EXODOS
Oregon Experimental Distributed Operating System

Jeff Eaton
Virginia Lo
Bill Nitzberg
George Rankin
Mark VandeWettering

CIS-TR-88-07
June 11, 1988

exodos@cs.uoregon.edu

Abstract

EXODOS is a distributed operating system we are developing at the University of Oregon. Our goal is to implement a portable, easily modifiable testbed for research in distributed operating systems. The EXODOS kernel provides mechanisms to support basic functionality without mandating policies, for it is these policies that we wish to study. The EXODOS kernel is replicated on every node and supports *local* interprocess communication, primitive memory management, and basic process management. Higher level operating systems functions are provided by server processes which interact with the EXODOS kernel through a unified message-passing interface. By elevating these functions to the server level, EXODOS provides a convenient testbed for the study of policy-level algorithms for scheduling, process migration, and virtual shared memory.

Department of Computer and Information Science
University of Oregon

1 Introduction

EXODOS is a distributed operating system being developed at the University of Oregon to support research in the design of policy-level algorithms for scheduling, process migration, and virtual shared memory. Most research in the areas of scheduling, load balancing, and process migration, have been conducted through simulation and performance analysis on a uniprocessor machine. For the most part, these studies have focused entirely on policy issues and have consciously ignored implementation concerns. EXODOS arose from the desire to augment this work through the design and testing of policy-level algorithms on a functioning distributed operating system. Experimentation through implementation will yield an understanding of implementation factors that affect policy decisions and the complex interactions that exists between policy and mechanism. EXODOS will also enable us to empirically test scheduling and load balancing algorithms that have previously been validated in the sheltered environment of simulation.

EXODOS consists of a simple, minimal kernel, replicated on each node, and a number of server processes distributed among the nodes of the multicomputer network. The kernel supports *local* message-passing, physical memory management, and basic process management. Other operating system functions are carried out by servers: *non-local* message passing, virtual memory, scheduling, process migration, device drivers, file management, and protection. EXODOS utilizes message-passing as the fundamental mechanism for user and server processes to request kernel services, for kernel-to-kernel and kernel-to-server communication, and for general interprocess communication. Another fundamental mechanism used in EXODOS is exception handling: specific conditions give rise to faults which are detected by the kernel and directed to the appropriate exception handler which in most cases is a server process. EXODOS faults include *communication faults* and *scheduling faults* as well as traditional page faults. The design of EXODOS was guided strongly by the principles of *simplicity, uniformity, orthogonality,* and *modularity.*

In this section, we discuss EXODOS research goals, design goals, and the history and current status of EXODOS. Sections 2 through 4 describe EXODOS interprocess communication, memory management, and process management in detail. This technical report will evolve dynamically along with EXODOS.

1.1 EXODOS Research Goals

EXODOS was designed to support a number of research projects in the area of distributed operating system design. A distributed system provides reliability, resource sharing, and parallelism in a convenient, transparent environment. While existing distributed systems have achieved much of their capacity for information sharing through distributed file systems and performance-efficient message-passing, the use of distributed systems for sharing of processing power is still an open area of research. Our research focuses on means to achieve the full power of compute sharing in distributed systems through load sharing and load balancing and through the use of distributed systems for (large-grained) parallel processing. More specifically, we are interested in policy-level algorithms to guide and provide dynamic process migration and virtual shared memory in distributed systems. EXODOS was designed to serve as an experimental laboratory for the testing of algorithms in these domains. Projects that are currently planned for experimentation on EXODOS are described briefly below:

- **Process Migration** One mechanism for achieving improved performance in distributed computing systems is *dynamic process migration*, the movement of processes from one node to another in response to dynamic changes in system loads or due to the dynamic behavior of distributed computations. We are engaged in an empirical study of existing distributed computations to determine what kinds of process traits are important for migration decisions, the difficulty and cost of ascertaining these traits, and the relation between static predictions of process behavior and the actual dynamic characteristics exhibited at execution time. Knowledge about these process traits will be used in the design of migration algorithms to decide *which* process to migrate and to guide migration based on both static and dynamic interprocess communication patterns.

