable time lag between the decision to migrate being made and the resumption of execution on the destination host,
- a continuing reliance on information stored on the source host (*residual dependencies*).

After describing post copy, a comparison, in general terms, is made between it and the other fundamental process migration algorithms. Post copy does not suffer from the second drawback above and has a noticeably smaller time lag than other algorithms also free of residual dependencies. Finally, an outline of a planned implementation of the algorithm is presented.

2. Design Issues for Process Migration Algorithms

The following tasks are performed when a process is migrated:

- The decision is taken to migrate the process.
- The execution of the process on the source host is suspended.
- The state of the process is transmitted from the source host to the destination host.
- The state of the process is reconstructed on the destination host.
- The execution of the process is begun ("resumed") on the destination host.
- Information about the process is removed from the source host.

These tasks are common to all process migration algorithms. For some of them, the details of how they are carried out vary with the actual algorithm. For others, the details are independent of the algorithm being employed. The tasks which fall in to the latter category are: the mechanism used to decide when to migrate a process (a policy decision), the reconstruction of the process at the destination host (dependent upon the mechanisms of the operating system) and the removal of the information from the source host (again dependent upon the operating system). Process migration algorithms also vary in the order in which the tasks are carried out, except that obviously the decision to migrate must come first and information can only be removed from the source host once it has been transferred to the destination host (although this may be on a piece by piece basis or all information at once). Process migration algorithms may also involve other entities besides the source and destination hosts. We shall not consider such algorithms. A detailed treatment of these issues may be found in [17].

The design issues which differentiate the various algorithms are then:

- how much of the process' state is transferred from the source host to the destination host?

	amount of state information transferred	process suspended	process resumed
1	all	on decision to migrate	on completion of full transfer
2	all	on decision to migrate	on completion of minimal transfer
3	all	on completion of transfer	on completion of full transfer
4	all	on completion of transfer	on completion of minimal transfer
5	partial	on decision to migrate	on completion of full transfer
6	partial	on decision to migrate	on completion of minimal transfer
7	partial	on completion of transfer	on completion of full transfer
8	partial	on completion of transfer	on completion of minimal transfer

Table 1 Possibilities for process migration algorithms

- when is execution of the process on the source host suspended?
- when is execution of the process on the destination host resumed?

The last point may be restated as: how much of the process' state must be present on the destination host before execution may be resumed? There are a number of other issues which affect a process migration mechanism, but these are either dependent on a particular system or a matter of designer choice (eg., how are residual dependencies from communication links avoided or is the migration mechanism to be implemented inside the kernel or outside of it?). Such issues are orthogonal to the actual process migration algorithm.

For each of the issues listed above there are a number of possible alternatives.

The amount of state of information that will be migrated may be:

- all of it
- only that portion of the state information which is required by the process for execution on the destination host

As it is impossible to determine in advance what portion of the state information will be

required, the second alternative leads to the state information being transferred piece by piece, as required by the process.

Execution of the process on the source host may be suspended:

- immediately the decision to migrate is taken
- when the state transfer is complete
- when some portion of the state transfer is complete

The second and third alternatives will normally require some repeated transfer, as the process may alter some data on the source host after that data has been transferred to the destination host. The data must be recopied after the process has been suspended to ensure that it will have an up to date version of it when execution is resumed. The most obvious portion of the state information to transfer for the third alternative is that part of the address space not currently in the process' working set (or the portion of the address space not currently in physical memory, etc., depending upon the system), as this information is least likely to be altered by the process during the transmission process. However, this does not

take account of changes in the working set during the transmission process.

Execution on the destination host may be resumed

- when the state transfer is complete,
- when the minimal possible amount of state information has been transferred (normally the minimal amount of kernel data necessary to reconstruct the process on the destination host), or
- when some other portion of the state transfer is complete.

The second and third alternatives will often result in the process requesting information which is not currently present at the destination host. The process will then be suspended while this information is recovered from the source host. The most obvious portion of the state information to transfer, in addition to the kernel data, is the process' current working set (or portion of the address space in physical memory, etc., depending upon the system) as this is the information most likely to be immediately required by the process when execution is resumed.

