Design of the XERO Open Distributed Operating System*

Kazuhiko KATO Shigekazu INOHARA
Atsunobu NARITA Shigeru CHIBA Takashi MASUDA

Technical Report 92-1
Department of Information Science
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-Ku Tokyo, 113 JAPAN
E-mail: kato@is.s.u-tokyo.ac.jp

February 1992

Abstract

The XERO distributed operating system aims at providing an open distributed processing environment in which logical information structures are preserved against geographic distribution, hardware and software architectural distribution, and temporal distribution. This paper describes the design of the XERO operating system. We propose a programming model at the operating system level, which supports *multicontexts* and *multithreads* in a virtual address space. XERO supports a *complex object file system* to preserve data types in a file system and to develop distributed file systems systematically. Implementation techniques to realize these concepts are also discussed.

1 Introduction

The recent rapid evolution of hardware technology, such as in high-performance microprocessors, large primary and secondary memories, and high-speed networks have been providing distributed computing environments in which computational power can be distributed according to user requirements. In current distributed computing environments it is uncommon to find distributed systems using only single architecture computers. Rather, the environments are composed of computers using various architectures with users wishing to organize them to make the best possible use of individual components. Furthermore, since progressive innovations in both device and architectural technologies have been continuous, computing components have experienced successive changes. Therefore, it is extremely important to create an operating system made up of a heterogeneous and distributed computing environment where computers with various architectures can coexist, intercommunicate and share data.

*To appear in Journal of Information Processing, Information Processing Society of Japan.

As well as this heterogeneity in hardware architectures, heterogeneity in software architecture is also important when designing distributed systems. By "software architecture" we mean the logical structure of program development and execution environments such as programming language systems, database systems and user interface systems. At present, it is common to use various development tools and environments to improve both the productivity and maintainability of application programs. In common with the best choices in hardware for applications, the best software development and execution environments should also be selected. Hence, distributed operating systems in the near future should be able to communicate and share data among programs with different software architectures on various hardware architecture computers.

This paper describes the design of the XERO *open* distributed operating system. By "open" we mean a system conforming to the following four aspects[1]:

1. *Geographic distribution.* A facility to communicate and share data among programs located at different geographic locations. The term "distributed transparency" used in previous distributed operating systems corresponds to this. This facility will be implemented with interprogram communication.

2. *Hardware architectural distribution.* A facility to communicate and share data among programs through computers having various hardware architectures. This facility will be implemented by transforming internal data representations such as word length, byte orders, and floating point number formats.

3. *Software architectural distribution.* A facility to communicate and share data among programs developed using various program environments each of which has its own software architecture. This facility will be implemented by transforming logical data representation, e.g. transformations among Pascal record type data, `struct` type data of C language, and the S-expression of Lisp.

4. *Temporal distribution (Persistency).* A facility to communicate and share data among programs at different times. This facility will be implemented by managing data persistently, i.e., retaining data whether the system is up or not. It is preferable to treat both volatile and persistent data in a uniform way.

We term these four properties *openness*, and a computing environment or an operating system having these four properties is considered *open*.

The most crucial issue to achieve such an open operating system is how to obtain a unified framework providing the above four properties. To implement this, we designed a programming model and developed efficient implementation techniques to realize it. To share persistently typed data independent of any application program, we designed a novel file system based on the complex object concept.

[1] At present the word "open" is heard here and there, especially in commercial operating systems. However, as far as the authors know, no academic research has been done on trying to reach the heart of "openness." We believe that openness is an indispensable property of current and future distributed systems. We will attempt to reach an understanding of what the openness is to provide an open computing environment, in this and subsequent works.

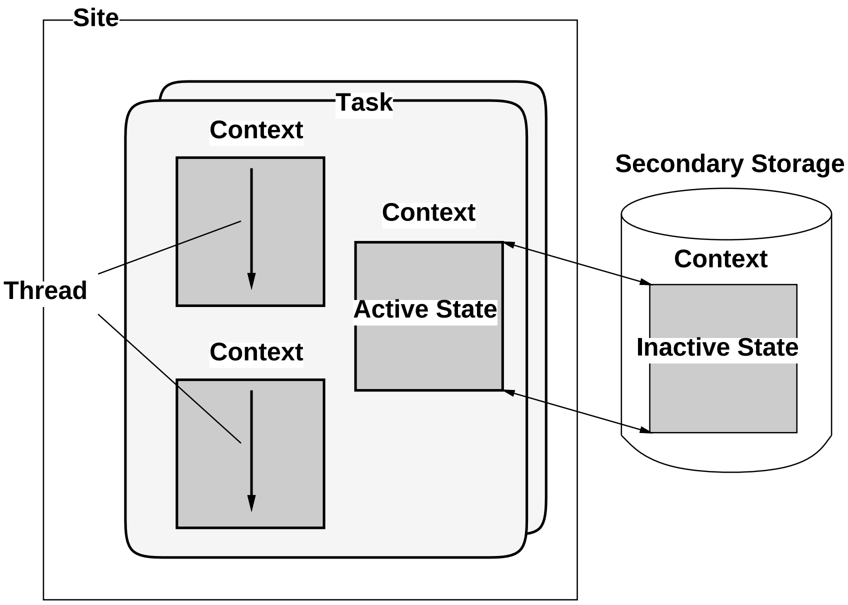

Figure 1: Basic concepts.

The rest of this paper is organized as follows. Section 2 proposes a programming model of the XERO operating system. Section 3 explains implementation methods for the fast manipulation of threads and dynamically loadable and unloadable multicontexts, which represent one of the principal concepts behind the programming model. Section 4 describes the design of a complex object file system and its efficient implementation scheme. Section 5 concludes this paper and suggests future work.

2 Programming Model of XERO

The key issue in designing an open operating system is how to preserve logical information structures, i.e. *data types* against geographic, hardware architectural, software architectural, and temporal barriers. One promising solution is to extensively utilize the concept of *abstract data types* [24] (ADT for short). The programming model for XERO, proposed in the rest of this section, is designed to constitute an infrastructure to allow ADT systems to be built providing distributed transparencies against the four barriers.

2.1 Basic Concepts

The programming model for XERO is described through the use of three basic concepts: *task*, *context*, and *thread* (see Fig. 1).

Task: A task is a virtual address space assumed to be addressed linearly. Any number of contexts and threads can exist simultaneously in any task.

Context: A context is a basic unit to manage programs and data. Basically, a context has three segments: *text*, *data*, and *stack*. A text segment is a memory object that can be directly interpreted by a CPU. Modifying this text segment is usually prohibited. A data segment is a writable memory object that can be manipulated by programs in a text segment. A stack segment is a writable memory object that stores the run time information for the programs.

Thread: A thread is the logical path of execution executing codes for the text segment in a context. Typically a thread consists of a set of registers and the program counter of a CPU. Any number of threads can exist in a task and concurrent computation occurs by running threads in contexts within a task.

These three concepts are independent and orthogonal to one another.

2.2 Types and States of Contexts

In the programming model all memory objects are treated as contexts. Contexts are classified into the following three types based on their segments:

- **Type I :** Data.
- **Type II :** Data + Text.
- **Type III :** Data + Text + Stack.

A Type I context represents data that does not include any machine-executable texts. Human-readable text files and binary data are represented by Type I contexts.

A Type II context includes a machine-executable text segment as well as a data segment. A Type II context is used to represent an ADT that encapsulates both internal data and operations, a library to be statically or dynamically linked to other contexts, and an executable program module.

When executing a program, a stack segment is needed to store run-time information. A Type III context includes a stack segment as well as data and text segments. A Type III context is used to represent and preserve the execution image of a program.

For all three types, every context is in either an *active* or *inactive* state. The active state indicates that the context is loaded onto a task and a CPU can access the context. The inactive state indicates that the context is not loaded onto a task and the context is persistent, i.e., the context exists whether the computer system is up or not. To guarantee persistence, contexts in the inactive state should be kept in persistent storage such as on magnetic disks. Treating both the active and inactive states of a context in a uniform way achieves the "temporal distribution" mentioned in Section 1.

We illustrate how the above-mentioned concepts are useful in varied information management, using language and database systems, as examples:

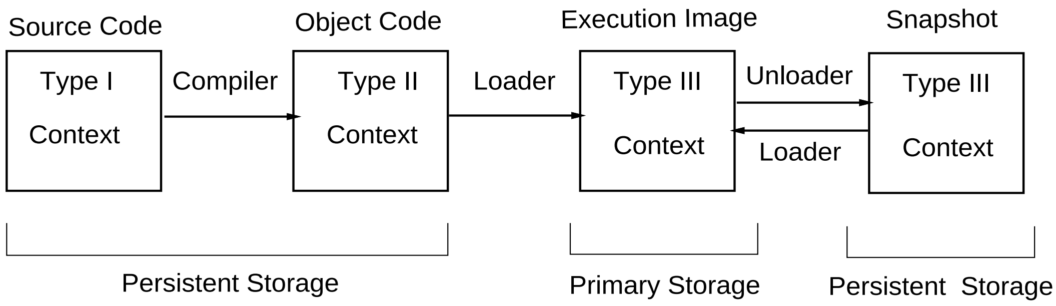

Figure 2: Language system example.