- **Distributed Virtual Shared Memory**[1] Experience with distributed computations indicates that the paradigm of shared memory is sometimes more appropriate than that of message-passing. It has been demonstrated by researchers in this area that such a system can be implemented efficiently and that parallel programs run on such a system

[1] Distributed virtual shared memory provides the abstraction of physical shared memory on loosely coupled processors.

yield linear and sometimes super-linear speedup [Li86]! We are investigating extensions to this work in the areas of prepaging algorithms and memory coherence algorithms for virtual shared memory systems, and policies and mechanisms for process migration in a virtual shared memory system.

- **Scheduling** Process migration can be subsumed into the more general notion of process scheduling by considering the dimension of space as well as that of time in the scheduling model. Thus, a scheduling server has access to scheduling queues which may be located on many of the nodes in the distributed system. We are interested in the integration of process migration and process scheduling and in determining whether an integrated approach offers benefits over the traditional treatment of scheduling and migration as disjoint operations.

While EXODOS was designed to support the research projects outlined above, we recognize that its development will yield insights in other areas, particularly in the design and implementation of distributed operating systems. In particular, we note that EXODOS is unique in assigning the functions of nonlocal interprocess communication and process scheduling and migration to the server level. We will be investigating the performance implications of this design decision and some interesting innovations that result from this arrangement. For example, several different memory management servers can co-exist, each providing its unique memory mapping scheme. Furthermore, a single process can utilize more than one type of memory server for distinct portions of its address space. This flexibility also occurs in EXODOS interprocess communication and scheduling, and is discussed in more detail in sections 2-4.

1.2 EXODOS Design Goals

The underlying goal behind EXODOS is to create a distributed operating system for a multi-computer network to serve as a testbed for research in the areas of scheduling, process migration, and virtual shared memory. This goal has focused EXODOS' development on the areas of interprocess communication, memory management, and process management. Other areas of distributed operating system management such as fault tolerance, the file system, and security have either been ignored or minimally addressed. In addition, the underlying goal to build an experimental testbed has led us to a structure for EXODOS which is strongly guided by design principles

which promote testing of many policy-level algorithms with a minimum of software development overhead. In this section, we describe the design principles which guided EXODOS development: simplicity, minimality, regularity, orthogonality, and modularity, In addition, we discuss areas we deliberately chose not to emphasize for EXODOS.

- **Minimality** The EXODOS kernel should be simple and minimal. The kernel supports *local* message-passing, physical memory management, and basic process management. These functions represent the minimal support needed for experimentation with a variety of policies in the areas of our research interests. Specific policies are implemented at the server level, decreasing the amount of time needed to develop, alter, and test portions of the operating system.

- **Regularity** EXODOS is regular and consistent. EXODOS supports message passing as the lingua franca for process-to-kernel communication, kernel-to-kernel, kernel-to-server, and for process-to-process communication. Exceptional conditions cause a fault to occur and the appropriate action is taken by the kernel and/or servers. For example, because the kernel supports only local interprocess communication, messages sent to a remote process result in a *communication fault*, causing the kernel to send a message to the sending process' IPC server.

- **Orthogonality** The basic objects in EXODOS are processes, pages, and messages. The fundamental operations are creation, termination, transfer (process migration, paging across the network, and sending messages) and access or use (process scheduling, mapping of pages, and message receipt). In a virtual shared memory system, this orthogonality enables use to provide process migration and transfer of large messages through the page fault handling mechanism.

- **Modularity** Assignment of most operating system functions to the server level allows easy removal and replacement of modules. EXODOS provides a "software backplane" for experimentation with a variety of servers within each domain of interest. For example, load balancing servers using a sender-initiated protocol could be replaced by servers using a receiver-initiated protocol without rebooting the system! In addition, more than one protocol could be active simultaneously by having one subset of nodes utilize the servers providing sender-initiated

load balancing, and a different subset of nodes using servers providing receiver-initiated protocol.

It is also important to discuss what EXODOS is not. While it is imperative that EXODOS provide efficient local IPC because of the use of message-passing for kernel access, we are willing to trade some degree of efficiency for non-local IPC in return for the flexibility of handling it at the server level. We have also accepted performance tradeoffs because EXODOS is not intended to be a production level system. Finally, we have deliberately de-emphasized protection, the file system, and naming because they have been shown to be conveniently handled at the server level in other systems.