It should be immediately obvious to the reader that some possible combinations of the above alternatives are unreasonable. In particular, those combinations where the process' execution is resumed after less state is transferred than required for process suspension can be ignored.

In this paper we will not consider the possibility of process suspension and resumption after partial transfer, concentrating instead on the basic alternatives. Again, for a more complete consideration of the design alternatives, see [honours thesis]. In most cases these additional alternatives lead merely to slight variations of the basic algorithms. This limitation will simplify our attempts to consider the basic migration algorithms. Also we restrict ourselves, for the most part, to process migration algorithms which involve only the source and destination hosts, and no third party (such as a file server). Given these restrictions, the possible alternatives for process migration are listed in table 1. It can be seen that options

4, 5, 7 and 8 are unreasonable, as noted in the previous paragraph.

3. Process Migration Algorithms

There have been a number of implementations of process migration described in the literature [eg., 1,4,7,18,20,21,22,23]. Most of these implementations may be directly related to one of the options found in table 1. Those that are not introduce other entities into the process migration mechanism [3,4] and an example of this is discussed in section 3.4. Process migration implementations described in the literature employ one of the following four process migration algorithms:

- eager copy
- lazy copy
- pre-copy
- flushing

3.1 Eager Copy

Eager copying corresponds to the first option in table 1. For eager copy, the tasks which must be undertaken to migrate a process are performed in the following order:

- The decision is taken to migrate the process.
- The execution of the process on the source host is suspended.
- The entire state of the process is transferred, including the entire contents of the address space and all relevant kernel data (such as information on open files, message channels, execution state, current directory, etc, depending on the particular system).
- The state of the process is reconstructed on the destination host.
- The execution of the process is begun ("resumed") on the destination host.

Execution of the process is not resumed until all state information has been fully transferred. The information held by the source host may be removed as it is transferred or when execution is resumed. Removal is often achieved simply by using the normal process destruction operation on the process image held by the source host. Eager copy process migration is depicted in figure 1.

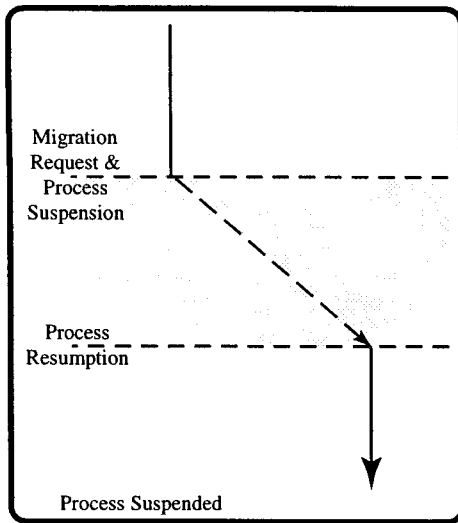


Figure 1 : Eager Copy

Eager copy [eg., 14,20] is the simplest, and probably the most commonly used, process migration algorithm. Its conceptual simplicity allows a relatively straightforward implementation. The length of the delay between process suspension and resumption will be dependent on the size of the process, but will generally be longer than with other process migration algorithms. For certain classes of processes (such as real time or those with large amounts of I/O) such a delay may be undesirable.

3.2 Lazy Copy

Eager copy transfers *all* information relevant to the process, before the process resumes execution on the destination host. In contrast lazy copy [22], which corresponds to option 6 from table 1, transfers only the minimum necessary information for the process to resume execution. This will usually consist of all or part of the data held by the kernel about the process and a small part (such as two to three pages, etc) of the address space. All other information is transferred only when referenced by the process executing on the destination host. The order of the tasks is as follows (and depicted in figure 2):

- The decision is taken to migrate the process.
- The execution of the process on the source host is suspended.

- The minimal necessary subset of the process' state is transmitted from the source host to the destination host.
- The state of the process is reconstructed on the destination host from the state information transmitted in the previous step
- The execution of the process is begun ("resumed") on the destination host.
- Other state information is transferred as required.

Once the process resumes execution, reference may be made by the process to state information which still resides on the source host. Any such reference is trapped by the destination host kernel, and the information requested from the source host. The execution of the process will be suspended while the state information is requested and sent from the source host. While this will appear to the process as a normal delay as, for example, with a page fault, it will typically be of longer duration than with local page fault servicing. The exact duration of the delay will depend upon the characteristics of the network connection between source and destination.

