

A metaobject protocol for accessing file systems

Chris Maeda

Published in Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS), Kanazawa, Japan. Springer-Verlag LNCS 1049. 1996.

© Springer-Verlag Berlin Heidelberg 1996.

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

A Metaobject Protocol For Accessing File Systems

**Chris Maeda
Carnegie Mellon University
School of Computer Science**

**cmaeda@cs.cmu.edu
15 Lumahai St.
Honolulu HI 96825
USA**

**808-395-5993
FAX: 808-395-0296**

keywords: metaobject protocols, operating systems, file systems, object-oriented programming, open implementations, microkernels

Submitted to ISOTAS '96

A Metaobject Protocol For Accessing File Systems

Chris Maeda (cmaeda@cs.cmu.edu)

Carnegie Mellon University

Submitted to ISOTAS '96

1. Abstract

This paper presents the design of a metaobject protocol (MOP) for accessing data stored in file systems. The MOP exposes an abstraction of the file cache machinery that is an inherent part of every file system implementation, and allows applications to control cache management decisions for the files that they use. Safety and protection are preserved by designing the file cache abstraction so that the operating system retains control over which applications may access data for each file and how global resource allocation decisions are made.

2. Introduction

A key problem for file system implementors is that application programs access data in a wide variety of different patterns. The fact that the file system implementation includes a file data cache means the file system would like to be aware of the application-specific access pattern, so as to pick an optimal cache management strategy --- prefetching file data into the cache before it is needed, and flushing it from the cache as soon as it is no longer needed. In the absence of knowing what the application-specific access pattern will be, operating system designers have been forced to implement one or more system-wide cache management policies designed to provide good performance for a wide range of applications. For example, the UNIX operating system typically implements an LRU-like cache management policy plus some special purpose code for handling sequential file scans [leffler89]. This approach is fine when the application is well-served by the standard cache management policy. However, several important applications, like those in multimedia and relational database processing, exhibit poor performance using the standard policy [stonebraker81].

Work on open implementation [kiczales92, kiczales94a, kiczales94b] suggests that the root of the problem is that the file system is a black box abstraction. Inside the black box, the implementation makes policy decisions based on erroneous assumptions about application behavior. Outside the black box, the applications have intimate knowledge of their own behavior but no way to communicate this information to the file system implementation. The result is one or more *hematomas*, where the application must code around the assumptions made by the file system. These hematomas make programs harder to build and maintain, reduce portability, and make performance unstable.

Work on metaobject protocols (MOPs) for programming languages [kiczales91] suggests a way to attack this problem. A MOP provides a way for clients of a system to modify the system's behavior to better suit their needs. A MOP for file systems could provide a way for applications to

tune the file cache management to their actual reference patterns. Unlike programming languages, however, MOPs in the context of an operating system must be careful to preserve the safety and protection guarantees provided by the operating system. Otherwise, a MOP could be used to compromise system integrity.

This paper presents the design of a MOP for accessing data stored in file systems. The MOP exposes an abstraction of the file cache machinery that is an inherent part of every file system implementation, and allows applications to control cache management decisions for the files that they use. Safety and protection are preserved by designing the file cache abstraction so that the operating system retains control over which applications access the data for each file and how global resource allocation decisions are made. The MOP is implemented, and is currently being tested in order to refine both its design and implementation.

Section 3 presents an analysis of the client interface abstraction provided by file systems, the way it is typically implemented, and how the typical implementation is unsuitable for certain applications. Section 4 enumerates the requirements for a file system MOP. The design of the MOP is presented in Section 5, and the ways in which one would use it are presented in Section 6. Finally, the status of the implementation is presented in Section 7.

3. Domain analysis

This section presents an analysis of the file abstraction in the context of the Unix operating system. The analysis is not Unix-specific, however, since most operating systems have a similar or identical file abstraction. For example, the Unix system calls that operate on the file abstraction have a natural mapping to the file functions in the Win32 API implemented by Windows NT and Windows 95 [microsoft93]; i.e. the UNIX `read` system call corresponds to the Win32 `ReadFile` function.

3.1 *The file abstraction*

In Unix, a file is conceptually an array of bytes. Clients interact with files by opening them, which causes an open file object to be created. An open file object contains a handle for a file and an offset into the file's contents. Open file objects are named by user processes using file descriptors; a file descriptor is an index into a per-process file descriptor table, which is essentially an array of pointers to open file objects. Open file objects may be named by more than one file descriptor. For example, the `fork` system call creates a child process whose file descriptor table contains references to the same open file objects as the parent, while the `dup` system call takes a file descriptor and creates a new file descriptor that refers to the same open file object. Figure 1 shows these objects and how they relate to each other.

