Accepted to EUUG Autumn Conf. 88
Portugal Oct. 88

DTIC FILE COPY

AD-A201 874

Distributed Light Weight Processes in MOS †

Amnon Barak
Dalia Malki

Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904 (Israel)

dalia@humus.huji.ac.Il

DTIC
ELECTE
OCT 1 9 1988
S D
D

1. Introduction

Integrated multicomputer systems consist a set of loosely coupled processors, each with its own local memory, into a single machine environment. In the distributed systems model, various user processes may run concurrently on different machines and possibly communicate to achieve a common goal. This form of concurrency encourages a programming style that uses large grain-size computation blocks. Such *distributed programs* consist of a set of execution entities (called *threads* or *tasks*) that perform considerable amount of work independently and communicate infrequently through messages. Threads are a convenient way of expressing concurrent programs and therefore, many programming languages embody thread-like entities in their syntax, ~~*e.g.* Occam [IN84a] and Linda~~ [ACG86]. However, the overhead of handling processes by the operating system is costly. For instance, it has been noted that the UNIX processes are *heavy-weight* in that they carry much associated state information. Therefore, operations on them (*e.g.* context switching) are slow.

Light Weight Processes (LWP) has been suggested by Kepes [Kep85] as a programming tool for supporting cooperating processes on a uniprocessor. In the LWP mechanism suggested by Kepes, a runtime support library provides the coroutine primitives within a single, heavy-weight-process (HWP). Another alternative for supporting LWPs is at the kernel level. On a multiprocessor, the kernel support version has a primary advantage of allowing real parallelism. One of the most recent operating system kernels that support LWPs is Mach [ABG86]. However, none of the kernel or user level LWP mechanisms provide concurrency in distributed environments.

This paper describes the Distributed Light Weight Processes (DLWP) mechanism, a facility for supporting distributed programs in MOS, a multicomputer operating system [BaL85]. The goal of the Distributed Light Weight Processes mechanism is to be able to exploit concurrency in a distributed environment. The mechanism is designed to be able to support a variety of application types by supporting processes as a programming tool. It exploits concurrency up to the level available in the system and provides additional, virtual concurrency through time sharing. In this way, it can be used both for efficient utilization of concurrency and for experimenting with large scale concurrent programs.

The DLWP mechanism is implemented immediately above the operating system kernel level, in the form of a user-level runtime library. It extends the uniprocessor Light Weight Processes mechanism through a new operation, *split*, which adapts the classical Light Weight Processes mechanism for distribution and dynamically disperses the workload among processors. A LWP *pod* within a HWP may *split* to create multiple pods that execute in different HWPs. The MOS dynamic load balancing [BaS85] automatically assigns the HWPs to different machines and provides concurrency.

The partitioning strategy takes into consideration past behavior of the LWPs, in terms of CPU consumption and communication. This profile information is used to reach a partition that splits the load evenly while incurring minimum communication overhead. For this purpose, the profile information is kept in a graph

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

† This work was supported in part by the U.S Air Force Office of Scientific Research under grant AFOSR-85-0284, in part by the National Council for Research and Development, grant no. 2525, and in part by the Foundation for Research in Electronics Computers and Communication, Israel Academy of Science and Humanities.

88 10 18 074

and a heuristic graph partitioning algorithm is employed.

2. Background: MOS

The Distributed Light Weight Processes project is part of MOS, a Multi computer Operating System that was developed at the Hebrew University of Jerusalem. MOS is a general purpose, time sharing operating system that intergrates a cluster of loosely coupled homogeneous computers into a single machine UNIX system.

Hardware

The hardware model for which MOS is intended consists of a cluster of homogeneous computers loosely connected by a local area communication network. Each computer is self contained, has its own memory and may have local peripherals. The specific configuration of the system as of spring 1988 consists of seven PCS/CADMUS 9000 machines, each with a MC68010 CPU, one megabyte of memory and an 80 Mb Fujitso disk drive. The machines are connected by a Proteon 10Mbps token ring based local area network.

Network Transparency

The basic requirement of the MOS design is to maintain in full the classic UNIX interface. This implies that user processes are oblivious to the existence of the network and take no account of the special system architecture. For example, user files are named and accessed uniformly throughout the network and the conventional UNIX semantics are preserved. Dynamic load balancing

Dynamic Load Balancing

The assignment of processes to processors may change dynamically in MOS during their runtime through *process migration* in order to disperse the workload in the system efficiently. The ability to migrate processes emerges from the homogeneity of the system and the network invisibility, so that in MOS the processes are unaware of the actual processor they are running on. More details about MOS are found in [BaL85].