As state information is only sent to the destination host when it is required there is no general method for determining when any particular portion of the process' state will be transferred. This results in the process' state being spread across destination and source hosts (or multiple source hosts if the process is migrated a number of times). The continued execution of the process then depends upon the continued execution of both source and destination hosts and the liveness of the connection between them. This is the *residual dependency* problem [4].

The advantages of lazy copy are

- the speed with which execution of the process is resumed upon the destination host and
- the possibility of an overall reduction in network traffic.

The first is achieved as only a minimal amount of information is transferred before execution may begin. The second results from avoiding the need to transfer all the state information.

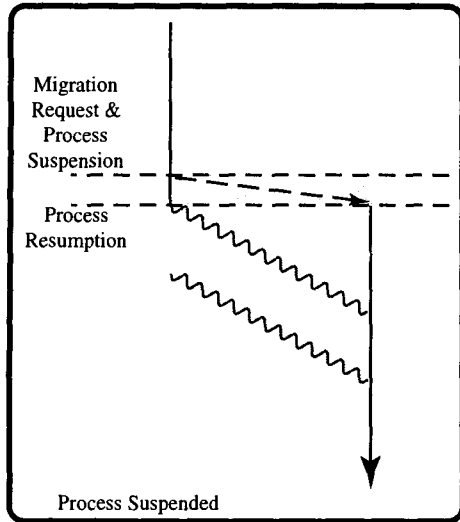


Figure 2 : Lazy Copy

There is still some delay between suspension and resumption, resulting from the network transmission time of the state information which is sent, and the cost of process creation on the destination host.

As lazy copy results in residual dependencies on the source host, it cannot be used to improve system reliability. Dynamic load balancing, may also be impaired, as the costs associated with remote references can adversely affect the source host. Even so, the copy on reference strategy has been successfully implemented in Accent [22], Mach [7] and OSF/1 AD [13] systems.

3.3 Pre-Copy

Pre-copy corresponds to option 3 from table 1. In contrast to the previous two algorithms, pre-copy does not suspend the process and transfer its kernel data until all (or at least most) of the address space of the process has been transferred from the source host to the destination host. Pre-copy takes place as follows (and depicted in figure 3):

- The decision is taken to migrate the process.
- The address space of the process is transferred.
- In *parallel* with this transfer, execution of the process continues on the source host.

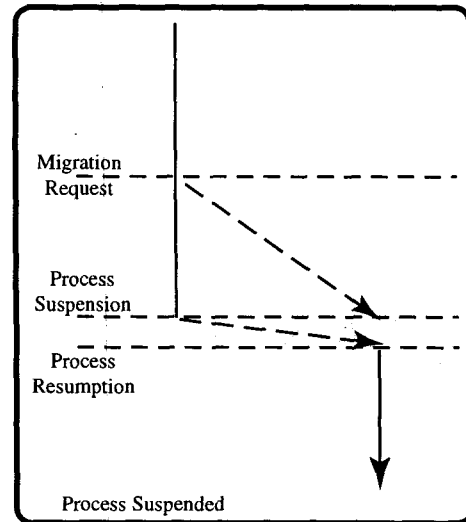


Figure 3 : Pre-Copy

- When address space transfer is complete, execution of the process on the source host is suspended.
- All relevant kernel data (such as information on open files, message channels, execution state, current directory, etc, depending on the particular system) and any portions of the address space which were altered after initial transfer are sent to the destination host.
- The state of the process is reconstructed on the destination host.
- The execution of the process is begun ("resumed") on the destination host.

There is no way of knowing, in general, whether a particular page (or segment, etc, depending upon the system) will be accessed by the process on the source host after that page has been transferred to the destination host. This means that all information must be retained on the source host until migration is complete. It also means that the transfer of the kernel data must be accompanied by a final flush of the address space to transfer any pages that have been altered after their initial transfer. Note that this means that pre-copy will transfer some information twice.