1.3 History

Our decision to design a new distributed operating system resulted from our study of a number of well-known existing distributed operating systems including V [Che88], Mach [TR87], Amoeba [TM86], Charlotte [FSK*86], Sprite [NWO88], Eden [ABLN85], Accent [RR81], and Locus [PW85]. We considered modifying an existing system for our research as the most time-effective way to build our desired testbed environment. However, the following factors led us to design and build EXODOS:

- Most of the above distributed operating systems are too big and complex for us to conveniently modify. These systems are in active use for normal computing as well as for research. CMU's Mach, for example, includes much of the Berkeley UNIX 4.3 kernel and contains over 100,000 lines of code. The size and complexity of a full-blown kernel implies a major investment of time and resources. Because we did not wish to provide a "complete" operating system, we were willing to ignore or de-emphasize functions such as fault tolerance and protection and we were willing to trade off much of the flexibility and convenience offered by these complex operating systems in order to better understand the operating system as a whole.

- The design of these existing operating systems is not well-suited for our specific research goals. Most of the above systems do not treat process management, interprocess communication, and memory management orthogonally, a design decision we found to be necessary for a clear study of the issues of process migration and virtual shared memory.

For example, in V only processes that are members of the same *team* can share memory and they must migrate as a group.

EXODOS is written in C for Tektronix 4404 graphics workstations and will be ported to Sun 3/60 workstations.

2 Interprocess communication

Interprocess communication is the cornerstone of EXODOS. By virtue of its client/server design, all services in EXODOS are provided via the message passing mechanism. This gives EXODOS a strong unifying design element that is absent in other systems, but also poses some special challenges. IPC must be efficient because every service is based upon it. EXODOS builds a simple and efficient basic IPC mechanism with the ability to use servers for more powerful abstractions.

2.1 Goals

In addition to the overall EXODOS goals, the IPC primitives are expected to be transparent, locally efficient, reasonably efficient for remote traffic, and network-independent.

- **Transparency** Sending a message to a remote process should appear to be the same as sending to a local process. This is essential because EXODOS supports migration: there is no guarantee that any two processes will continue to reside on the same node.
- **Efficient Local Messages** All services are implemented with message passing. The overhead of local message-passing should not significantly lengthen the response time of system services, when compared to procedure call.
- **Reasonable Costs for Remote Messages** Process migration and distributed problem solving are important EXODOS goals, so the cost of remote IPC shouldn't be prohibitive. The efficiency of remote IPC is secondary, however, to local IPC efficiency.
- **Network Independence** Reliable network transmission is a difficult task that should be implemented by server processes, which are easier to build and debug than kernels. Allowing multiple servers should allow support for multiple inter-machine protocols.

2.2 Kernel Primitives for IPC

The EXODOS kernel provides primitives for the transfer of fixed-sized, untyped messages between local processes. Messages are unbuffered and blocking; the sender blocks until it receives a reply. This mechanism is similar to remote procedure call [BN84]. Additional functions, such as typed, non-local, or variable-length messages, will be provided at a higher level by IPC servers.

Only the message-passing primitives themselves are implemented as system calls to a local kernel; all others are implemented as messages to *any* kernel, either local or remote.

`SEND` Sends a fixed-length message to the specified process-id. The sending process blocks until some process replies to the message (see `FORWARD`). The sender may not be migrated while it is awaiting a reply.

`RECEIVE` The process executing the `RECEIVE` blocks until a message is ready for it. The receiver must either `REPLY` to the message or `FORWARD` it before it can execute another `RECEIVE`. Processes may be migrated while blocked for a `RECEIVE`.

`REPLY` Unblocks the sending process and gives it the fixed-length reply message. If the sender has been killed while awaiting a reply (nothing else may happen to it), the reply will fail.

`ERROR_REPLY` Allows the receiver to force the sender's `SEND` to return an error code.

`FORWARD` This call allows a process to transparently forward messages to other processes. Neither the original sender or the new receiver know that the message has been forwarded. This call does not block.