Language system example. The source program of an ordinary programming language is represented by a Type I context (see Fig. 2). A language *compiler* can be regarded as a transformer that changes an inactive Type I context (source codes) into an inactive Type II context (executable codes). A *linker* generates an inactive Type II context from several inactive Type II contexts, resolving the symbol references between them. A *loader* adds a stack segment to an inactive Type II context and creates an active Type III context on a task. If a user wants to save the existing execution image of the Type III context after execution starts, an *unloader* creates an inactive Type III context from it. The saved Type III context can be loaded onto a task, which need not necessarily be identical to the task from that the unloaded context was established.

Object-oriented database system example. In object-oriented database systems [5, 25], which are currently attracting much attention from database researchers and developers, a database system manages methods (procedures) as well as persistent data, in the form of "persistent objects." With the programming model, such a persistent object can be represented as a Type II context. By using the extended RPC mechanism described in Section 2.3, sending messages to a persistent object is naturally realized. The activation of a Type II context corresponds to beginning the transaction with the object. Inactivation with write-back corresponds to committing the transaction and inactivation without write-back corresponds to aborting the transaction. Using the multilanguage RPC mechanism described in [21] , it is possible to transparently send a message to any object even if the user program languages and the method implementation language differ.

2.3 Extended Remote Procedure Calls

In the programming model, intercontext communication is achieved through the use of extended remote procedure calls (RPC). Extension involves dealing with the temporal distribution; the RPC mechanism is unique[2] in not only providing message transfer between different address spaces on different sites but also causing state transition of contexts.

[2]Scheme, a dialect of Lisp, supports a primitive called `call/cc` (`call-with-current-continuation`) [30], which integrates a function call mechanism and a program context manipulation.

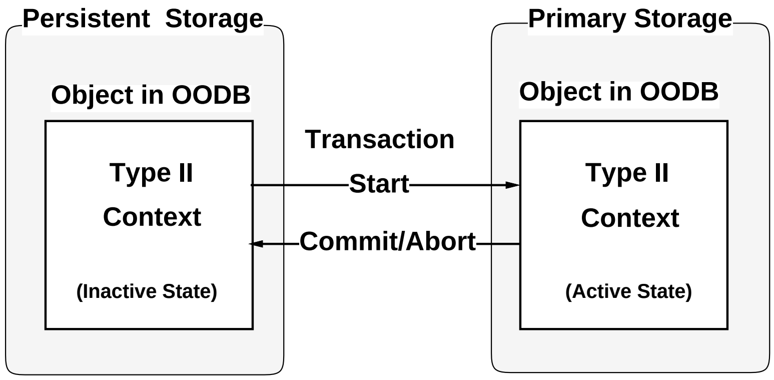

Figure 3: Object-oriented database system example.

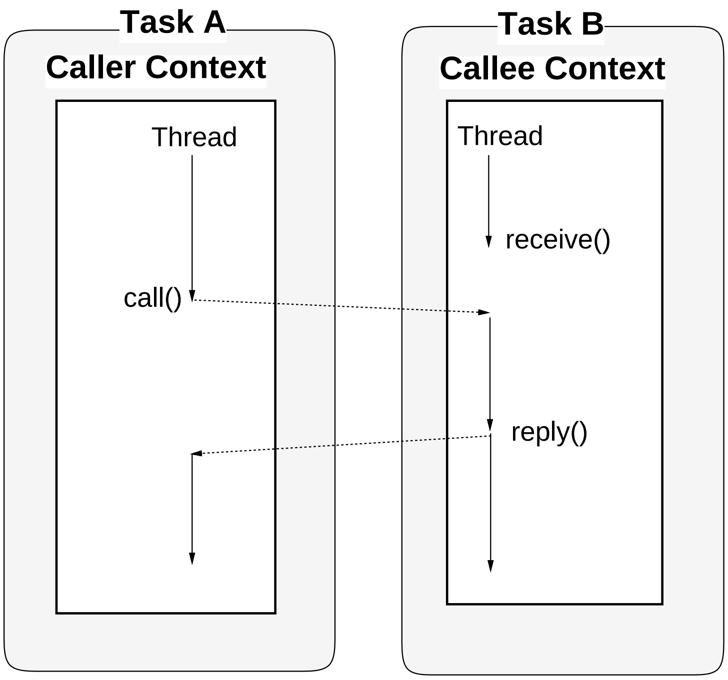

Figure 4: Remote procedure calls between contexts.

Table 1: RPC primitives and state transition of contexts.

Callee→ Caller↓	`receive()`	`receivecc()`
`call()`	State transition does not occur	Callee is loaded to the caller task
`callcc()`	Caller is loaded to the callee task	Impossible

Two active contexts for Type II or Type III communicate with each other in the style of RPC using the primitives `call()`, `receive()`, and `reply()` as shown in Fig. 4. A caller context calls a callee by issuing the `call()` primitive specifying several call arguments. The callee context receives the call by issuing the `receive()` primitive. The `call()` and `receive()` are synchronized, and the execution of either the caller or the callee issuing the primitive beforehand is blocked until synchronization is established. Just after synchronization is established, the arguments specified by the caller are passed to the callee, which continues execution. When the callee issues the `reply()` primitive specifying several reply arguments, the block for the caller's execution is lifted and the caller receives the reply arguments.

When a caller context and a callee context are located on different sites with different CPU architectures, data representation must be converted to preserve logical data type information. We have developed a communication technique to preserve all statically determined data types including the function type on heterogeneous CPU architectures using multilanguages [28, 21]. With this technique, procedures accessing local data are automatically transformed into procedures that access remote data automatically. Hence, this technique makes data transfer from and/or to remote procedures transparent to programmers.

The state of a context changes if the context issues either `callcc()` (call-with-current-context) or `receivecc()` (receive-with-current-context) instead of `call()` or `receive()`, respectively. If an active caller context issues `callcc()`, then the caller context changes its state to the inactive state: the context is unloaded from the task and is placed in a persistent storage. Similarly, if an active callee context issues `receivecc()`, then the callee context changes to the inactive state.

The `call()` and the `callcc()` primitives synchronize with the `receive()` and the `receivecc()` primitives as shown in Table 1. The rules for synchronization and state transition are as follows:

- When the inactive caller that issued `callcc()` synchronizes with the callee that issued `receive()`, the caller in persistent storage is loaded to the task (*callee task*) in which the callee exists as shown by Fig. 5 and 6.
- When the inactive callee that issued `receivecc()` synchronizes with the caller that issued `call()`, the callee in persistent storage is loaded to the task (*caller task*) in which the caller exists as shown by Fig. 7 and 8.
- Synchronization between the inactive caller that issued `callcc()` and the inactive callee that issued `receivecc()` is impossible since an address space to which the two contexts need to

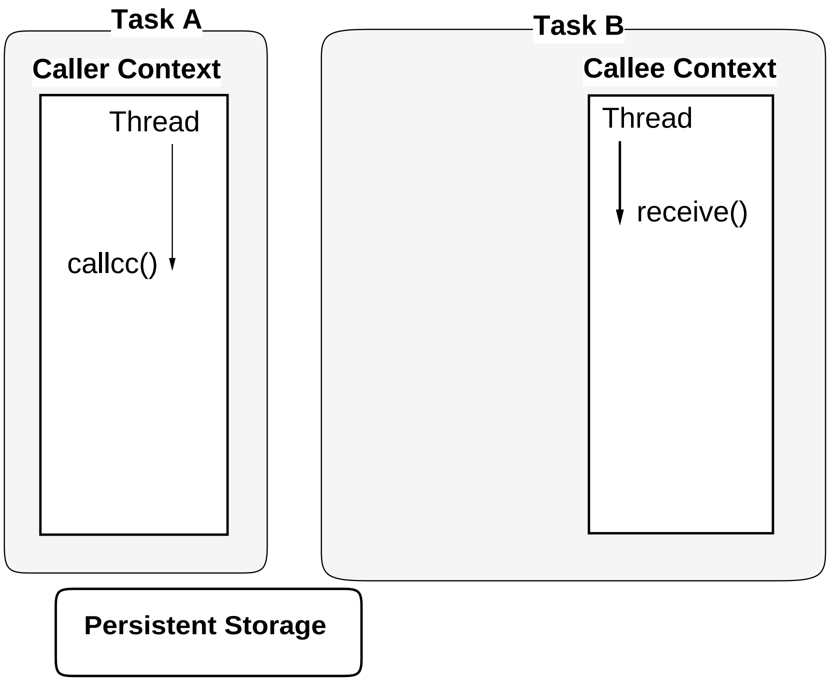

Figure 5: Context migration between tasks with `callcc()` (before synchronization).

be loaded has not yet been determined.