Clients access the contents of a file through the `read` and `write` system calls, which copy regions of the file to and from memory buffers in the process's address space. The `read` and `write` system calls access the file contents at the current file offset, and update the offset to point to the byte following the bytes read or written. An additional system call, `seek`, is used to update the offset directly.

3.2 Implementation

File systems are typically implemented using an in-memory cache of file contents. The reason for this strategy is that disks are several orders of magnitude slower than CPUs. Thus CPUs must access file contents from in-memory caches in order to achieve reasonable performance. While greatly improving performance, file caches also greatly complicate the implementation. The implementation must decide how to allocate cache slots to the various files, when to fetch parts of files into the cache, and when to reclaim cache slots. These cache management policy decisions are usually set by observing a “typical” application mix and constructing a policy that works well for it [ousterhout85]. For example, the UNIX operating system manages cache slots using a global LRU list and prefetches blocks when it detects a sequential scan of a file’s contents [leffler89].

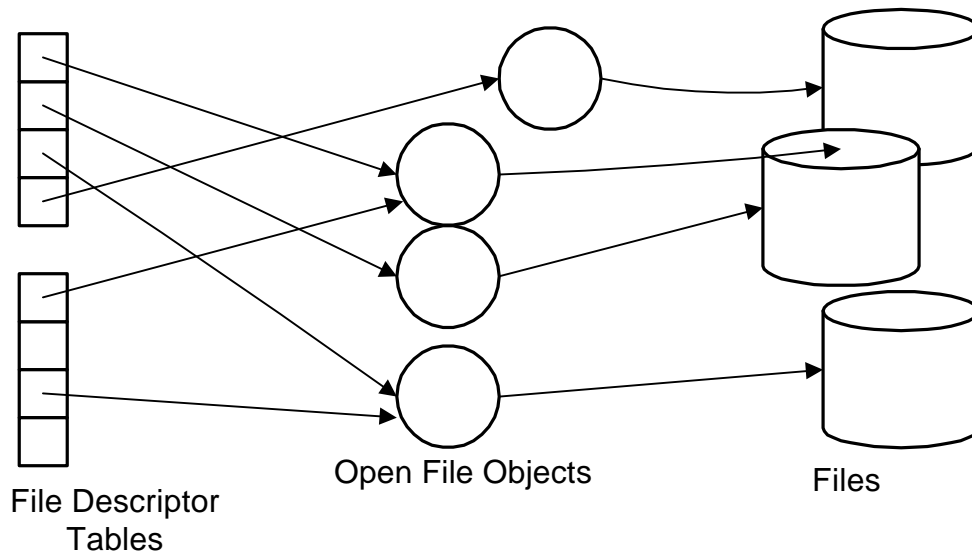


Figure 1: Files, open file objects, and file descriptor tables in the Unix operating system. Each process has a private file descriptor table that references open file objects. Each open file object encapsulates a reference to an actual file, and an offset into the file’s contents.

3.3 Problem applications

Some applications have file access patterns that interact badly with the system’s cache management policy. For example, relational databases access large files in patterns that, to the file system, appear essentially random, and almost always miss in the file cache as a result. Implementors of relational database systems typically build their own cache management modules in order to work around the system buffer cache [stonebraker81]. Another example is implementors of differential video encoding schemes, such as MPEG, who would like to ensure that base frames remain in the file cache while delta frames are discarded as soon as they are decoded and displayed. Since they have no way of having the file system cache do this, they must resort to keeping base frames in virtual memory buffers. Because the file cache manager provides no way for applications to modify the cache management policy, file data is double buffered with a concomitant increase in system overhead in the form of physical memory consumption.

4. Requirements for the meta-interface

The basic requirements for the meta-interface are easy to state: it must provide application programmers with appropriate mechanisms to inform the file system about the application's specific file access pattern. This functionality can be further divided into:

- Introspective mechanisms, that allow the application to ask about the current state of the buffer cache --- which file blocks are resident and which are in transit to and from the disk.
- Interspersory mechanisms, that allow the application to change or replace the file system's cache management strategy.