Pre-copy has been implemented on the V operating system [21]. Pre-copy does succeed in avoiding the large time lag present in eager

copy between process suspension and process resumption. However, the overall lag between the decision to migrate and the commencement of execution on the destination host will, usually, be longer than with eager copy, due to the need to transfer some information twice. Depending on the exact memory activity pattern of the process, the final delay may also be quite long if large sections of the address space must be re-copied.

3.4 Flushing

Most implementations of process migration reported in the literature directly involve only the source and destination hosts. However, a small number involve a third entity (such as a file server). An example of this is the process migration mechanism of Sprite [4]. In this section we give a brief description of the underlying algorithm, called flushing (see figure 4). For a discussion of the issues raised by introducing a third party to the process of migration, see [17].

Flushing depends upon the operating system's implementation of virtual memory. In Sprite backing storage for virtual memory is implemented using ordinary files. These backing files are stored by the network file server. They are thus equally accessible from anywhere in the network.

Migration using flushing proceeds as follows:

- The process is suspended on the source host.
- All dirty pages are flushed to the file server.
- The kernel data of the process is packaged and transferred to the destination host.
- The kernel data is rebuilt on the destination host.
- The process resumes execution on the destination host, with pages being retrieved from the file server using the standard page fault handling mechanism.

Flushing can avoid all residual dependencies, and may significantly reduce the time lag

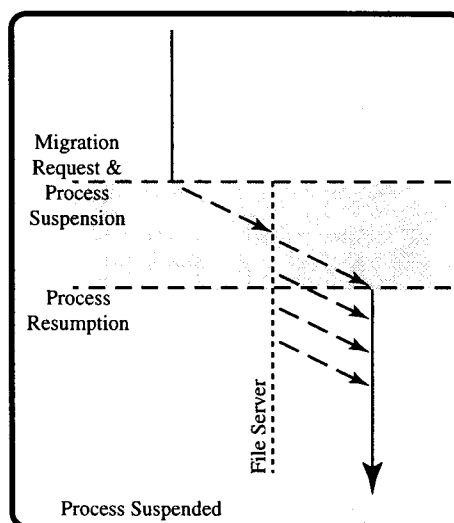


Figure 4 : Flushing

experienced by a process between suspension and resumption. Any actual reduction is dependent on implementation policy decisions and network speed. Flushing has been found to be an effective strategy for migration under Sprite. However, there is the requirement that memory be supported using the same mechanism as files so that page faults can be efficiently resolved using the file server.

4. Migration by Post-Copy

In the above discussion of the process migration algorithms previously described in the literature, the reader may have noticed that one of the viable options from table 1 (number two) has not been covered. This algorithm, which we call **post-copy** displays some advantages over the other algorithms, as discussed in the following section.

Post copy displays some similarities to the other algorithms. This is to be expected, as it does not differ from the others in all the design areas identified in section 2. As with lazy copy, process execution is resumed on the source host as soon as possible after the migration decision is made. In common with pre-copy, state transfer takes place in parallel with process execution, but for post-copy the execution is on the destination host, not the source.

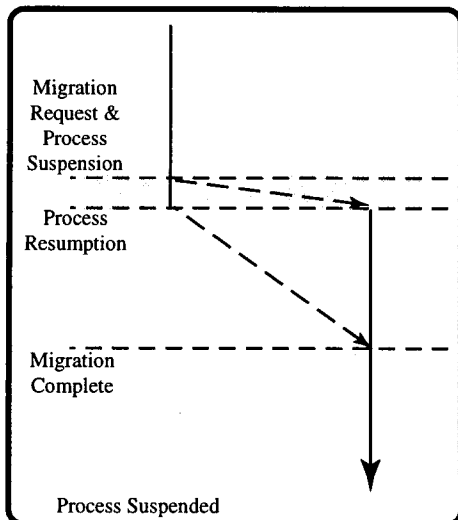


Figure 5 : Post-Copy

Briefly, post-copy process migration is carried out as follows (see figure 5):

- The decision is taken to migrate the process.
- The execution of the process on the source host is suspended.
- The minimal necessary subset of the process' state is transmitted from the source host to the destination host.
- The state of the process is reconstructed on the destination host from the state information transmitted in the previous step.
- The execution of the process is begun ("resumed") on the destination host.
- In *parallel* with the execution on the destination host, the source host transmits the remaining state information to the destination host.
- At any time the process requires state which has not yet been transmitted, a request can be sent to the source host.
- The source hosts replies with the requested state information.