After synchronization is established with `callcc()` or `receivecc()` the semantics for these two primitives are the same as `call()` or `receive()`, respectively.

2.4 Summary

The advantages of the described programming model are summarized as follows:

- Method invocation to persistent objects (or message transfer to persistent objects) can be treated in the same way as that to volatile objects in primary storage. This eases the implementation of persistent ADT systems or object-oriented database systems. From another viewpoint, we can think that the programming model can be considered a starting point to achieve a so-called *single-level store* system in heterogeneous distributed computing environments.

- The state transition mechanism of contexts in the model can be used to attain *context migration* between tasks. Generally, it is not easy to implement *process migration* transparently to the migrated process itself or other processes connected to the process, due to both technical and performance reasons. In the model, a context migrates between processes under the control of the migrated context itself. Although context migration is a fairly restricted way to process migration, we think that it provides a realistic solution.

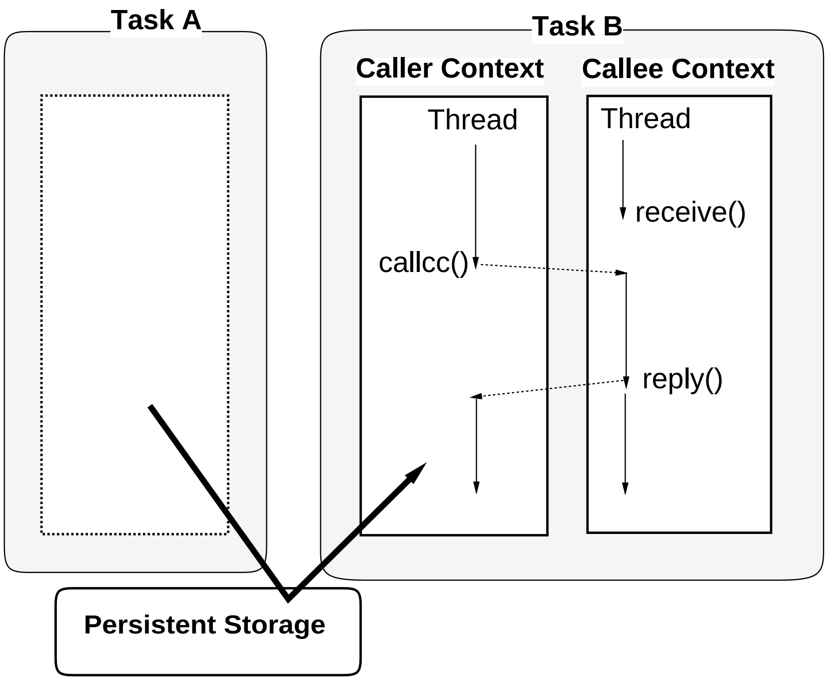

Figure 6: Context migration between tasks with `callcc()` (after synchronization).

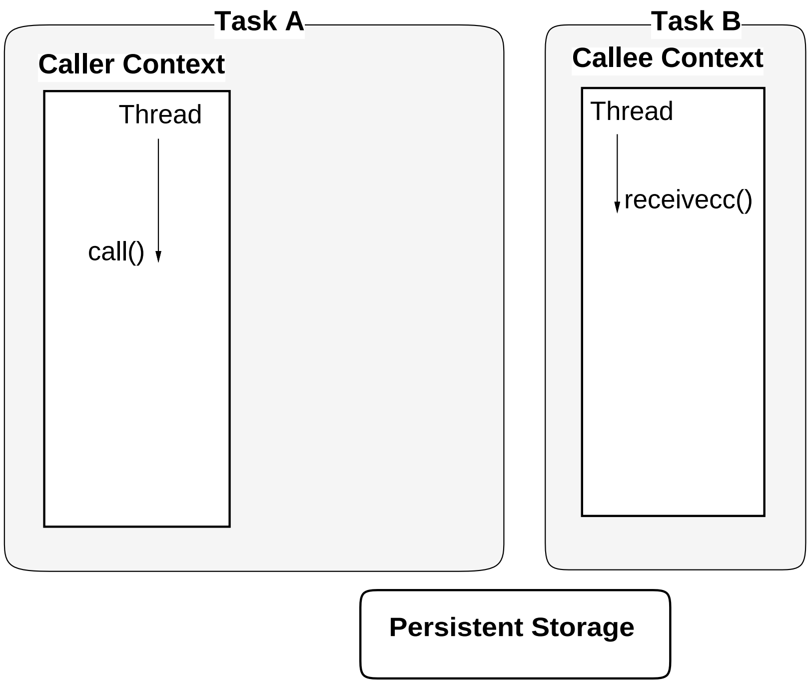

Figure 7: Context migration between tasks with `receivecc()` (before synchronization).

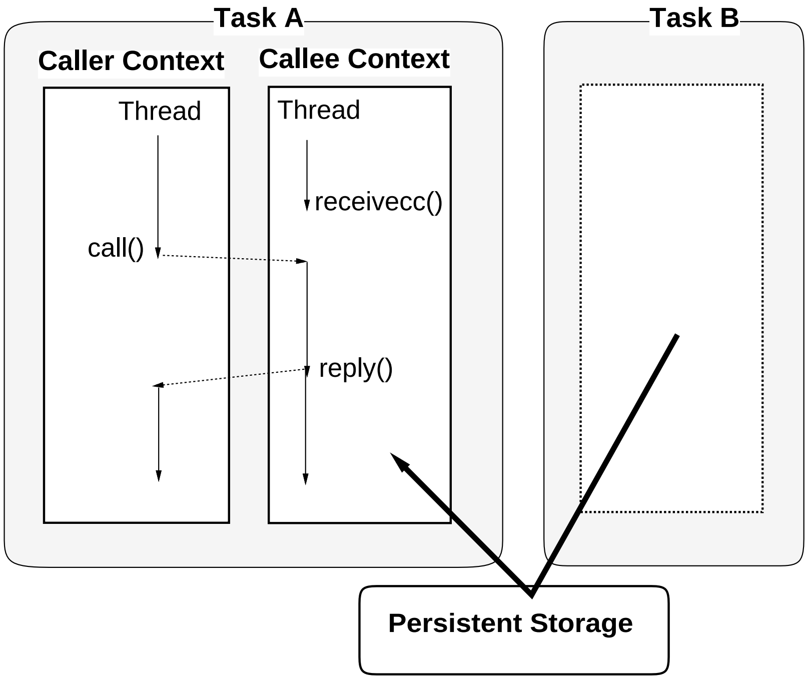

Figure 8: Context migration between tasks with `receivecc()` (after synchronization).

- RPC has semantics very close to the procedure calls of programming languages. Using the fact that almost all programming languages have procedure calls or similar mechanisms to connect program modules, we can implement multilanguage communication in a distributed environment [28]. We have already developed a distributed C language, in which ordinary procedure calls are interpreted as remote procedure calls [26, 22].

3 Internal Structure of the XERO Operating System

Most of the recent distributed operating systems have adopted the "kernelized kernel" principle, meaning the kernel of an operating system should be minimized and most operating system facilities should be implemented as ordinary user programs. This principle is very attractive since distributed operating systems are required provide many functions that must evolve and change according to user requirements. The XERO operating system, too, is designed along the lines of this principle.

Unlike other recent distributed operating systems, XERO has an outstanding feature in its structure; every task has a program module in its user address space, which supervises the execution of all programs in the task. This program module is called a *task supervisor* (see Fig. 9). When a task is created, the task supervisor is invoked for the task and the control of the (virtualized) CPU is passed to it first. The task supervisor manages threads and contexts in the task, cooperating with the kernel. The rest of this section describes how the programming model has been realized in our current implementation of XERO.