2.3 IPC servers

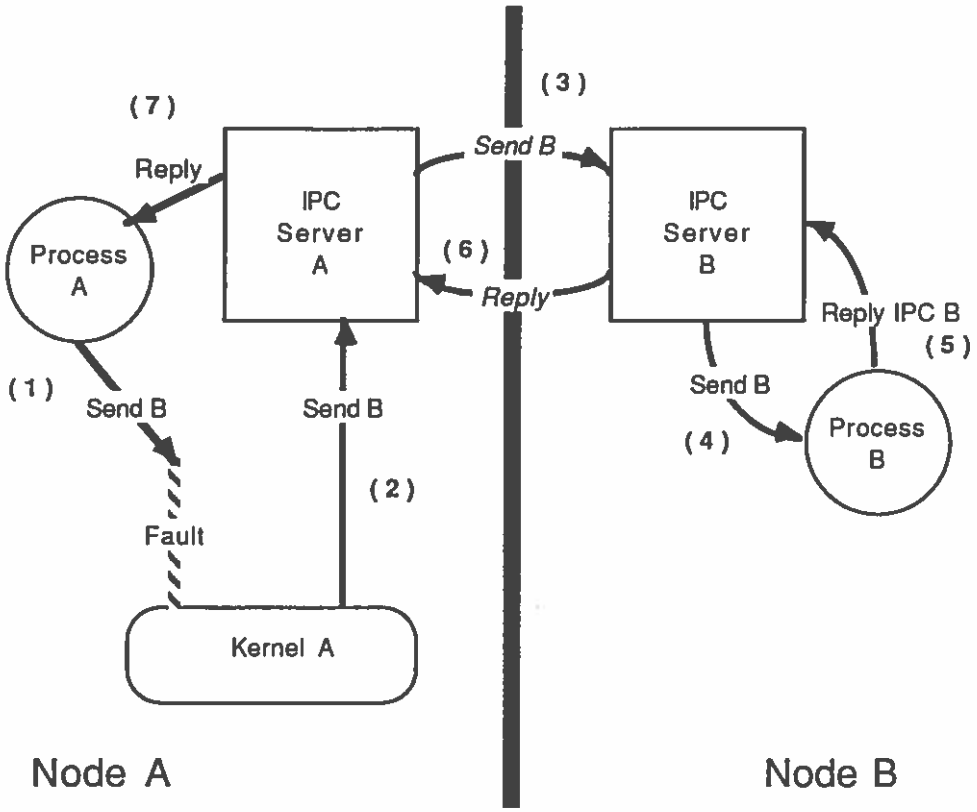
When the destination of a message is not a local process, the kernel generates a *send fault*. This causes the message to be forwarded to the process' designated IPC server.[2] How the message is delivered, and who it is really delivered to, is a decision left to the server. This allows multicast and group

[2] If a process' IPC server is not local, the message is passed to the *kernel's* IPC server—which must be local.

messages to be implemented: the server may interpret the process ID as a group identifier and then send the message to each group member.

We now present an example of how non-local IPC may be implemented. This example is very simplified: the server blocks until it gets a reply from the remote process. In actual practice, the IPC servers would invoke child processes to do their message-passing, and would communicate with processes that control the network hardware. Such a real design would rely on an inexpensive process creation mechanism, which is discussed in section four.

2.4 An Example: Non-local Communication



1. Process A sends a message to process B. Process B is not local, so an IPC fault happens.
2. The kernel doesn't know process B, so it forwards the message to A's IPC server.
3. The IPC server determines B's location, and sends the message across the network to IPC server B.
4. IPC server B sends the message to process B.
5. Process B replies. The reply is locally delivered to IPC server B.
6. The reply is sent back across the network to A's IPC server.
7. A's server replies to process A, and A continues.

Figure 1: Non-local Communication

2.5 IPC design decisions

Unlike other distributed systems, the EXODOS kernel supports only small, fixed-length messages. Not only does this simplify the kernel design, but it allows policy decisions to be made at a higher level. We noticed that the implementation of arbitrarily large message transfers fit a more general shared memory design, especially when efficient copy-on-write implementations are considered. Since it is easy to see how distributed shared memory (or multiple `SEND`s) can be used to implement this type of IPC, EXODOS leaves the problem to the higher-level servers.