3. Light Weight Processes in MOS

The *Light Weight Processes* (LWP) mechanism is a programming tool that supports the existence of program parts as virtual parallel processes within one, *heavy weight process*. The LWPs share information and may exchange partial results during runtime via *message passing*. Designed at the user level, the LWPs package in MOS is a library of routines providing scheduling and message passing services. Any program may use these services by linking its object modules with the LWPs library and by requesting the services via the interface routines. In particular, calls to the LWP primitives may be generated automatically by compilers of high level parallel programming languages, in order to provide support for parallel programs.

DTIC COPY INSPECTED 4

The LWPs package simulates an abstract parallel machine. Any number of coexisting LWPs may execute on it and are supposed to run in parallel on homogeneous processors. The only restriction is on the total amount of memory allocated for all of the LWPs for their private stacks.

The Scheduler

The separate threads of control are maintained by a scheduling module that conducts a *time sharing scheduling*, similarly to the scheduling of processes in a time sharing operating system. First, the scheduling routine scans the list of ready LWPs and selects one for execution. Then it performs a *context switch* to the selected LWP, *i.e.* resumes its execution from the point it was last suspended. The LWP is then allowed to run until an interrupt or the next scheduling point.

per NP

A-1

Message Passing

In distributed applications it is essential to have the ability to communicate messages between its distinct components. The handling of *message passing* is therefore an improtant aspect of the LWP package. Furthermore, the strength of the LWP package depends heavily on the strength of the message-passing part, in terms of generality, applicability and efficiency.

In MOS, a fully connected network of LWPs is used, with a semi-synchronous communication protocol (*i.e.* a *receiving* LWP waits for a message to arrive, while a *sending* LWP resumes operation immediately). Messages are addressed to LWPs using the LWP identification scheme. This message-passing scheme is believed to be sufficiently powerful to support most languages' communication requirements.

Memory Handling and Shared Memory

Another function of the LWPs package is the handling of the LWPs' memory. The LWPs' data reside in the heavy weight process' memory space. This space is accessible to all of the LWPs, since they are essentialy parts of one program. The LWPs address two types of data storage: a static, global data area and a private, runtime area.

The global area of memory is shared among all the LWPs in the heavy weight process. It is allocated at compile time and its allocation is not changed during the execution of the program. The global area contains variables that are shared by all the processes, system parameters, used by the processes and constant values that are initialized by the compiler.

A logical division of the heavy weight process' memory is provided by allocating each LWP a private memory portion. The private area is managed as a stack and is used for local variables, temporary storage, procedure activation records and per-LWP specific information. The private areas are not protected by hardware access permissions and thus, it is required that the LWPs cooperate in confining themselves to their local address space. The advantage of this organization is that the LWPs have their stack segments at different locations, which makes the *context switch* an inexpensive operation that involves only the saving and resuming of registers' contents, including the stack-pointer and (last) the program-counter.

4. Distributed Light Weight Processes

In distributed environments, concurrency is achieved in the form of processes that run in parallel on different machines and interact through interprocess communication. A collection of processes executing in a distributed system may indeed cooperate for a common purpose, but usually they represent programs that fulfill different goals and were developed independently. Furthermore, the goal of distributed systems is usually to enhance the overall system performance, rather than the performance of a single application. Consequently, the handling of processes is usually costly and hence they are termed *heavy-weight-processes.*

The Distributed Light Weight Processes (DLWP) mechanism is an extended version of the Light Weight Processes mechanism for supporting concurrent applications in distributed systems. Using this mechanism, both the conveniency of the LWP interface is supported and actual concurrency is achieved. Thus, on one hand, the LWP mechanism provides general services that are detached from actual hardware limitations and allow a large number of virtual processes. At the same time, the distributed version exploits the actual concurrency in the system by breaking down applications to several components that execute in parallel. This is done by mapping a (possibly large) set of cooperating LWPs into a smaller number of LWP clusters.