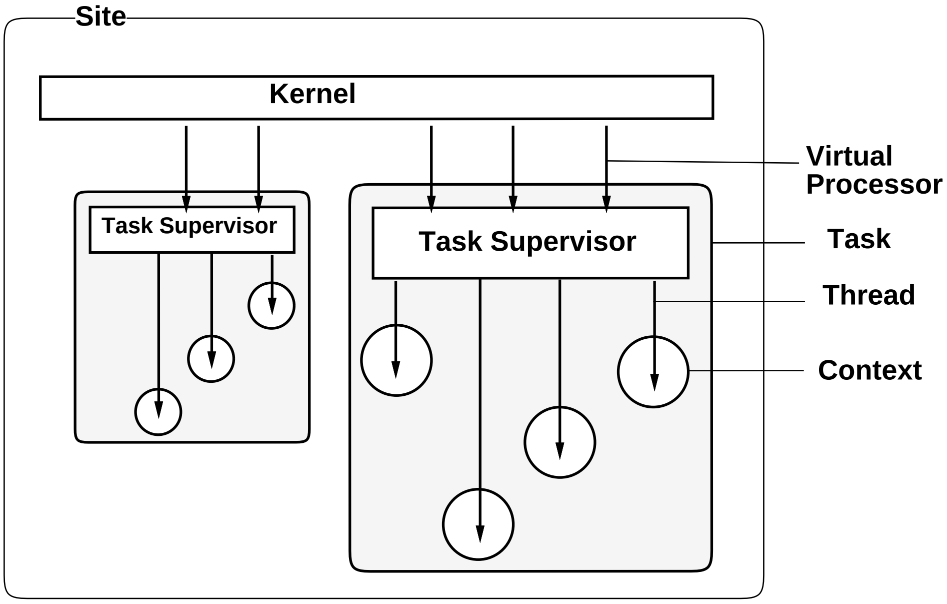

Figure 9: Internal structure of multithreads.

3.1 Attainment of Multithreads

In addition to the XERO operating system, many other recent operating systems also support multithread mechanisms. Previously proposed multithread mechanisms are classified into two: the *user-level supports* and *kernel-level* for threads. Examples of the former are the SunOS Lightweight Process Library [35] and the Xerox PCR (Portable Common Runtime) [40]. Examples of the latter are the V system [7], Mach [37] and Clouds [12].

The features of the two types of thread mechanisms are summarized as follows:

- In the user-level support of threads, the operating system kernel is not aware of the existence of multithreads in a virtual address space, and the kernel provides only one unit of CPU execution to an address space. Therefore, when one thread in an address space issues a system call to the kernel, other executable threads in the address space cannot run until the service to the system call ends. We call this the *thread blocking problem*. Because of this, the multithread mechanism of the Sun LWP libraries has a defect in that concurrency between multithreads in a address space is prevented. On the other hand, thread switching in an address space is very fast since it does not need the help of the kernel.
- In the kernel-level support of threads, on the other hand, the kernel directly controls and schedules threads. Hence, in this system, even when a thread issues a system call and waits for its completion, the execution of any other thread in the address space is not blocked. Direct scheduling of threads by the kernel, however, causes considerable overhead to switch mapping between the CPUs and threads in the task, since switching can be performed only by the kernel and requires hundreds of machine instructions, which is comparable to system

call processing.

The mechanism for thread management in the XERO operating system combines the virtues of the above two types of thread mechanisms while eliminating their drawbacks. The approach taken by XERO is to control the execution of multithreads by using the cooperation between the kernel-level program module and the user-level program module named the *task supervisor* (see Fig. 9). The multithread mechanism of XERO is based on "double-multiplexing" CPU by a kernel and task supervisors. A kernel provides one or more *virtual processors* for a task by multiplexing a physical CPU. Furthermore, a task supervisor provides threads by multiplexing the virtual processors provided by the kernel. A virtual processor is a unit of CPU scheduling from the kernel's point of view, whereas a thread is that from the user's or the programmer's point of view. These two-level abstractions of a physical CPU, i.e., virtual processors and threads, are completely transparent to users. Users only recognize threads as virtualizations of CPUs.

To establish a two-level abstraction of a physical CPU, i.e., virtual processors and threads, completely transparent to users, the cooperating protocols between kernels and task supervisors were carefully designed. The protocols allow that a CPU control right is returned to a kernel when the virtual processor cannot proceed its execution—e.g. when an exception including page fault occurs, when a system call is issued, or when the virtual processor exhausts its execution time quantum (i.e. *preemption*). After the kernel recognizes the return of control, the kernel passes the CPU-context (namely the contents of all CPU registers including the program counter and processor status word), saved by the interrupt handler or the exception handler in the kernel, to the relevant task supervisor. The task supervisor saves the passed CPU-context to enable thread scheduling.

To avoid the thread blocking problem, the task supervisor informs the kernel, using task supervisor–kernel protocols, that another new virtual processor should be created for the task when blocking occurs. This means that several virutal processors are created and allocated to the task automatically. The created virtual processors are eliminated when the task supervisor requests the kernel to do this or when the task itself is destroyed. Further details on the cooperation between task supervisors and the kernel in XERO are described in Reference [16].

The cooperative approach of managing multithreads just described has the following three advantages. First, fast thread operation can be attained, such as creation, switching, and destruction, since operations are performed only in the user address space and do not require kernel service unless operations to virtual processor are required. The performance of multithread operation will be discussed in Section 3.3. Second, the cooperative approach avoids the thread blocking problem due to the carefully designed protocols between the task supervisor and the kernel. Third, since a task supervisor is only a user-level program module, it is possible to create a task supervisor customized to a knowledge of each application.

3.2 Attainment of Multicontexts

As described in Section 2.2, multiple contexts, which may include text segments (i.e. compiled binary codes), can be dynamically loaded to a task and unloaded from a task. Such dynamic loading and unloading facilities for contexts require the relocation of contexts so that loaded contexts will function for any task. While some previous operating systems [10, 15] provide program relocation using specific CPU architectures with a "segmentation" mechanism, XERO provides a *machine-independent* relocation mechanism. This is because XERO aims at providing the hardware architectural distribution described in Section 1.

Since Type I contexts cannot contain any program codes that should be managed by the operating system, we do not have to consider relocating Type I contexts. As Type II contexts can be regarded as a subset of Type III contexts, we will describe a way to relocate Type III contexts in the following.

Two crucial aspects in attaining machine-independent relocation are the following:

- Making text segments relocatable.
- Making data and stack segments relocatable.

The first aspect is achieved by modifying existing compilers to generate relocatable codes. The second is achieved by maintaining all address space-specific information during the entire lifetime of each context. In the following we discuss these two issues in more detail.

3.2.1 Generating relocatable object codes

To minimize memory usage, text segments should be able to be shared by all contexts loaded from the same object codes to a task[3]. The requirements, relocation and sharing of text segments using only standard hardware, are satisfied by employing the CPU *register-relative* addressing mode to access all named memory cells (i.e., symbols in programming languages).

Let us examine our modifications to the GNU C compiler [33] on a SONY NEWS workstation, which has a M68030 CPU. Our modifications are twofold:

1. The `a5` register of the M68030 CPU is used as a "base" register, in which the starting address of a loaded context is set.
2. All references to the named memory cells are performed using the register-relative addressing mode of the CPU.

[3] In our current implementation the text segment of a context loaded from the same object codes cannot be shared between distinct tasks. Without using hardware mechanisms such as segmentation registers, it is very difficult to share relocatable text segments between tasks.

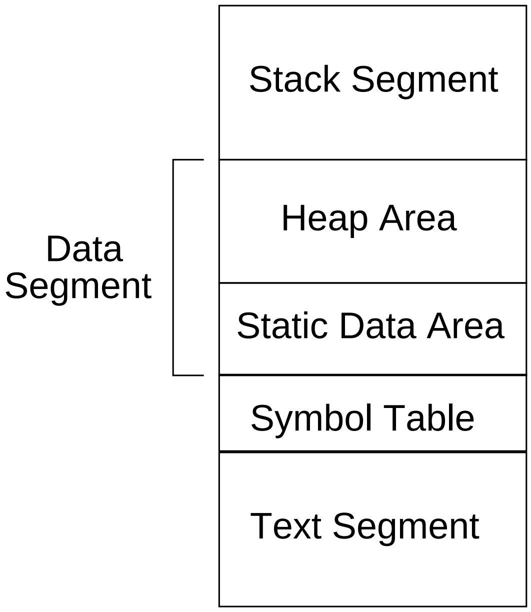

Figure 10: Internal structure of a context.

In the following, the left-hand generated codes of the original compiler will change to the right-hand codes in the modified compiler.

Before modification		After modification
`move.l d1,_var`	$\Longrightarrow$	`move.l d1,(_var,a5)`
`jsr _printf`		`jsr (_printf,a5)`

3.2.2 Maintaining address space-specific information in data and stack segments

The contents of data and stack segments are classified into two types: *basic data types* and *pointer types*. Basic data types, e.g., integers, float point numbers, characters, etc., are types that can be directly interpreted and manipulated by CPUs. The pointer type are themselves addresses in an address space. While the basic type is generally independent of any address space, the pointer type is dependent on the address space on which the segments exist. To relocate data and stack segments, only the pointer type needs to be dealt with.