We decided to leave the questions of security and reliability to the IPC servers. The kernel only ensures reliable communication between two processes on the same node; it is up to the client processes to select an IPC server that provides the correct level of inter-network reliability for their needs. Security is handled in the same manner. EXODOS does not provide a *from* field for message authentication because the mechanism would not work when two processes were remote. Instead, security must be provided at a higher level with IPC servers. Whether or not this is possible may prove to be an interesting research question.

3 Memory Management

The main goal of EXODOS memory management is to provide support for many diverse memory abstractions. It must be possible to design, test, and utilize new memory management ideas with relative ease. In particular, EXODOS needs to be powerful enough to implement and study the effects of distributed virtual shared memory.

Memory management in EXODOS is supported by servers and the underlying memory management functions of the EXODOS kernel. The kernel provides access to two basic resources: a client's virtual address space and the physical memory of the kernel. The kernel handles page faults by notifying the memory server in charge of the fault address. The server will repair the fault, and then the client will continue.

3.1 Memory Servers

The real work of EXODOS memory management is done outside the kernel by servers. Memory management is accomplished through the combined activities of one or more memory servers and the kernel on which the client

process resides. When a page fault occurs, the kernel sends a message to the server assigned to manage the missing page. The server will then use its particular memory abstraction to load the page onto the client's node. Specifically, the server can invoke kernel primitives to transfer responsibility for a process's virtual address space (SETVIRTUALRANGE), get a node's physical memory (GETPAGE and FREEPAGE), and to map the virtual pages to physical pages (MAPPAGE and UNMAPPAGE.)

Under EXODOS, a user application process (the client) utilizes the services of memory servers that implement the desired memory abstraction. In a demand paged memory management scheme, for example, the memory server might load pages from a fileserver. For virtual shared memory, the missing page might be loaded from the physical memory of another node. There is no restriction on the location of the server which manages a given virtual page.

The EXODOS kernel enables processes to share memory conveniently and efficiently. One server obtains ownership of a set of physical pages through the GETPAGE primitive. The page map of each of the processes sharing these pages all point to this server. If a process resides on the node whose physical memory contains the shared pages, the corresponding page map entries will indicate the correct mapping. If a process resides on a different node, the page map entry will be *invalid* and a page fault will occur when the missing page is referenced. As a result, the appropriate server will be contacted to get the page.

EXODOS memory management also provides primitives to facilitate migration of processes. Memory can be transferred between nodes at any time by memory servers talking to network servers. In a virtual shared memory system, the copying could be initiated explicitly through pre-paging or implicitly (dynamically) as the page faults occur on the new node. After a process migrates, it is usually more efficient to use a server residing on the destination node. SETVIRTUALRANGE transfers control over of the client's virtual pages from the calling server to the target server.

3.2 Kernel Primitives for Memory Management

SETVIRTUALRANGE Transfers control over the virtual pages of a client process to another server process.

GETVIRTUALRANGE Returns a list of virtual pages that server processes have control over for the client process.

`GETPAGE` Gives a server exclusive mapping control over a physical page of memory.

`FREEPAGE` Releases exclusive mapping control over a physical page.