When all of the process' state (as held by the source host) has been transmitted, the source host may discard the process image, using the normal process destruction mechanism. Note that the replies to the requests from the destination host for specific portions of the process' state should take precedence over the

transfer of the remaining state information, as the need for it is more urgent. That is, when the source receives such a request, it should reply to it before it continues to send any of the remaining state information. Also, information that is sent in reply to such a request should be marked by the source host as transferred, and not re-sent later.

Once the process resumes execution on the destination host, the transfer of pages from the source host to the destination host should be transparent to the execution of the process. When reference is made to state which already resides on the destination host (as a result of earlier transmission) the process should be able to access the state in the usual manner for the operating system. Any reference which is made to state which does not reside on the destination host must be either waiting to be transmitted, on the source host, or in transit from the source. In both cases, the reference is trapped by the kernel, and a request for the information is sent to the source host, with the process being blocked until the information arrives. This will appear to the process as a normal delay due, for example, to a page fault, though the delay will usually be of a longer duration than with local page fault servicing. The exact duration of the delay will depend upon the characteristics of the network connection between source and destination, and the location of the state in the transmission steps.

5. Comparison of Post-Copy with other Algorithms

Comparing process migration algorithms is a difficult problem. Many reports of implementations of process migration include a quantitative assessment of the implementation [4,20,22]. However, comparing implementations, which may vary in their hardware, the network and process environment in which they were tested and the algorithm employed, is problematic at best. This makes quantitative comparisons of the different algorithms difficult to obtain.

As we have not yet implemented post-copy, quantitative evaluation is currently unavailable.

It is possible to make some qualitative comparisons between post-copy and the other algorithms. We will consider the following areas:

- delay between decision to migrate and the resumption of execution on the destination host,
- total delay experienced by a migrating process,
- the amount of residual dependencies, and
- the amount of network traffic required.

Other areas (such as delay to reduction of processing load on source host) could be examined, but those listed will give some indication of the potential value of post-copy.

Using post-copy, the delay between the decision to migrate a process and its resumption of execution on the destination should be as short as that provided by lazy copy and shorter than any other algorithm. This results from the initial transfer of only a minimal subset of the process' state to the destination host. All other algorithms (except lazy copy) require the transfer of additional information before execution is resumed, so it is reasonable to assume they would result in a longer delay.

The total delay to execution of a process refers to the amount by which the transfer of kernel state and address space over the network delays completion of the process' execution. We expect that the delay experienced with post-copy would probably be slightly worse than with pre-copy and as good or better than that with either lazy or eager copy, although this will vary dramatically with individual process and network behaviour.

Making such comparisons in general is difficult. For example, with pre-copy the process continues execution on the host while another, new, process transfers the address space to the destination. This additional process may cause some delay to the migrating process' execution. The amount of this delay is hard to quantify in general terms. Ignoring this factor, the only delay suffered by pre-copy is the final flush of pages altered after their initial transfer. Depending upon the amount of time taken to carry out the pre-copying, the number of pages affected would probably be reasonably low,

usually approximating the number of pages in the working set. Processes migrated using post-copy, in addition to an equivalent transfer of the working set, will experience delays due to the requests for pages before they have been transferred in their normal order. This additional delay results in the total delay being slightly more than with pre-copy. However, when the affect of the transferring process of pre-copy is taken into account the difference between the two will most likely be slight. Only detailed comparative measurements upon a single system will show the exact nature of the difference.

The delay using post-copy will probably be shorter than with lazy copy as, although the latter does not need to transfer all information, the process is delayed for every initial request for a page, whereas with post-copy, some pages will be transferred before the process requests them. This will be affected by exactly how many pages are requested by a process migrated using lazy copy. With eager copy, all information is transferred before the process resumes execution, whereas post-copy allows some parallelism between transfer and execution, reducing the delay.