The Split Concept

In MOS, actual parallelism is introduced into the DLWP mechanism through a new operation, called *split*. A LWPs pod may decide to split by self partitioning into two disjoint subsets intended to exist together and run independently (in parallel) on different machines. A *split* operation is carried out in several steps. First, the pod is duplicated, creating a new LWPs pod that contains replica of the code and data needed for managing the LWPs. The set of LWPs of the original pod is then partitioned into two disjoint subsets. One partition remains in the original pod while the other partition shifts to the new pod's environment. After the split, the LWPs assigned to each of the pods resume their operation without interrupt. The pods are

created and destroyed dynamically. Interaction between pods is done according to the LWP needs and sharing of data between pods is not allowed.

An important underlying assumption of the split operation objective is that the pods have exclusive resources, *i.e.* they do not compete among themselves, but just manage their internal resources. This assumption requires of the underlying distributed operating system the support of concurrent execution of pods (up to the system concurrency limit). In MOS, processes dynamically migrate between machines and receive their share of the system resources by the dynamic load balancing mechanism. Thus, newly created pods migrate within MOS to run on different machines in parallel.

The split/merge operations of pods, together with a suitable strategy for their activation, facilitate a dynamic suboptimal load balancing which disperses the workload among the different machines.

Pod Interaction

The pods need to communicate between themselves to maintain the integrity of the split application. Thus, the DLWP mechanism requires from the underlying system the ability of pods to communicate via some underlying RPC/IPC mechanism. The pods interact to achieve two goals: inter-pod message passing and load dispersing.

Note that the LWPs are unaware of the underlying communication and they interface only to the local pod.

4.1. Message Passing in the DLWP

In the conventional LWP scheme, the message passing module of a single pod allows any pair of LWPs to communicate via a *send/receive* interface. This property is essential for supporting distributed applications and must be preserved by the DLWP mechanism. Thus, the DLWP mechanism must support communication between any two LWPs within different pods (that originate from a single application). In other words, it must preserve the message passing operations semantics across pod barriers.

In order to support interpod message passing, the mechanism must provide a method for addressing LWPs anywhere in the system. The problem is not only the naming of LWPs but, primarily, the LWP detection, due to the possible pod splitting. In terms of addressing, the problem is that the binding of LWP IDs to pods is not static: the pods split and reunite dynamically and the LWPs migrate among them. Therefore, the DLWP system must maintain a *dynamic global name space* for the LWPs, *i.e.* a global identification space that dynamically binds LWP IDs to pods' addresses at runtime.

The solution has an unavoidable cost in some duplication of information and in messages sent in the process of detection. The suggested technique employs two mechanisms: *hierarchical LWP-ID space* and *homes*. These are interwined together to reliably locate any LWP using a reasonable amount of messages.

The Home Concept

Every LWP has a *home-pod*. The home pod, unlike the *hosting* pod, in which it currently runs, does not change during the LWP's lifetime. The home pod maintains track of its LWPs by keeping up-to-date information about their *current location*. This location may change by any of the operations: *create*, *terminate*, *split*. Using the home scheme, each LWP whose home is known is located in two steps:

1. Find the home-pod of the target LWP.
2. Send a *query location* message to the home-pod and receive the current location (pod) of the LWP.

The Hierarchical LWP Name Space

LWPs form a hierarchical structure: they are created by other LWPs, via a *create* operation that causes the created LWP to become the invoking LWP's child. An initial LWP is always present - the ancestor of all LWPs - and all the activity is originally created by it.

The LWPs' name-space corresponds to the hierarchical structure of the parent-child relation. The ancestor tree is labeled in a preorder fashion, by integer chains labels. The chains represent the path from the root of the LWP-tree to the identified LWP. The *k-th* child of a node at tree level *l* is assigned the number *k* at the *l-th* position of the chain. Thus, the *k-th* child LWP has a LWP-ID composed of the parent ID (as prefix), and *k* in the last position. For example, if <> is the initial, ancestor to all LWP, then <3 , 3 , 1> is

the first child of the third child of the third child of the initial LWP.

The Homing Algorithm

The homing algorithm requires that for each LWP the home-pod of all its ancestors and direct children is kept by its current hosting pod. In addition, tracking information keeps up-to-date information about LWPs at their home pod. During a *split* operation, the following rules are applied to maintain these data:

Let pod **Cn** be a splitting pod and **Cm** be created by the split. For each LWP t that moves from **Cn** to **Cm**:

1. The home pod of t is notified of t's new location (and the corresponding tracktable entry there should be updated)
2. If $t = <t_1, t_2, \cdots, t_r>$ then the hometable entries of all prefixes - $<t_1>, <t_1, t_2>, \cdots, <t_1, t_2, \cdots, t_{r-1}>, <t_1, t_2, \cdots, t_r>$ – are copied to **Cm**.
3. For each LWP s which is a direct child of t, *i.e.* $s = <t, s_{r+1}>$ for some s_{r+1}, the hometable entry of s is copied to **Cm**.