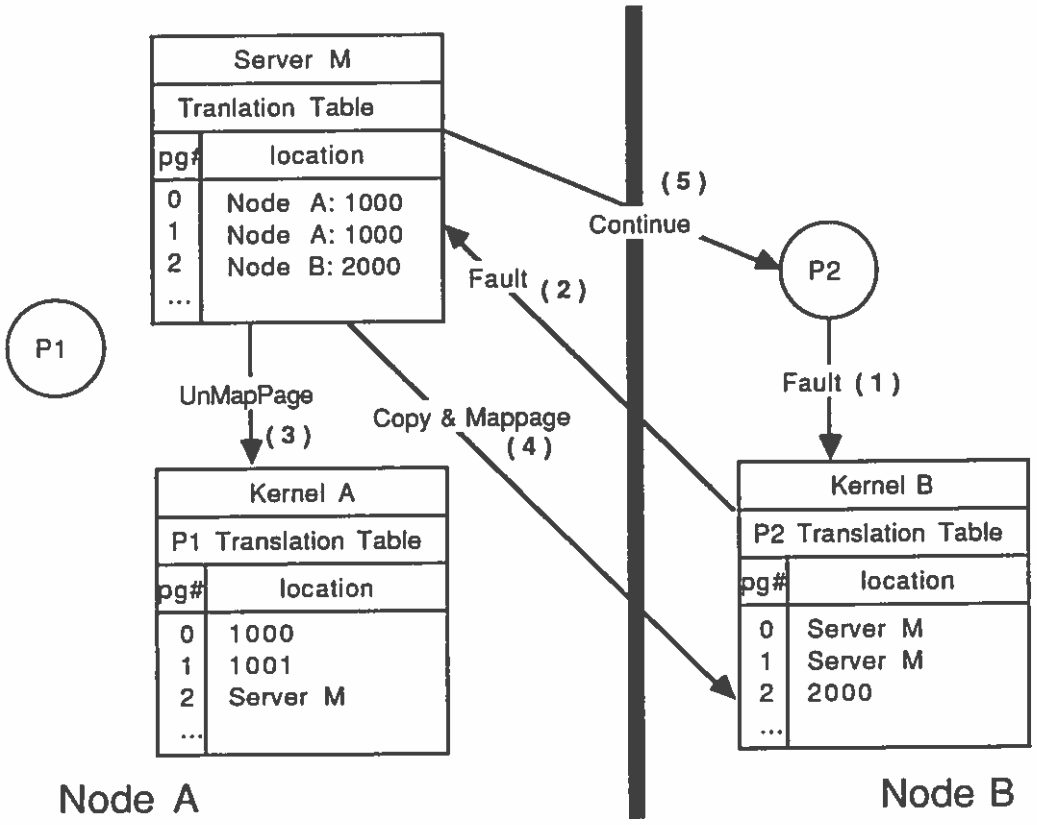
`MAPPAGE` Maps a virtual page of a client to a physical page of a server, with or without access restrictions.

`UNMAPPAGE` Removes the mapping of a virtual page of a client process.

`SETPAGEOWNER` Transfers mapping control of a physical page to another server process.

`GETPAGEINFO` Returns information about a physical page.

3.3 An Example: Distributed Virtual Shared Memory



1. Process *P2* on Node *B* tries to access page 0 of its address space. This causes a page fault.

2. Kernel *B* translates the page fault into an exception message and sends it to Memory Server *M*.

3. The server invalidates the shared page in process *P1*'s translation table.

4. It then copies the physical page onto Node B, and maps it into *P2*'s address space.

5. Server *M* resumes process *P2*.

Figure 2: Distributed Virtual Shared Memory

4 Process Management

Process management is the creation, termination, and scheduling of processes in the operating system. EXODOS is unique because it introduces the idea of scheduling at the server level, by means of the exception mechanism.

4.1 Goals

The main goals of EXODOS process management are to support light-weight processes, diverse scheduling policies, and preemptive process migration.

- **Light-weight Processes** The distinction made in other operating systems between processes and threads is unnatural, and only made for efficiency. An EXODOS light-weight process is a combination of a traditional "heavy-weight" process with the potential efficiency of a thread.[3]

- **Diverse Policies** As a research oriented operating system, EXODOS must support the development and testing of new scheduling policies. It must be relatively easy to develop a new scheduler, and install and test it. In addition, as EXODOS is a distributed operating system, support for remote scheduling is necessary.

- **Preemptive Migration** One important motivation for EXODOS was to provide a base for investigating process migration. Process migration has been an area of recent interest (Amoeba [TM86], Emerald [JLHB88], Charlotte [ACF86], and V [Che88].) EXODOS provides primitives for implementing both user-directed and high-level, system-directed process migration.

4.2 Process Scheduling

The kernel provides two independent process management mechanisms: one for process scheduling, and one for process creation and termination (also used for migration). The kernel provides a rudimentary priority queue based scheduling mechanism. Every process has three scheduling parameters: priority, CPU allocation, and quantum. The priority determines which process will run; the CPU allocation is the total amount of CPU time a process will

[3] A separate mechanism is used to determine when light-weight processes share their address space. When this is the case, the kernel will be able to context switch between them with very little overhead.

get; and the quantum is the time slice for a process. The kernel scheduler executes the following loop:

```
Let P be the head of the process queue
Run P for at most quantum time units
Decrement the CPU allocation by the execution time
Put P back into the process queue,
        behind processes with the same priority
```

If a process blocks or runs out of CPU allocation, it is removed from the process queue until it is unblocked or given more CPU allocation. Notice that the kernel scheduler does not recompute priorities, but will run processes with the same priority in a round robin fashion.