The way we relocated data and stack segments without the use of any specific hardware is:

1. Each context of Type II and Type III retains memory-typing information called *symbol tables*, which are initially produced by a compiler. (In the case of the GNU C compiler, such memory-typing information is produced by specifying the "-g" option in the compiler invocation command.)

2. When a task supervisor loads a context to the task, it adds the difference between the starting

address of the currently existing task and the previously existing task, to every pointer type occurrence.

We will now describe how data and stack segments are relocated.

Data segments. A data segment is composed of two areas: *static data area* and *heap area* (see Fig. 10). The static data area is statically typed when compilation occurs and memory allocation is initially determined to be invariable. In contrast, the heap area is allocated and typed at the time of execution.

All typing information for the static data area is included in the symbol table generated by the compilers. When a task supervisor loads a context to the task, the difference is added between the starting address of the currently existing task and the previously existing task, to every pointer type occurrence in the static data area. In our current implementation on the SONY NEWS workstations, this calculation is denoted as:

```
[New Pointer Value] = [Old Pointer Value] + ( new_a5 − old_a5 )
```

Typing information for the heap area is not determined in advance and may change dynamically. As a result programmers must explicitly provide typing information for the heap area using a library routine which for convenience has been named `typing`.

```
typing(ptr, type, elements)
char *ptr;      /* pointer to the variable */
char *type;     /* type name (of an element) */
int elements;   /* number of elements */
```

`typing(ptr, type, elements)` specifies that the heap area starting from the `ptr` address contains `type` related data, and the data number is designated by `elements`. An example of this usage written in C language is:

```
         :
ptr = malloc(sizeof(struct S)*100);  /* (1) */
typing(ptr, "struct S", 100);        /* (2) */
         :
free(ptr);                           /* (3) */
untyping(ptr, "struct S", 100);      /* (4) */
         :
```

The typing information provided by the system function `typing()` is added to the symbol table. `untyping()` removes the typing information from the table. The symbol table of a context includes type declarations of the source programming languages. Line (1) in the above example specifies that one hundred times the number of bytes in the C structure `S` are allocated in the heap segment of the context. Line (2) specifies that the allocated heap memory starting from address `ptr` should be typed with the following one hundred C structures, each having type `struct S`. After executing `typing()`, the calculation for relocation is the same as static memory areas in data segments. When the allocated heap memory is useless, type information must be explicitly removed. Lines (3) and (4) free the allocated heap memory and remove the additional type information from the symbol table.

Stack segments. A stack segment is a dynamically allocated memory similar to the heap area. Unlike with the heap area, both typing information and size of the stack segment can be determined automatically. The size of the stack segment is obtained by checking the stack pointer register. The typing information of a stack segment is obtained by analyzing "stack frames" (see Fig. 11). Since all variables allocated on a stack segment are limited to the local variables of functions or procedures and all typing information on local variables for every function or procedure is stored in a symbol table, the typing information of a stack segment can be obtained by following the frame pointers and checking the return addresses pointing to local variable definitions in the symbol table.

The typing information for the i-th stack frame is obtained by examining the "return address" of the $(i+1)$-th stack frame $(1 \leq i \leq n)$ in Fig. 11. How can the corresponding function to the top frame (`fun_n()` in Fig. 11) in a stack segment be identified? In the XERO programming model a context is unloaded from a task only when it issues the `callcc()` or `receivecc()` primitives described in Section 2.3. A function corresponding to the top frame can be identified by constituting the two primitives to initially call a dummy `suspend_context()`.

After typing information on the stack segment is obtained, the calculation for relocation is the same as for static memory areas in data segments.

3.3 Achieved Performance

The mechanism for the multithreads and multicontexts have been implemented on SONY NEWS workstations with MC68030 CPUs and MC68881 FPUs. The kernel and the task supervisor of XERO are written in both C language and assembly language. In the current version of the XERO kernel, a portion of the 4.3 BSD UNIX kernel codes were used to build the XERO kernel. Approximately 4,300 BSD code lines were modified and approximately 4,200 C code lines were created from scratch along with 3,200 assembly code lines. The thread management portion of the task supervisor has about 2,100 lines of C codes and 900 lines of assembly codes. Additionally about 5,600 C code lines in the task supervisor were created to manage contexts. About 200 lines of the GNU C compiler written in C language were modified to generate relocatable codes. Using this implementation we measured the actual performance of the multithread and the multicontext mechanism.

.bb .bb

Figure 11: Stack segment; `fun1_()` calls ..., `fun_n-1()` calls `fun_n()`, and `fun_n()` calls `suspend_context()`.

Table 2: Performance of multithread operations (in machine instructions for MC68030; N/A: not supported as a system call; –: no meaning).

Operation	Thread	Virtual Processor	Task	Total	4.3 BSD Process
Creation	92	669	11,983	12,744	18,977
Switching	40	205	N/A	–	N/A
Destruction	90	546	7,989	8,625	4,324
Suspension	13	187	N/A	–	N/A
Resumption	46	226	N/A	–	N/A

3.3.1 Multithreads

The cost of primitive operation for threads, virtual processors, and tasks was measured at implementation. Table 2 illustrates the expenses measured for the number of machine instructions executed during each operation. Measurement was done using the trace mode of the MC68030 CPU. All codes for operations to manipulate threads were executed in the user mode (i.e. in the user address space) of CPUs, and all codes for operations to manipulate virtual processors and tasks were executed in the kernel mode (i.e. in kernel address space). For purposes of comparison, the cost of primitive operations on 4.3 BSD UNIX processes on the SONY NEWS were measured in the same way. In the table, N/A (not available) appears when the operation does not have a system call interface.

As clearly shown by Table 2, the costs for all thread operations are very low. The costs for the operations of the virtual processor are 5 times (when switching or resumption)–14 times (in suspension) those of thread operations. This is because the structure of the data managed for a virtual processor is more complex than the structure for a thread. The creation and destruction of a task are 18 times and 15 times those in a virtual processor, respectively. This is because the creation and destruction of a task requires the expensive manipulation of virtual address space.

The total costs for creation and destruction operations are 12,744 and 8,625, respectively for a thread, a virtual processor and a task. The former cost is 6,233 instructions less than the cost of creation using 4.3 BSD process. This decrease is due to the system call for the 4.3 BSD process creation, the *fork* call, which includes the copying of contents from the parent's process to the child's. The cost for destruction is 4,301 instructions more than the cost of destruction using the 4.3 BSD process. This increase is because any multiple CPU activity in a kernel must settle in XERO.

3.3.2 Multicontexts

To examine the performance of the multicontext mechanism, we repeated the loading and unloading of a Type III context to and from a task one hundred times, varying the size of the context (no

Table 3: Performance for loading and unloading contexts (in milliseconds).

Program	Context Size (KBytes)	Size of Symbol Table (KBytes)	No. of Pointers	User Time	System Time	Response Time
echo.c	18	1.0	45	11.3	25.3	100.3
grep.c	36	6.6	19	23.1	40.4	170.8
mv.c	56	17.2	3	34.3	54.7	254.5
write.c	107	34.2	8	78.3	92.6	376.6
ls.c	311	109.1	990	665.0	220.0	1,215.0

effective work was done by the contexts). The source codes for the examined contexts were extracted from 4.3 BSD UNIX utility programs written in C language. Table 3 shows the average costs in terms of time. In the table, user time is the required time for relocation, and system time is the cost of executing disk file I/O. The response time in the table is the whole cost in terms of time including user time, system time, and physical disk operation time to load and unload a context. All overheads to relocate a context are included in user time since all relocating operations are executed in the user mode. Except for the program "ls.c," user time is 10–20 % of the response time. With "ls.c," user time is 55 % of the response time. This is because of the large size of the symbol table, the large number of pointers in the context and because current implementation does not use an efficient table management technique. Performance with a large symbol table and a large number of pointers can be improved by using a more efficient table management technique such as hashing or a binary searching table. Since we expect that loading and unloading are generally not performed as frequently, we are able to state that the relocation mechanism can withstand practical use.