Though lazy copy achieves both a minimal execution delay and a reasonably short total delay, it does so at the expense of residual dependencies. In contrast, post-copy while initially mimicking lazy copy with its transfer of minimal state information avoids residual dependencies by transferring the remaining process state after the process has been resumed. In this it matches eager and pre-copy. While a particular implementation may not totally avoid residual dependencies, this is a function of the implementation and not an inherent feature of the algorithm. While post-copy does retain some state on the source host after the process resumes execution on the destination, this state information will be transferred to the destination host within a finite time period.

Post copy will probably result in higher rates of high network traffic than the other migration algorithms. This is fairly obvious in the case of lazy copy, as the information that is never transferred will not generate any network

traffic, whereas post-copy requires all state information to be transferred. With eager copy, while all state information must be transferred, no request messages are required, whereas such messages will usually be sent when a process is migrated using post-copy. In contrast we do not expect post-copy message costs to be significantly higher than those of pre-copy. With pre-copy, some pages (those altered after initial transfer) are sent twice. With post-copy, those requested from by the destination will require two messages (request and transfer). The exact affect of this on network traffic will probably depend upon the relative size of network messages and memory pages (or other division of memory).

Th comparisons made here are hypothetical only and await confirmation from actual testing. We plan to implement the other basic algorithms in our test environment to enable direct comparisons between them and post-copy.

6. Future Work

As noted earlier, we believe that post-copy process migration has not previously been described in the literature. Correspondingly we are unaware of any implementation of the algorithm. We are therefore carrying out an initial implementation of post-copy process migration. The primary goal for this work is to develop a working implementation of post-copy migration. This will allow us to establish the practical viability of the algorithm. We will then be able to use this implementation to carry out direct performance comparisons with other migration algorithms described in section 3 above. These algorithms will be implemented on the same system as post-copy, allowing some meaningful comparison to be made between the algorithms. As a secondary goal, we hope to achieve migration transparency at the process level, ie. no user level process will be aware of any process migration, before, during or after the migration. While this is a valuable attribute of a fully useable implementation of process migration, we believe it is not absolutely necessary for our initial implementation

The implementation is based on the Open Software Foundation Research Institutes mk6.1 release of the Mach microkernel for the Intel 386/486 architecture, using the mkLinux operating server as our user interface to Mach [5]. Development is being performed using the supplied OSF build tools on the Linux operating system. The implementation of process migration will be at the Mach task level, below the operating system server. A Mach task corresponds to the general notion of a heavy-weight process, consisting of a collection of system resources, a large (potentially sparse) address space, and a number of threads which may reference these resources [5]. Implementing at the task level has the advantage of simplifying our migration mechanism, as state directly related to the operating system server (such as open files and pipes) are maintained by the operating system server, and will be unaffected by migration. Unfortunately this means that a dependency on the OS server at the source host will remain at the OS process level. This dependency is a result of the level at which we are implementing, not the algorithm itself and has been identified as being an issue in previous migration implementations [OSF Tech Report]. It is expected that the migration of the OS server at a similar time to the task migration will result in no dependencies remaining to the source host.

7. Conclusion

By examining the fundamental issues in the design of process migration algorithms we have identified a process migration algorithm that we believe has not been described elsewhere in the literature. As such, it has not previously been implemented on any platform. Post-copy transfers the minimal of process state to the destination host for resumption of execution as soon as the migration decision is made. The remaining state is then transferred in parallel with the process' continued execution on the destination host. In common with the other process migration algorithms, post-copy may be used with any migration policy.

Post copy allows rapid load balancing and minimal delay to the process between the decision to migrate and the resumption of execution. Unlike lazy copy, which has similar properties, post copy avoids the problems of residual dependencies. As well as being useful for migration in general systems, it follows that post-copy would be attractive for process migration in real-time systems, where delays must be minimised. Any delays suffered by the process occur in small portions, rather than the single large delay suffered under eager copy. Unlike pre-copy, post-copy does not add extra load to the (already heavily loaded) source host by continuing the process' execution in parallel with the process responsible for pre-copying the process state. Instead, this temporary load is transferred to the destination which, under most migration policies, will be relatively lightly loaded.