The meta-interface must provide incrementality: clients should be able to make small changes in an existing cache manager in order to get small changes in the cache management policy. The meta-interface must also provide scope control; if a client changes the cache management policy for a file, the policy for other files will not be affected.

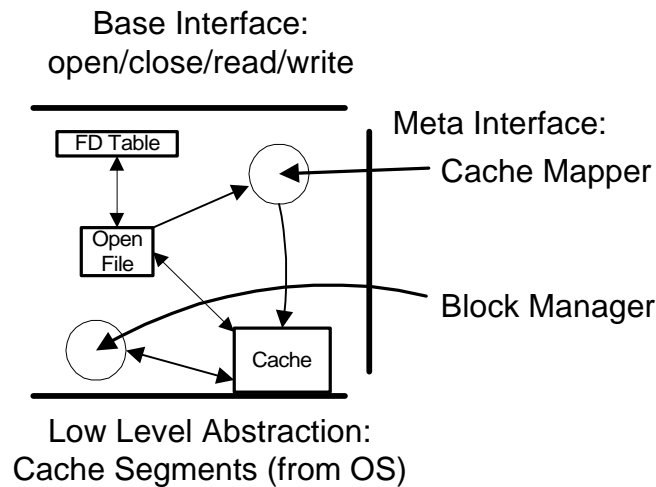


Figure 2: The File System MOP is part of a library that is layered on top of a low level cache abstraction provided by the operating system, and that implements the file abstraction. The meta-interface allows cache management decisions to be tailored to the needs of the client application.

5. Design

This section describes the design of the file system MOP, starting with its overall structure and progressing to the details of each of the metaobjects. Figure 2 shows the overall structure of the MOP. The Base Interface is simply the ordinary UNIX file abstraction described in Section 3.1.

The MOP itself implements the UNIX file abstraction using *cache segments*, a low level abstraction provided by the operating system. A cache segment is a virtual buffer cache that is mapped into the address space of a user process. The implementation manages one or more cache segments; each cache segment contains data from one or more files.

Previous examples of MOPs did not emphasize the importance of the low level abstraction. However, in a file system MOP, where issues of sharing and protection are paramount, careful design of the low level cache segment abstraction is necessary to preserve the safety and protection guarantees of the operating system. Cache segments are the mechanism that provides scope control and incrementality to the meta-interface: clients may have several cache segments and may set cache management policy on a per cache segment basis. Policy changes for one cache segment do not affect the behavior of other cache segments in the system.

The MOP itself consists of five metaobjects: cache segments, open files, file descriptors, the Cache Mapper, and the Block Manager. Of these, only the Cache Mapper and Block Manager may be specialized or replaced by the client. Since the base interface provides UNIX system call semantics, the MOP implements a file descriptor table and open file metaobjects just as in UNIX. The Cache Mapper metaobject determines which cache segment object is used to access each open file. It is also responsible for creating and destroying cache segment metaobjects when necessary. The Block Manager metaobject is responsible for implementing the fetch and flush policy used for each cache segment metaobject. The cache segment metaobject calls the Block Manager when the client program accesses data that is not in the cache. The Block Manager is responsible for loading the data into the cache, and it may also flush or prefetch other data in the cache at this time.

By querying the Cache Mapper and Block Manager metaobjects, application programmers may discover the current cache state for each file. By providing new implementations of these metaobjects, application programmers may influence how many cache buffers are allocated to each file and how the cache management policy is determined for them. The cache segment abstraction preserves the safety and protection guarantees of the operating system by ensuring that the operating system decides how physical memory pages are allocated to the cache segments, and by enabling it to reallocate physical memory when necessary.

The remainder of this section describes the protocol for each metaobject. Some readers may wish to skip to Section 6, which shows how the MOP can be used, and refer back to this section as needed. Note that the MOP is implemented in C, since it must be linked with existing C programs. Thus the metaobjects may only be replaced at link time. However, user-supplied implementations of the metaobjects could allow more fine-grained (i.e. per cache object) modifications by implementing a virtual function scheme on top of the basic call interfaces described below.

5.1 File descriptors and open files

The protocols for the file descriptor and open file objects mirror the UNIX system calls required by the file abstraction. For example, the MOP's implementation of the open system call creates a new file descriptor object and a new open file object, stores them in the file descriptor table, and initializes the open file's offset to zero. The read and write system calls require more interaction with the cache object, since the open file object is responsible for mapping the UNIX file-as-byte-stream abstraction to the cache's view of files as a sequence of blocks. The implementation of read, for example, computes the file blocks that contain the data at the current offset and asks the cache object to load them. When the cache object has done so, the file object returns the data to the caller and updates the file offset.