The detection algorithm is as follows:

let $q = <q_1, \cdots, q_s>$ be a LWP which must locate $p = <p_1, \cdots, p_r>$, such that for $1 \le i \le k$: $q_i = p_i$. Then q performs the following steps:

I. Let C denote the current pod

II. for $i := k$ to r do {
    find at C's hometable the home of $<p_1, \cdots, p_i>$ and denote it by **H**
    query **H** about $<p_1, \cdots, p_i>$'s location and denote it by C
}

A detailed description of the homing algorithm and a discussion of its complexity is given in [Mal88].

4.2. Load Dispersing

The load dispersing strategy controls the distribution of the workload among the different machines. The strategy determines the threshold number of LWPs for activating the *split* operation. It also determines the conditions for coalescing two existing pods via a *merge* operation. The coalescing is done either when the number of LWPs in a certain pod drops below a minimum value or when the inter-communication between two existing pods becomes heavy. In the latter case, the pods join and immediately re-split to yield a better partition. The details of the strategy depend on the implementation, *e.g.* on the amount of memory available for the heavy-weight-process.

4.3. Memory Handling

As indicated in the previous chapter, LWPs within a single pod access two types of memory space: the *global area* and the *private area*.

The global area is not used in MOS for shared variables but rather, contains library management parameters and constants. Therefore, different pods need not synchronize accesses to this area and simply need to duplicate it during the split operation.

The private area consists the runtime local space of the LWPs. The private memory portions of the LWPs construct a Cactus Stack (tree structured) corresponding to the LWP hierarchy. The tree structure facilitates sharing of the inner parts by outer parts that directly connect to them. Thus, pods are responsible for keeping replicas of memory portions of all of their LWPs' ancestors. This guarantees access of LWPs to their ancestors' data. Since the LWP system does not support *shared variables*, these memory portions are shared only for *reading* and not or *modifying*. The replicas consistency is maintained by not allowing a parent LWP (which is allowed to modify its private memory) to run in parallel with its children LWPs. The parent must block on children creation and await their termination before resuming operation.

5. Partitioning Algorithm

The splitting of LWP pods provides concurrency through parallel execution, at the cost of the resulting interpod communication. Therefore, its efficiency depends on a good split of the load while minimizing the interpod communication. In MOS, the partitioning is done by collecting profile information of the LWPs behavior during their execution. The partitioning strategy is based on the past behavior of the LWPs in terms of CPU consumption and communication with other LWPs. The goal is to use these parameters to achieve a partitioning algorithm that, assuming *past repeat* behavior, will split the load equally among two pods and will also result in minimum interaction between the two partitions. This prediction information is hardly exact. However, it is preferable to advance declarations of the user about the amount of resources consumed by each activity. In addition, the exact data is irrelevant since the mechanism employs heuristic, suboptimal algorithms.

Hill Climbing Algorithm

The profile information can be represented as a graph, in which the vertices' weights (sizes) represent the respective LWPs' CPU consumption and the edges represent the amount of communications between the LWPs. Communication accounting is taken as the weighted connectivity matrix of the LWPs graph. In this form, the partitioning problem becomes a *graph partitioning* problem. It is easy to verify that a graph partitioning that minimizes the cut-size assures minimum interaction between the represented LWPs. The load balancing considerations are incorporated into the algorithm through the size constraints; graph partitions that are balanced in size represent equally loaded LWP pods.

The *hill climbing* algorithm starts with a given, random partition. It then iterates *exchange* steps over a given partition; an exchange step consists of swapping between two vertices from the two partitions. In each iteration of the loop, degrading steps are rejected and only steps that improve the current partition are accepted. Experiments with the hill climbing algorithm show that it converges rapidly to almost optimal solutions for random graphs with up to 300 vertices ([Mal88]).

6. Implementation Details

LWPs are implemented within a single MOS process that contains the code for LWP management. The environment and resources of a LWP pod are those of a single MOS process. The LWP services are provided by a *library* of routines, written in C. Any program may gain LWP services by linking its object modules with this library and by requesting the services via the interface routines.