Work has already begun on an implementation of post-copy migration, on the Mach operating system kernel. This will be used to carry out quantitative comparisons between post-copy and other process migration algorithms. The results of this work will be reported in [17] and a future publication.

References

- [1] Artsy, Y. & Finkel, R., Designing a Process Migration Facility: The Charlotte Experience, *IEEE Computer*, 22(9), pp. 47-56, September 1989.
- [2] Barak, A. & Shiloh, A., A Distributed Load Balancing Policy for a Multicomputer, *Software Practice and Experience*, 15(9), pp. 901-913, September 1985.
- [3] Douglis, F. & Osterhout, J. Process Migration in the Sprite Operating System. *7th International Conference on Distributed Computing Systems*, pp. 18-25, 1987.
- [4] Douglis, F. & Osterhout, J. Transparent Process Migration Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8), pp. 757-786, August, 1991.
- [5] Loepere, K., (ed.), OSF Mach Approved Kernel Principles, Open Software Foundation, June 1993.
- [6*] Milojevic, D.S., Giese, P. & Zint W., Load distribution on Microkernels. In Proceedings of IEEE workshop *Future Trends in Distributed Computing Systems*, Lisboa Portugal, September 1993.
- [7*] Milojevic, D.S., Zint, W., Dangel, A. & Giese, P., Task migration on the top of the Mach microkernel. In Proceedings of *The 3rd USENIX Mach Symposium*, Santa Fe, April 1993.
- [8] Mullender, S.J., van Rossum, G., van Renesse, R. & van Staveren, H., Amoeba - a Distributed Operating System for the 1990s, *IEEE Computer*, 23(5), pp. 44-53, May 1990.
- [9] Nuttall, M. A brief survey of systems providing process or object migration facilities, *Operating Systems Review*, 28(4), pp. 64-80, October 1994.
- [10] Nuttall, M. Survey of systems providing process or object migration facilities. Technical Report DoD 94/10, Imperial College, London, May 1994.
- [11] Osterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N. & Welch, B.B., The Sprite Network Operating System, *IEEE Computer*, 21(2), pp. 23-36, February, 1988.
- [12] Paindaveine, Y., The OSCAR project. OSF Institute Technical Report TR-98, Belgium, March 1996.
- [13*] Paindaveine Y. & Milojevic, M.S., Process vs task migration. In Proceedings *29th Annual International Conference on System Sciences*, Wailea Hawaii, January 1996.
- [14] Powell, M.L. & Miller, B.P., Process Migration in DEMOS/MP, In Proceedings of *The 9th ACM Symposium on Operating Systems Principles*, pp. 110-119, October 1983.

- [15*] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. & Jones, M., Mach: A foundation for Open Systems. In Proceedings of the *Second Workshop on Workstation Operating Systems (WWOS2)*, September 1989.
- [16] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. & Jones, M., Mach: A system software kernel. In Proceedings of *The 34th Computer Society International Conference COMPCON 89*, February 1989.
- [17] Richmond, M. A New Process Migration and its Implementation. Honours Thesis, University of Sydney, 1996.
- [18] Schrimpf, H., Migration of processes, files and virtual devices in the MDX operating system. *Operating Systems Review*, 29(2), pp. 70-81, April 1995.
- [19] Smith, J.M., A Survey of Process Migration Mechanisms, *Operating Systems Review*, 22(3), pp. 28-40, July 1988.
- [20] Steketee, C., Socko, P., & Kiepuszewski, B., Experiences with the implementation of a process migration mechanism for Amoeba. In Proceedings of *The 19th Australasian Computer Science Conference*, Melbourne Australia, pp. 140-148, Jan-Feb 1996.
- [21] Theimer, M.M., Lantz, K.A. & Cheriton, D.R., Preemptable Remote Execution Facilities for the V-System. In Proceedings of *The 10th ACM Symposium on Operating Systems Principles*, pp. 2-12, 1985.
- [22] Zayas, E. R., Attacking the Process Migration Bottleneck. In Proceedings of *The 11th ACM Symposium on Operating Systems Principles*, pp. 13-24, November 1987.
- [23] Zhu, W., Groscinski, A., & Gerrity G., Process Migration in RHODOS.