5.2 Cache segments

A cache segment is a region of virtual memory divided into page-size slots. Each slot contains resident data from a single open file; each cache segment may contain pages of data from one or more files. The contents of the cache segment are controlled by a cache segment manager that loads parts of files into memory and maps them into the cache segment's memory region.

The contents of the cache segment are described by a cache descriptor region that is mapped read-only by the client and managed by the cache segment manager. The descriptor region is an array of slot descriptors that each describe the contents of a single slot. The client may consult the descriptor region to quickly determine if a given file page is accessible through the cache segment. If it is, the file data may be accessed by reading the contents of the cache slot. If the file data is not accessible, it must be loaded into the segment using the cache control interface.

The manager may revoke the mapping to a given page unilaterally and at any time; for example, when another application locks the mapped page. When a page is revoked, the manager informs the application through an upcall mechanism. The application may not process the upcall in time, however, and try to access the revoked page anyway. When this occurs, the application will fault on the page. In order to recover, the faulting application is restarted in a fault handler that retries the access using the normal mechanism or requests the manager to reload the page into the cache segment.

5.2.1 Initializing cache segments

To create a new segment, the client makes a `segment_create` call (see Table 1) with the requested segment size. The segment manager creates new memory objects for the descriptor and the data area and maps them into the client's address space. The virtual addresses are then returned to the client along with the page size used in the cache segment.

By creating a segment, the operating system implicitly allows the client to use as many physical memory pages as the cache segment can hold. The design can support resource negotiation by adding an intermediate step where the client negotiates to get a capability for some number of physical pages. This capability could then be supplied with the `segment_create` call.

API Call	Description
<code>segment_create</code>	Create a new cache segment of the requested size and map it into the client's address space. Return the virtual addresses of the segment descriptor and data area, as well as the cache block size.
<code>segment_map</code>	Associate an open file (named by a Unix file descriptor) with a segment. The segment manager returns a file handle for the open file, which is used in subsequent <code>get</code> and <code>put</code> calls.

Table 1: Segment creation API

The `segment_map` call is used to associate an open file with a cache segment. The client must supply a capability for the open file in the `segment_map` call. In the current implementation, the file capability is a Unix file descriptor. The segment manager first verifies that the file may be mapped, and then maps it by linking the file's data structure to the segment data structure. The manager also allocates a file handle for the file, which is a unique number used to name the file in subsequent cache management requests.

5.2.2 Cache control interface

Once open files are mapped to cache segments, the client can manage the cache contents by asking the segment manager to load or unload pages from the cache. The client and segment manager communicate through an asynchronous interface described in Table 2.

The `segment_get` call is used by the client to load a slot into the cache segment. The client supplies a cache slot, a file handle for a previously mapped file (see Section 5.2.1), and a file offset. After sending the `segment_get` message, the client may either block waiting for the response, or may do other things while the get operation is in progress. The segment manager loads the cache slot and sends the client a `segment_didget` message.

API Call	Description
<code>segment_get</code>	Ask segment manager to load a page of a file into the specified slot of the cache segment. Segment manager will fetch the page and send a <code>segment_didget</code> message when the slot is loaded.
<code>segment_put</code>	Ask segment manager to unload a slot from the cache segment. The file page will be placed on the manager's pageout queue for eventual writeback or reclamation.
<code>segment_flush</code>	Ask segment manager to unload a slot from the cache segment and immediately flush it to disk. This is used by programs, such as relational databases, that need to ensure persistent data has been committed to disk.
<code>segment_didget</code>	Used by segment manager to inform client that a cache slot has been loaded.
<code>segment_didput</code>	Used by segment manager to inform client that a cache slot has been unloaded. The segment manager may unilaterally unload a cache slot and send this message.
<code>segment_didflush</code>	Used by segment manager to inform client that a cache slot has been unloaded and written to disk.

Table 2: Cache control API

The `segment_put` and `segment_flush` calls are used to unload the contents of a cache slot. The flush call unloads the slot and immediately writes the file data to disk. The put call unloads the slot and schedules the file data for eventual writeback. Delaying the write improves overall system throughput, but some applications need immediate writeback for correctness. The segment manager sends the `segment_didput` or `segment_didflush` messages when the operation completes. The segment manager may also send didput messages unilaterally, whenever it needs to revoke segment access to a file.