4 Complex Object File System

Most recently designed operating systems have supported file systems managing secondary storage as a byte-stream file set that is logically located in a hierarchical name space [31]. By "byte-stream file" we mean a file that is just a sequence of bytes with any amount of data in the file system being accessed by specifying the byte offset from the top of the file (or from the current offset pointer). We can see such a file system as an *untyped* persistent heap memory. To achieve an open distributed computing environment, however, persistent data types must be handled by operating systems which share and interchange this data between programs on heterogeneous CPU architectures using multiple programming languages. The approach adopted here was to build a file system based on the concept of *complex objects*, whose importance and usefulness have recently been well established in the database research community [6, 2] and whose mathematical properties are currently being investigated [1, 27]. A complex object is composed of basic data type objects with two simple and adequate data constructors: *tuple* and *set* constructors. Each object has its own *object identifier* (abbreviated OID) and composition is attained by embedding the OIDs of

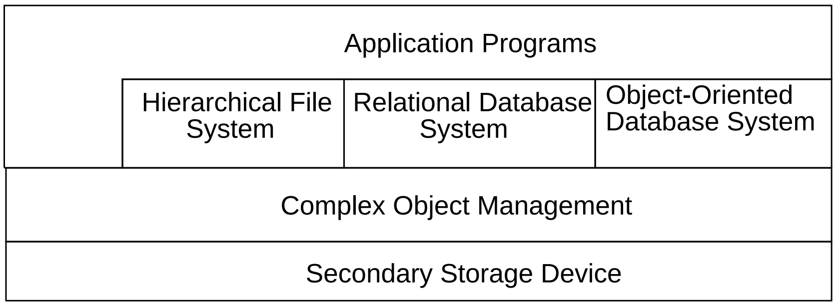

Figure 12: Secondary storage management in the XERO operating system.

other objects within the referencing object. More precise definition on complex objects will be given in Section 4.1.

In the XERO operating system, instead of building a complex object management system on a hierarchical file system, our complex object management system is built on the physical secondary storage management layer as shown in Fig. 12. As will be described in subsequent sections, a UNIX-like byte steam file is represented as an instance of the basic data type implemented by software, and a file system with a hierarchical name space is implemented as an application of the complex object system (see Fig. 12). We named this new configuration of secondary storage management a *complex object file system.* The complex object file system has the following two advantages that none of the previous file systems have:

- Persistent data types are managed independently of any application programs. This feature represents the bottom line in sharing persistent data between multiple software architectures using multiple hardware architectures.
- The configuration of a complex object file system eliminates redundancies [34, 18] between the file system layer of an operating system and the physical data management layer of a database management system (DBMS) in the configuration for the previous systems (see Fig. 13). An instance of these redundancies can be found between the disk page management using a tree structure in an operating system (e.g. UNIX i-node structure [31]) and management using an index structure in a DBMS (e.g. B-tree structure [9]). Another instance of redundancies can be found between the buffer managements of an operating system and a DBMS. With our complex object file system, such tree structures and buffer managements are unified, since DBMS implementors can utilize the complex object facility provided immediately on the physical storage management layer. With the elimination of redundancies we can expect improvements in the performance of the DBMS, compared to the previous configuration, and can expect to reduce the cost of implementing DBMSs.

In the rest of this section we present the definition, the use, and the internal structures of complex objects in XERO.

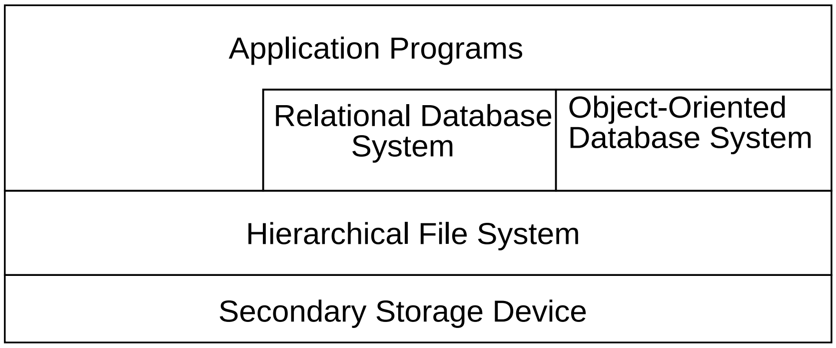

Figure 13: Secondary storage management in conventional operating systems.

$$\text{Types of complex objects} \begin{cases} \text{Basic data types} \begin{cases} \text{Machine-primitive types} \\ \text{Abstract data types} \end{cases} \\ \text{Composite data types} \begin{cases} \text{Tuple} \\ \text{Set} \end{cases} \end{cases}$$

Figure 14: Type structure of complex objects.

4.1 Description of Complex Objects

A *complex object* is typed persistent data that may recurrently be composed of several other objects. Formally, a complex object O is represented by a triple: *(OID, type, value)*.

An *OID* is a unique identifier independent of location given for each complex object. An OID is generated by a system when a complex object is created and it continually identifies the object until the object is removed. Given an OID, the system is responsible for finding the location of corresponding object and providing its own value to users.

A *type* is a definition of structure, and a *value* is an instance of that definition. A type is either *basic, tuple,* or *set* (see Fig. 14), and a value depends on its type:

1. Basic types are composed of two: *machine-primitive types* and *ADTs* (abstract data types) (Fig. 14). The former types are directly interpreted by CPU architectures, while the latter types are supported by software [24, 39]. The set of values of the type is only defined indirectly as it consists of all values that can be generated by the software module. Examples of machine-primitive types are integers, floating point numbers, and characters, among others. An example of an ADT is the "byte stream" type as in a UNIX file system.

2. A tuple type represents an aggregate of objects. If $T_1, \ldots, T_n$ are types, then $[T_1, \ldots, T_n]$ is a tuple type. If $O_1, \ldots, O_n$ are objects of types $T_1, \ldots, T_n$, respectively, then $[O_1, \ldots, O_n]$ is a value of this type.

3. A set type represnts a collection of objects for a type. If T is a type, then $\{T\}$ is a set type. Any set of objects for type T is a value for this type. The number of objects in a set is variable.

4.2 Use of Complex Objects

The concept behind complex objects is quite expressive and many persistent data structures can be defined using it. Here we illustrate how complex objects can represent various data in both operating systems and database systems. Although the usefulness of complex objects has been recognized in the database community, the indication of usefulness for an operating system proposed here is the first as far as the authors know.

4.2.1 Operating System Use

By using complex objects, several methods of information management, each of which has been implemented ad hoc in previous operating systems, can be attained in a unified way. We demonstrate usefulness by illustrating uses in managing name tables and hierarchical file systems.

Name Table. To operate a distributed system, the operating system must manage various name tables, for example, the mapping tables linking user names with their information (such as `/etc/passwd` in UNIX), tables linking symbol names with network addresses (such as network name servers [13]), and mapping tables linking symbol names with the identifier of persistent data (such as a UNIX *i-node* table), etc. In XERO, such a name table can be attained using a set of tuples, each having mapping information. For efficient retrieval to the table, users can add the index tree (B^+-tree). A name space can be in a hierarchical or even a network structure by decomposing the name table into a set of tables (table itself is a set of tuples) and referencing them with the OIDs of the tables.

Hierarchical File System. The following description assumes knowledge on the internal structure of the UNIX file system [3]. A UNIX-like hierarchical file system can be built as an application of complex objects. To do this, prepare an ADT implementing the data type of the UNIX-like byte stream file (i.e. UNIX ordinary file). Represent the *i-node* table by a set of tuples, each of which corresponds to the i-node information of a file, with each OID corresponding to the *i-number* in UNIX. A directory file can be represented by a set of binary tuples, each of which has a symbol name (i.e. file name), and the OID of either an ordinary file object or a directory file object. By using a network-wide OID, the file system described here can be extended naturally to a distributed file system [23].

4.2.2 Database Use

Relation. The first normal form relationships in the relational data model can be represented as a set of tuples, e.g., $\{[A_1 : O_1, \ldots, A_n : O_n]\}$.

Nested Relation. The nested relation, discussed in [29, 32] for example, can be represented as a set of tuples whose attributes are other sets or other tuples, e.g.,$\{[A_1 : \{\ldots\}, \ldots, A_n : [\ldots]]\}$.

Shared Objects. Furthermore, the concept behind complex objects can explicitly represent objects which share with several distinct objects. Any object can have multiple parents referencing it via its OID; thus, the objects being referenced are termed *referenced* objects, and objects that are themselves referencing are termed *referencing* objects. The referencing of an object is achieved by including the OID of the referenced object in the value for the referencing object.

4.3 Internal Structure

We will now describe how complex objects are internally managed in XERO. Basically, management differs according to type. We present this in the order of basic, tuple, and set type.

4.3.1 Basic Type Objects

As mentioned in Section 4.1 basic type objects are composed of machine-primitive type objects and ADT objects. With machine-primitive type objects, since the internal representation of values for these types are different for each CPU architecture in general, representation conversion is required when objects for these types are moved (or copied) to other architectural sites. In converting internal representation, XDR [11] was used for the current implementation.

ADT objects are implemented using the Type II contexts described in Section 2.2. A Type II context has inside a data segment and an text segment. When a Type II context is used to represent an ADT object, a data segment should contain the internal *state* and the text segment should contain the *methods* manipulating the internal states. These internal states and methods are encapsulated, and the user of an ADT object can only call published methods for the object. The calls to an ADT object are performed by the RPC mechanism which is extended to the persistent space, described in Section 2.3.

The management of physical storage for basic type objects is the same as the management for single attribute tuple type objects, as described next.