Software Architecture

The DLWP provides various services, implemented by the following modules:

- *The Scheduler*
  Support the coexistence of program parts as virtual, light weight processes. The running time is divided between the ready-to-run LWPs by alternating their execution. The scheduling is *asynchronous*, *i.e.* the LWPs execute different amounts of time between process switching.
- *Sleep/Wakeup Mechanism*
  This module provides atomic synchronization operations for the LWPs. At any stage, an existing LWP may be *ready-to-run* or *waiting*. Processes may switch between states using the sleep/wakeup primitives. Using these primitives, other synchronization mechanisms may be implemented. For example, the LWP *message passing* module uses them for synchronizing operations.
- *LWP Message Passing*
  Communication between LWPs is enabled by the message passing module. This module provides asynchronous communication services through structured *Send/Receive* operations. Any two LWPs may communicate through this mechanism.
- *Split*
  A pod may decide to split itself into two duplicate pods and divide the load between the two. The splitting module is responsible for duplicating the pod and for performing the data manipulations required for the division.

- *The LWP Router*
  The routing module locates any LWP in the DLWP system. The message passing mechanism uses this service to forward LWP messages to their destination.
- *IPC*
  Pods interact to exchange load information and to pass LWP messages. The interaction is implemented by using the IPC mechanism which provides structured communication between pods.
- *Interaction between pods*
  This module exchanges information between pods and carries out their local decisions. The pods exchange load information in order to make split/join decisions. They also maintain dynamic tracking information about the LWPs for the *homing* scheme.

DLWP Primitives

The following interface routines implement the DLWP system and provide the caller with the LWP/DLWP services:

`char *Getlwpid()`	- return the calling LWP's id (an integer vector)
`Lwpclose()`	- closing routine of the DLWP system
`Lwpinit()`	- initializing routine of the DLWP system
`Process *Maplwpid(id)`	- return the process structure of the spcified LWP (containing information maintained by the system about the LWP)
`char *Receive()`	- Receive a message (character buffer) Blocking call
`Ident Route(from, to)`	- Find the location of the destination LWP (to). Used for low level communication; routing is done automatically using 'Send/Receive'
`Send(to, data, len)`	- Send a message to the specified LWP
`Slp(event)`	- Sleep on 'event'
`Spawn(nf, func)`	- Create a child process that runs 'func'
`Spawnn(nf, func)`	- Create 'n' children processes that run 'func'
`Spawnv(nf, vfunc)`	- Create 'nf' children processes that run the corresponding elements of 'vfunc'
`Terminate()`	- Terminate the calling LWP (the call does not return)
`Waitchilds()`	- Wait for all children termination
`Wakeproc(p, event)`	- Wake the specified LWP on 'event'
`Wakeup(event)`	- Wake all LWPs sleeping on 'event'
`Yield()`	- Give up execution control

In addition, a *split()* operation is available for explicit invoking (in addition to the automatic activation when the LWPs pod becomes "too heavy").

6.1. The Hard Issues

There are considerable difficulties with the implementation of LWPs in general and with the implementation of DLWP in particular. Most of the difficulties arise in the LWPs implementation and result from the UNIX semantics. These are briefly described here; [Kep85] gives a more detailed description. Note that all of these difficulties may be overcome, but sometimes, the solution is so cumbersome that it is preferable to leave things as they are and notify the user.

Blocking System Calls

UNIX system calls such as *read*, *write*, *pause* may block for an arbitrary amount of time. Thus, one LWP within the pod performing such a call might suspend the whole pod for that duration and prevent it from executing. The problem is overcome by linking the program with a special library which provides a non-blocking version of these system calls.

Signals and Timeouts

UNIX supports one timer interrupt and one signal handler (per signal) in a process. The LWPs may request for different signal handlers or a few timeouts. This results in the later calls overriding existing ones. Again, a system call library which supports multi timeouts and selects interrupt handlers according to the LWP context solves this problem.

Global Memory Access

The LWPs in a pod have access permissions to the entire process' memory. Thus, there is no protection of the LWP local memory segments just as there is no protection of the library's global variables. In view of the split operation, which separates the accessible space of some of the LWPs from others, it seems best to require the programmer to restrict access to the local data. Of course, bugs resulting from loose pointers that go out of bounds are very hard to detect with this "solution".