5.3 Cache Mapper

The Cache Mapper meta object determines which cache segment will be used to access a given file. Table 3 shows the protocol for the Cache Mapper meta object. It has one function which is called when an open file is being mapped to a cache object. The meta object should pick an appropriate cache object, creating a new one if necessary.

Function	Description
<code>cache_choose(file)</code>	Called when an open file object needs to be mapped to a cache object. The implementation of this function should return an appropriate cache object, creating a new one if necessary.

Table 3: The Cache Mapper protocol

Function	Description
<code>cachemgr_init(cache)</code>	Called when a new cache object is created. Any block manager state should be initialized.
<code>cachemgr_findblock(cache, file, offset)</code>	Called when the cache object needs a given block of a file. Returns the block number if the block resident, otherwise returns -1. The caller must call <code>cachemgr_unref</code> when it has finished with the block.
<code>cachemgr_fetchblock(cache, file, offset)</code>	Called when the cache object would like a file block loaded into the cache. Allocates a cache slot, writing back the current contents of the block if necessary, and asks the segment manager to load the slot with the file block. Returns the block number. The caller must call <code>cachemgr_unref</code> when it has finished with the block.
<code>cachemgr_unref(cache, block)</code>	Called when a block is no longer needed. The block manager may reuse the cache slot.
<code>cachemgr_didget(cache, file, offset, block)</code>	Called when a cache slot is filled by the cache segment manager.
<code>cachemgr_didput(cache, block)</code>	Called when a cache slot has been removed by the cache segment manager.
<code>cachemgr_getfail(cache, file, offset, block)</code>	Called when a request to fill a cache slot has failed.
<code>cachemgr_putfail(cache, block)</code>	Called when a request to remove a cache slot has failed. (The cache slot is still free, but the file write failed.)

Table 4: The Block Manager protocol

5.4 Block Manager

The Block Manager metaobject determines which fetch and flush policy will be used to manage a given cache segment. Table 4 shows the protocol for the Block Manager metaobject. When an application wants to access file data, it calls a function provided by the cache object. The cache object in turn calls the `cachemgr_findblock` function to see if the file block is resident in the cache. If the block is resident, `cachemgr_findblock` returns the index of the cache slot that contains the file block. The cache object then accesses the data and calls `cachemgr_unref` when it has finished. If the file block is not resident, `cachemgr_findblock` returns -1. The cache object then calls `cachemgr_fetchblock` to load the file block into the cache. The `cachemgr_fetchblock` function must choose a cache slot to hold the file block (writing back the current contents of the slot if necessary) and have the cache segment manager load the slot with the file data. Once the cache manager has loaded the file block into the cache slot, it returns the index of the cache slot. The cache object then performs the access as described above and calls `cachemgr_unref` when it has finished.

The other functions in the Block Manager protocol (`cachemgr_didget`, etc.) correspond to the callback functions provided by the cache segment protocol (see Table 2). They must be called so that the Block Manager may update its internal state whenever the underlying cache segment undergoes a state change.

6. Using the MOP

This section discusses how applications can use the MOP to discover and influence the cache management policy applied to files that they are accessing. The default implementation of the metaobjects provides a simple LRU list manager that is adequate for many needs. The implementation of this manager is described in Section 6.1. Section 6.2 shows how the MOP can be used to support applications that scan files sequentially. Finally, Section 6.3 sketches how a relational database could make use of the MOP.

6.1 The default LRU list manager

The default implementation of the metaobjects implements a simple LRU list manager. The manager keeps a freelist of blocks that contain no file data, and an LRU list of blocks that contain data but are not currently in use. When `cachemgr_init` is called, the manager creates a block data structure for each cache slot that contains the cache slot index and a reference count, and places them on the freelist.

When the cache object calls `cachemgr_findblock`, the manager iterates over the cache segment descriptor (see Section 5.1) to find the desired file block. If the file block is found, the manager updates the reference count in the block data structure and returns the index of the cache slot. Otherwise it returns -1.

When `cachemgr_unref` is called, the block reference is decremented. A block is placed on the LRU list when its reference count goes to zero.