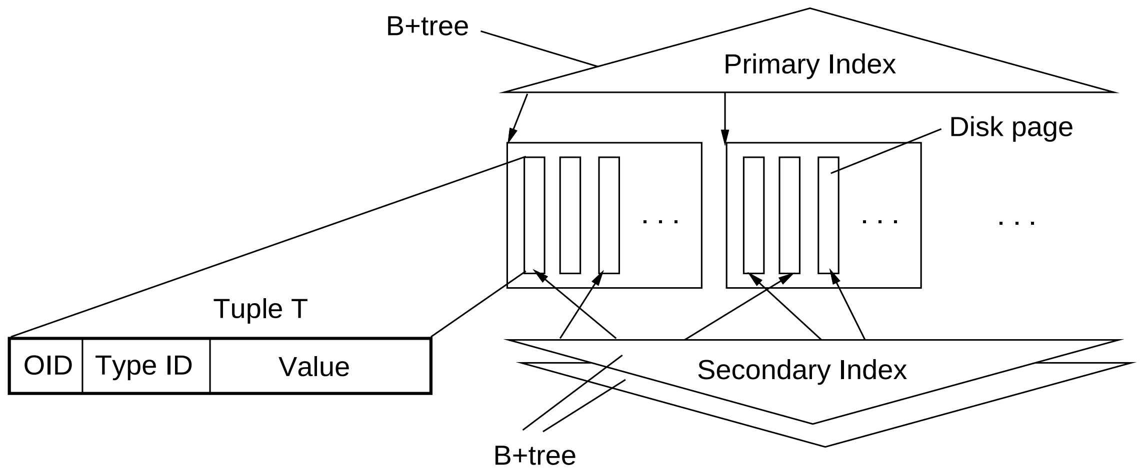

Figure 15: Internal structure of tuple-type objects.

4.3.2 Tuple Type Objects

Tuple and set-type objects are basically managed with a B^+-tree [9]. Figure 15 shows the internal structure of tuple-type objects. All instances of the same tuple type are stored in the leaf pages of a primary index B^+-tree. The set of leaf pages in a primary index can have several secondary index B^+-trees for any of the attributes to accelerate content-addressing. Refer to the content of tuple T in Fig. 15 consisting of OID, Type ID, and Value. The OID of T is just the object identifier for T. The type ID of T is in reality the OID of a tuple object specifying the type of T (note that the specification of a type is kept in a tuple type object). The value of T is determined according to the type description of T. When a user has only the OID of T, the system guarantees the user access to Type ID and Value by retrieving through the primary index in the figure. When a user has a value (perhaps partial) the user can access tuples having values using proper secondary indices if the indices are prepared in advance; if there is no proper index, the tuples for the type must be searched tuple by tuple.

4.3.3 Set-Type Objects

The management of set-type objects is similar to that of tuple-type objects, except that each leaf page of a primary index for a set object contains the OIDs of member objects of the set. Figure 16 illustrates the internal structure of a set-type object. For each set type definition, a primary index for the set type is maintained and its leaf pages include OIDs of the instances of the set type. For each set instance, a parimary index is maintained to keep the member elements of the set in its leaves.

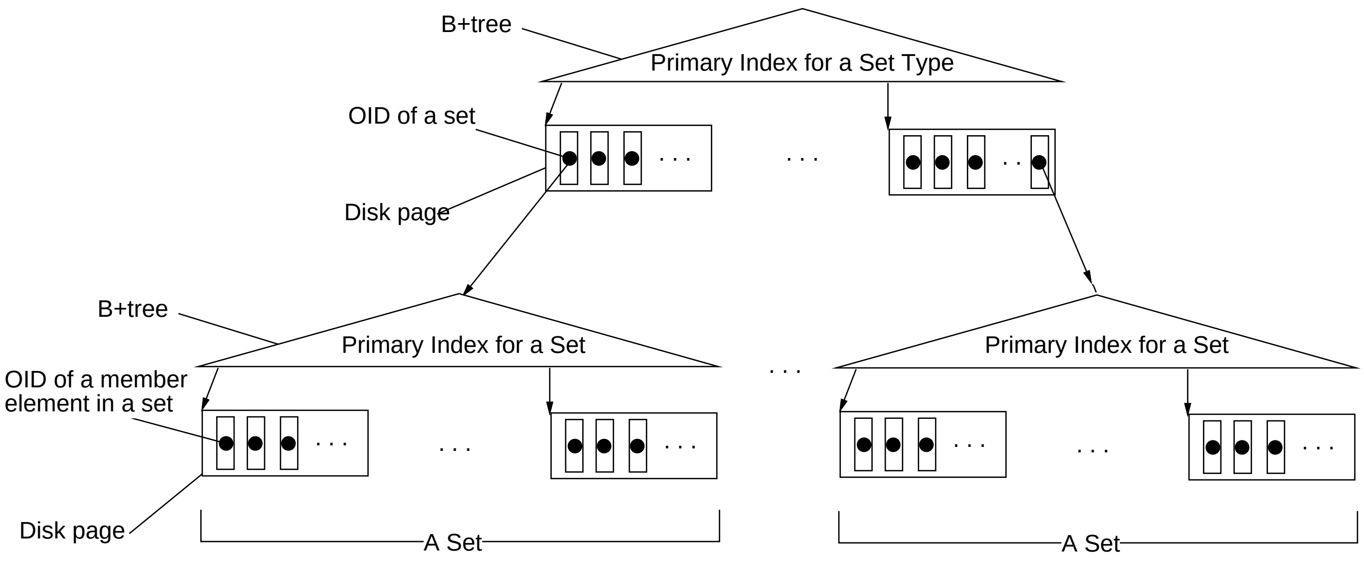

Figure 16: Internal structure of a set-type object.

4.4 Performance Enhancement with Persistent Caching

Complex objects are able to represent persistent data structures dynamically, but this virtue appears to be at the expense of the physical manipulation. The use of only the above straightforward internal structures certainly results in poor performance in terms of "navigation" operation, as this follows the OIDs in accessing the referenced objects even though it is an essential operation in dealing with complex objects. Navigation is difficult to implement efficiently since every operation inherently causes single disk access operation. To overcome this we developed a technique termed *persistent caching* [19, 17] to notably accelerate operation of the navigation by increasing the effective number of objects on one disk page. The main concept behind the technique is threefold. The first is to store a cached value within a complex object referencing another complex object. Second, when a referenced object is to be updated, update propagation to the persistent caches is delayed until the cached value is referenced. The third idea involves the use of a hashed table on the main memory to efficiently validate the consistencies between the cached values and the original values. Further discussion on the algorithm and an analysis of performance are given in reference [19].

In the implementation of a complex object file system, the persistent caching technique is used extensively. (Refer Fig. 17.) When a tuple object includes reference to another object stored on a different disk page (page 2 in figure), the values of "referenced" objects can be copied and retained on the disk page (page 1 in figure) on which the "referencing" object is stored as a persistent cache. As long as the the persistent cache is not invalidated, a navigation operation from the referencing object to the referenced object is conducted by accessing only the first disk page.

The persistent cache contains a copy of any information whose validity can be checked based on OID. A useful utilization is to persistently cache the physical address of referenced objects in referencing objects. This reduces the cost of object location by amortizing lookup costs over

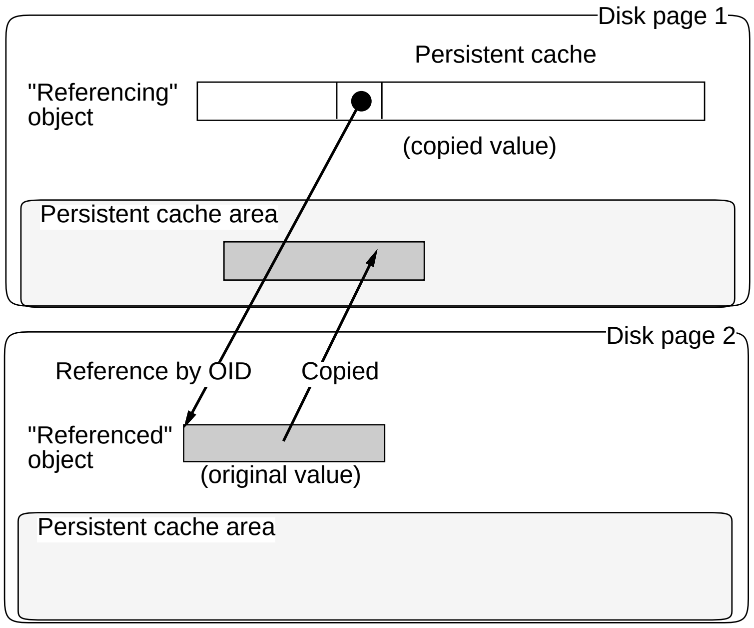

Figure 17: Tuple object with persistent caching.