Shared State Variables

LWPs may access state variables that effect the whole pod. The access may be implicit in a procedure call. Typical examples in UNIX are the *file offset*, kept in the file descriptor of the process' open files, which is modified by the *read*, *write* and *seek* calls, and the *errno* variable that holds the return value of system calls. LWPs that perform seek-read/write sequences all effect these variables and need to be synchronized. [McS87] suggest a few ways to overcome this problem. Two of them are:

1. Avoiding the state variables altogether. For example, in the file offset instance, add an offset parameter to the read/write calls.
2. Making the state variables part of the LWPs' private context. Their values are to be looked up by the concerned procedures.

7. Performance

The performance of the system was tested to give the "raw" system performance, *i.e.* the minimal cost of system operations such as process creation, context switching, etc. The "stripped" version of the system was used and the following costs were measured (in CPU time, both user and system time):

1. System *startup* (expected to be negligible, but should be subtracted from any further measurement)
2. LWP creation time and space
3. LWP context switching, the `Yield()` operation
4. Message passing

The test programs were run 10 times for each checked value; the recorded results are the average of those runs. The programs were run with different amounts of LWPs, to check their effect on the complexity of the various algorithms. Table 1 is a summary of the results; the net overhead per operation was computed from the measured times by subtracting the startup cost and dividing by the number of operations performed.

no. of LWPs	1		20		100		300	
	user	sys	user	sys	user	sys	user	sys
startup	100	1000	100	1000	100	1000	100	1000
LWP create	0	0	3	11	6	7	13	9
context switch	0.1	0	0.1	0	0.1	0	0.1	0
Yield	0.4	0	0.7	0	5	0	12	0
local message	4	0	5	0	9	0	15	0
remote message	4	40	7	45	11	45	17	40

Table 1 - LWP operations times, in milliseconds

8. Conclusions

We have presented the Distributed Light Weight Processes mechanism, a facility for supporting ditributed programs in MOS. The main features of the mechanism are:

dynamic configuration

The mechanism utilizes memory and CPU according to the application's needs. In particular, the binding of threads to CPUs is done dynamically by *splitting* at runtime. This property relies on the ability of the underlying operating system to dynamically assign processes to processors. In the current implementation, the MOS dynamic load balancing facility provides the required flexibility.

massive parallelism

The mechanism is designed to provide massive support of parallelism, free from the hardware constraints. The threads may be scheduled in a time-sharing scheme when they exceed the available concurrency in the system. Thus, the only limit on their number is their total consumption of memory.

applicability

The mechanism provides services via a set of general purpose interface routines. This set may be easily modified or added to, *e.g.* to allow different communication styles. This allows the mechanism to support a variety of parallel programming languages in providing the underlying runtime support for precompiled programs. For example, an occam support package has already been implemented on top of a single machine LWP system [MaS86] and is now being ported to MOS to use the DLWP mechanism.

portability

The DLWP library is written on a UNIX compatible system, in a high level programming language (c).

References

[ABG86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, Mach: A New Kernel Foundation for UNIX Development, Technical Report, Carnegie Mellon University, August 1986.

[ACG86] S. Ahuja, N. Carriero and D. Gelernter, Linda and Friends, Computer , 19(8), Aug 1986, pp. 26-34.

[BaL85] A. Barak and A. Litman, MOS: A Multicomputer Distributed Operating System, Software - Practice and Experience, 15(8), Aug 1985, pp. 725-737.

[BaS85] A. Barak and A. Shiloh, A Distributed Load Balancing Policy for a Multicomputer, Software - Practice and Experience , 15(9), Sep 1985, pp. 901-913.

[Kep85] J. Kepes, Lightweight Processes for UNIX Implementation and Application, in *Usenix technical conference proc.* , Usenix Association, Portland, Or, Summer 1985, pp. 299-308.

[MaS86] D. Malki and G. Shwed, A Unix OCCAM+ System, Technical Report CS-87-7, Hewbrew University of Jerusalem, Jerusalem, Il, June 1986.

[Mal88] Dalia Malki, Distributed Light Weight Processes for MOS, M. Sc. Thesis, Dept of CS, Hewbrew University of Jerusalem, Jerusalem, Il, June 1988.

[McS87] P. R. McJones and G. F. Swart, Evolving the UNIX System Interface to Support Multithreaded Programs, Research Report, Digital Systems Research Center, Palo Alto, CA, Sep 1987.