When `cachemgr_fetchblock` is called, the manager first tries to use a block from the freelist. If the freelist is empty, it takes the first clean block from the LRU list. It incidentally schedules writes for any dirty blocks that it finds before the first clean block. If no clean blocks are found, for example if all blocks were dirty or if the LRU list is empty, the manager blocks until the block

writes complete or some other thread places a block on the LRU list. Once the manager finds a cache slot, it asks the cache segment manager to load the slot with the appropriate file data. When this operation completes, the manager returns the index of the cache slot.

6.2 Sequential scans

A common file reference pattern is a sequential scan [ousterhout85]. In fact, this pattern is so common that many operating systems have special-case code in their buffer cache managers to handle it [leffler89]. An application programmer can use the MOP to easily implement a cache management policy suitable for sequentially scanned files. When `cachemgr_init` is called, the new Block Manager should prefetch the first *n* blocks of the file to be scanned, where *n* is the number of cache slots available. When `cachemgr_unref` is called and the Block Manager knows the program has finished scanning the block, the block should be immediately written back using `segment_put`, and the next block immediately prefetched with `segment_get`. Since the cache segment get and put interfaces are asynchronous calls, the application can continue scanning the file while the disk I/O operations are in progress.

6.3 An RDBMS cache manager

The MOP is ideal for implementing file cache managers for relational database systems. For example, the DBMIN buffer management algorithm [chou85] assumes the database has a global buffer pool that is shared by every open file. Each database file is classified by its current reference pattern, and the number of buffer pages needed to hold the file's working set is computed. When a cache miss occurs, the DBMIN algorithm determines which file page to replace based on the file's current reference pattern. The global buffer pool in DBMIN maps naturally to a single cache segment, and the DBMIN replacement algorithm, which is essentially a generalization of the scheme in Section 6.2, can be implemented in a new Block Manager metaobject.

7. Status

An initial implementation of the MOP has been completed as part of a system based on the Mach 3.0 microkernel and a user-level server providing compatibility with Digital UNIX version 2.1. Cache segments are implemented using the Mach External Memory Management interface [young89]. The pager for the cache segments resides in the UNIX server and shares the same page pool as the UNIX buffer cache. The MOP is currently being refined by using it for a variety of file intensive applications such as the UNIX compress and sum programs, and an MPEG video decoder. In addition, an implementation of an RDBMS cache manager discussed in Section 6.2 is being planned for the Postgres95 relational database system.

Performance experiments are in progress, and numbers should be available for the final paper. Cache segments allow buffer cache pages to be accessed without requiring a context switch into the kernel or to a user-level server. This fast path to the resident file data, combined with a better cache hit rate due to the application-specific cache management policies enabled by the MOP, should result in performance that will be at least as good as that of monolithic kernel operating systems like Digital UNIX.

8. References

- [chou85] Chou, H. T., and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. Proceedings of the 11th International Conference on Very Large Databases. 1985. Reprinted in [stonebraker88], pages 174-188.
- [kiczales91] Kiczales, G., J. des Rivieres, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press. 1991.
- [kiczales92] Kiczales, G.. Towards a New Model of Abstraction in Software Engineering. Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architecture. 1992.
- [kiczales94a] Kiczales, G.. Foil for the Workshop on Open Implementation. Xerox PARC. 1994. Available at <http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94/foil/main.html>.
- [kiczales94b] Kiczales, G.. Why are Black Boxes So Hard to Reuse? Invited talk, OOPSLA '94. 1994. Available at <http://www.parc.xerox.com/PARC/spl/eca/oi/gregor-invite.html>.
- [leffler89] Leffler, S. J., M. K. McKusick, M. J. Karels, and J. S. Quarterman. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley. 1989.
- [microsoft93] Microsoft Corporation. Microsoft Win32 Software Development Kit for Windows NT Programmer's Reference Manual. Microsoft Press. 1993.
- [ousterhout85] Ousterhout, J. K., H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2BSD File System. Proceedings of the Tenth Symposium on Operating Systems Principles, pages 15-24. 1985.
- [stonebraker81] Stonebraker, M.. Operating System Support for Database Management. Communications of the ACM, volume 24, number 7, pages 412-418. July 1981.
- [stonebraker88] Stonebraker, M., editor. Readings in Database Systems. Morgan Kaufmann. 1988.
- [young89] Young, M. W.. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Technical report CMU-CS-89-202. Carnegie Mellon University, Computer Science Department. 1989.