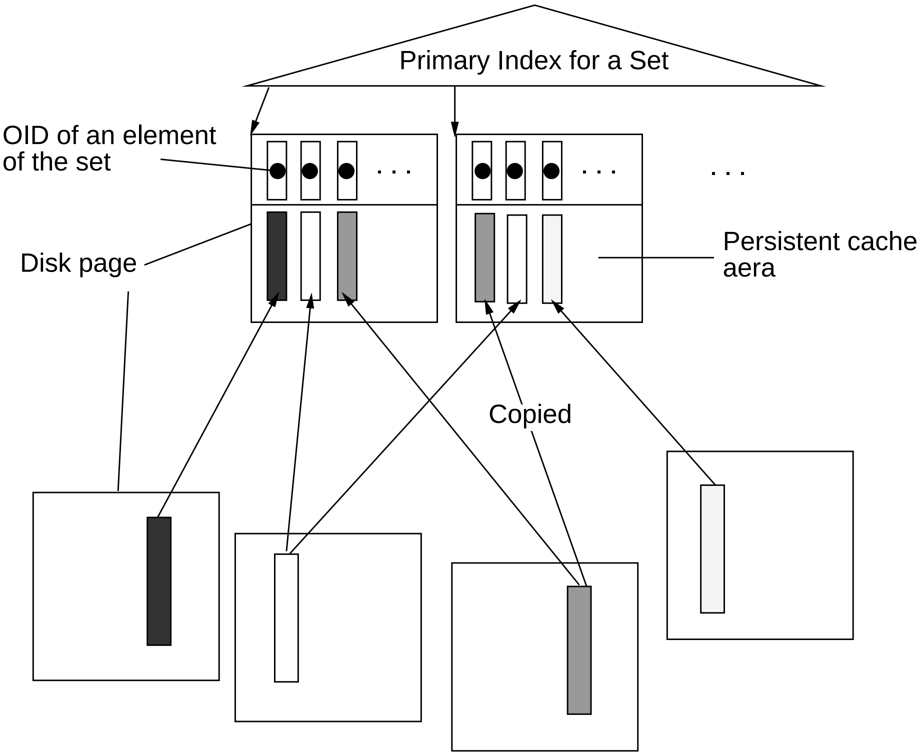

Figure 18: Set object with persistent caching.

multicasting location requests to network sites [36].

As with tuple objects, persistent caching accelerates accesses to elements of a set object. (See Fig. 18.) By persistently caching a portion or all the element objects in the disk pages storing set element OIDs, these elements can be directly accessed so long as the the persistent caches are not invalidated.

5 Conclusions

We described an open distributed computing environment namely the XERO operating system providing a programming model using multithreads, multicontexts, and the extended RPC mechanism. The internal structure of the operating system includes two-level multithread management, relocatable multicontext mechanisms. XERO supports a complex object file system to preserve data types in a file system. For efficient management of complex objects, persistent caching technique is used.

According to the design described in this paper, we are now implementing the XERO operating system. Although our objective is to realize a heterogeneous distributed computing environment, the current target machines which allow implementation are SONY NEWS workstations.

Future work will concentrate on the following research:

- Developing a set of tool kits to adapt existing programming language systems to the XERO programming model.
- Integrating a complex object space using RPC-communication. One idea is to use the concept of distributed shared data [14, 4].
- Developing an easy-to-use manipulation language and a transaction model for distributed processing to complex objects [20].
- Managing replicas of complex objects to quickly access in distributed environments [38, 8].

Acknowledgments

We would like to thank the Supermicro Systems Group of SONY Corporation for providing technical information on SONY NEWS workstations. The authors also would like to thank anonymous referees for their helpful comments in improving the clearness of this paper.

References

[1] S. Abiteboul, C. Beeri, M. Gyssens, and D. V. Gucht. An introduction to the completeness of languages for complex objects and nested relations. In S. Abiteboul, P. C. Fischer, and H. J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 117–138. Springer-Verlag, 1989. LNCS-361.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of First Int. Conf. on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, December 1989.

[3] M. J. Bach. *The design of the UNIX operating system.* Prentice-Hall, 1986.

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322, 1989.

[5] F. Bancilhon. Object-oriented database systems. In *Proc. ACM Symp. on Principles of Database Systems*, pages 152–162, 1988.

[6] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Thirteenth Int. Conf. on Very Large Data Bases, Singapore*, 1987.

[7] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 22:314–333, March 1988.

[8] S. Chiba, K. Kato, and T. Masuda. Optimization of distributed communication in multiprotocol tuple space. In *Proc. IEEE Third Symp. on Parallel and Distributed Processing*, December 1991. To appear.

[9] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2), Jun. 1979.

[10] F. J. Corbató, C. T. Clingen, and J. H. Saltzer. Introduction and overview of the MULTICS system. In *Proc. AFIPS, FJCC*, pages 185–196, 1965. Vol. 27.

[11] J. R. Corbin. *The art of distributed applications: programming techniques for remote procedure calls.* Sun Technical Reference Library. Springer-Verlag, 1990.

[12] P. Dasgupta, R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. To appear in *IEEE Computer*.

[13] K. J. Dunlap and M. J. Karels. Name server operations guide for bind release 4.8. Technical report, University of California Berkeley, 1989.

[14] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[15] IBM Corporation, General Systems Division, Atlanta, CA. *IBM System/38 Technical Developments*, July 1980.

[16] S. Inohara, K. Kato, A. Narita, and T. Masuda. A thread facility based on user/kernel cooperation in the XERO operating system. In *Proc. IEEE 15th Int. Computer Software & Applications Conference (COMPSAC)*, pages 398–405, Tokyo, September 1991. Also available as Technical Report 91-007, Department of Information Science, University of Tokyo.

[17] K. Kato. Storage management techniques of object-oriented database systems. *Journal of Information Processing Society of Japan*, 32(5):532–539, May 1991. In Japanese.

[18] K. Kato, S. Inohara, K. Wakita, and T. Masuda. XERO: A distributed operating system for DBMS and database applications. *IEICE Research Reports of Computer System, Japan*, CPSY-89-29, 1989. In Japanese.

[19] K. Kato and T. Masuda. Persistent caching: An implementation technique for complex objects with object identity. Technical Report 89-021, Department of Information Science, Faculty of Science, University of Tokyo, 1989.

[20] K. Kato, T. Masuda, and Y. Kiyoki. A comprehension-based database language and its distributed execution. In *Proc. IEEE Tenth Int. Conf. on Distributed Computing Systems*, pages 442–449, Paris, May 1990.

[21] K. Kato and A. Ohori. An approach to multilanguage persistent type system. In *Proc. IEEE 25th Hawaii Int. Conf. on System Sciences*, January 1992. To appear. Also available as a Technical Report of Deparment of Information Science, University of Tokyo.

[22] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on higher-order remote procedure call techniques. Submitted for publication.

[23] E. Levy and A. Silberschatz. Distributed file systems: concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[24] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9:50–59, 1974.

[25] S. Miyazaki and K. Kawagoe, editors. *Journal of Information Processing Society of Japan: Special Issue on Object-Oriented Database Systems*, volume 32. Information Processing Society of Japan, May 1991. In Japanese.

[26] T. Murakami, K. Kato, and T. Masuda. On the distributed C language based on remote procedure calls. In *Proc. IPSJ 42nd Conf.*, March 1991. In Japanese.

[27] A. Ohori. Formalization of object-oriented databases. *Journal of Information Processing Society of Japan*, 32(5):550–558, May 1991. In Japanese.

[28] A. Ohori and K. Kato. Higher-order remote procedure calls. Available as a Technical Report of Deparment of Information Science, University of Tokyo.

[29] Z. M. Ozsoyoglu and L-Y. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1), Mar. 1987.

[30] J. Rees and W. Clinger (editors). The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[31] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[32] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 4:13, Dec. 1988.

[33] R. M. Stallman. *Using and porting GNU CC*. Free Software Foundation, Inc., 1991.

[34] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), July 1981.

[35] SUN OS Release 4.0 Reference Manual, SUN Microsystems. *Lightweight Process Library*, 1988.

[36] D. B. Terry. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):48–54, January 1987.

[37] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach threads and the Unix kernel: the battle for control. In *Proc. Summer 1987 USENIX Conf.*, July 1987.

[38] T. Tonouchi, K. Kato, and T. Masuda. Persistent caching algorithm on distributed environment. *IEICE Research Reports of Computer System, Japan*, CPSY-90-51:79–84, 1990. In Japanese.

[39] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[40] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proc. of the 12th Symp. on Operating Systems Principles*, pages 114–122. ACM, December 1989.