High level scheduling is handled through the exception mechanism in EXODOS. Just as a process can have a memory fault or an IPC fault, it can also have a *scheduling fault*. Scheduling faults are generated when a process is: installed, terminated, blocked sending, blocked receiving, unblocked, and out of CPU allocation. For efficiency, each of these faults can be independently disabled.

Every process has an associated scheduling fault handler. This scheduling server is notified (via standard exception messages) whenever the state of a client process changes.

The scheduling server can change priorities, migrate processes, or give more CPU allocation to processes, implementing any policy desired. Using the exception mechanism for high level scheduling allows an EXODOS system to have multiple schedulers per node, remote schedulers, centralized schedulers, and fully distributed schedulers. In addition, as the scheduling server is determined on a per process basis, new servers can be easily installed and debugged *on-the-fly*.

In addition to the fault messages, scheduling servers can use *scheduling assistants* to obtain information. An example of a scheduling assistant is a process that sends a message to the scheduler every time it is run. By installing this process with priority *PRI*, the scheduler will be notified whenever all processes with priority greater than *PRI* are inactive.

The kernel scheduling mechanism is one place where minimality and simplicity were overshadowed by the need for efficiency. The minimal approach would be to have the process queue only exist in the scheduling server, not in the kernel. The only kernel scheduling primitive would be to execute a process for a particular amount of time (a quantum). A typical server would perform the scheduling loop above, however,

```
Run P for at most quantum time units
```

would be a kernel call. This method would have been too inefficient, and suffered from a serious drawback: there was no way of scheduling the scheduling server. To avoid these problems, EXODOS provides rudimentary support for scheduling in the kernel.

4.3 Process Packages

An EXODOS process is defined by a process package. A process package contains a snapshot of the kernel process control structure. It includes a globally unique process ID, user ID, a list of fault handler IDs, and a few miscellaneous parameters including a machine dependent process control block. Processes are created using the `INSTALL` kernel call, passing a process package as a parameter, and destroyed using the `TERMINATE` call. Process packages can be created with appropriate parameters, or by using the `PACKAGE` kernel call on an existing process. Migration is accomplished by `PACKAGE`ing a process, `SEND`ing it to a remote node, `INSTALL`ing the package on the remote node, and `TERMINATE`ing the process locally.

Migration is not the only use of the package facility. Checkpointing can be accomplished by obtaining the process package of a running process and saving it to secondary storage. A fork system call can be simulated by packaging a process and installing it with a new process ID. Packages also have uses in debugging, as they contain, among other things, the program counter, stack pointer, and other state information of the process.

4.4 Kernel Primitives for Process Management

`INSTALL` Creates a new process by installing a process package. The new process is created in the `SUSPEND`ed state, and must be `RESUME`d before it will start executing.

`TERMINATE` Kills a process.

`PACKAGE` Returns the process package of a process. The process must be `SUSPEND`ed before it can be `PACKAGE`d.

`SUSPEND` Stops a process from executing. A process cannot be `SUSPEND`ed if it is awaiting a `REPLY` message. This prevents a process from being `PACKAGE`d, which would make it unable to receive the reply message.

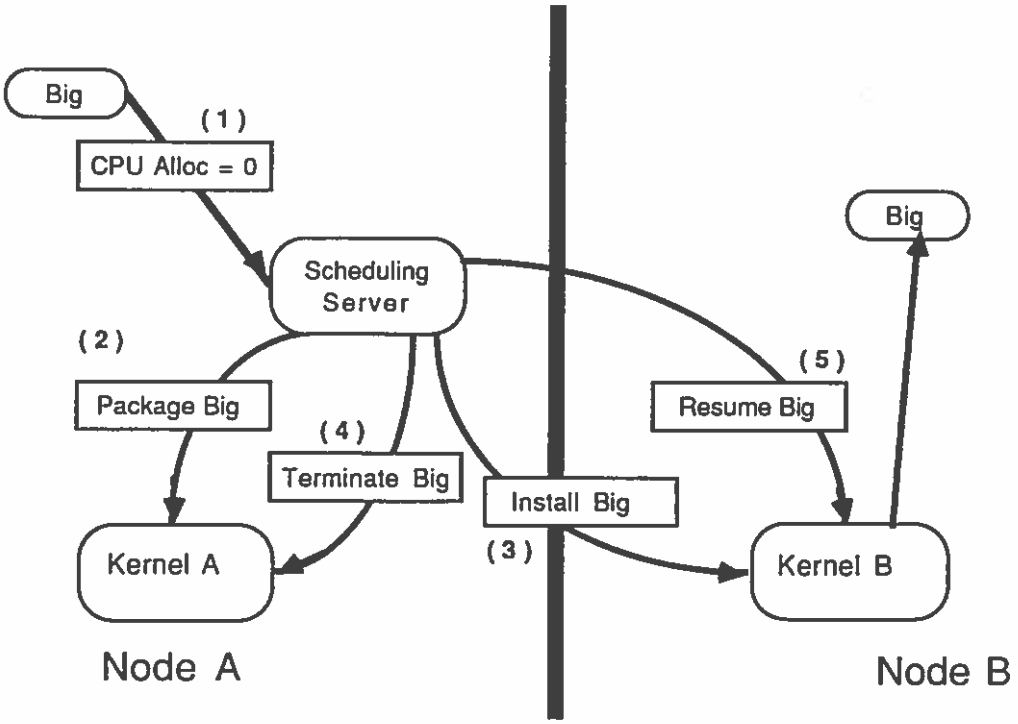
`RESUME` Allows a `SUSPEND`ed process to continue executing.

SET_PRIORITY Changes the priority of a process. There is also a GET_PRIORITY call.

SET_CPUALLOC Changes the amount of CPU time allocated to a process. After a process executes for its CPU allocation, it is automatically SUSPENDed. There is also a GET_CPUALLOC which returns the amount of allocation remaining for a process.

SET_QUANTUM Changes the size of a process's time slice. There is also a GET_QUANTUM call.

4.5 An Example: Migration



1. Process *BIG* runs out of CPU allocation, and a fault message is sent to *BIG*'s scheduling server.
2. The scheduling server determines that *BIG* is a good candidate for migration, and locates a suitable remote node. It then PACKAGEs *BIG*.
3. The server INSTALLs *BIG* on the remote node. (Now there are two *BIG* processes, one on each node.)
4. The server TERMINATEs the local *BIG*.
5. Finally, the server RESUMEs the remote copy of *BIG*, and the migration is complete.

Figure 3: Migration Example

5 Summary

$$\sum_{i=0}^{\infty} \mathsf{EXODOS}_i$$

References

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.

[ACF86] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel. *Processes Migrate in Charlotte.* Technical Report 655, University of Wisconsin Dept. of Computer Science, August 1986.

[BN84] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Che88] D. Cheriton. The V distributed system. *CACM*, 31(3):314–333, March 1988.

[FSK*86] R. A. Finkel, M. L. Scott, W. K. Kalsow, Yeshayahu Artsy, Hung-Yang Chang, Prasun Dewan, Aaron J. Gordon, Bryan Rosenburg, Marvin H. Solomon, and Cui-Qing Yang. *Experience with Charlotte: Simplicity versus Function in a Distributed Operating System.* Technical Report 653, University of Wisconsin Dept. of Computer Science, July 1986.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University Dept. of Computer Science, September 1986.

[NWO88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[PW85] G. J. Popek and B. J. Walker. *The LOCUS Distributed System Architecture.* MIT Press, 1985.

[RR81] R. F. Rashid and G. G. Robertson. Accent: a communication oriented network operating system kernel. In *Proceedings 8th Symposium on Operating System Principles*, pages 64–75, December 1981.

[TM86] A. S. Tanenbaum and S. J. Mullender. An overview of the Amoeba Distributed Operating System. *Parallel Computers and Computation*, 1986.

[TR87] A. Tevanian and R. F. Rashid. *MACH: A Basis for Future UNIX Development.* Technical Report, Carnegie Mellon University Dept. of Computer Science, June 1987.