A SERVER OF DISTRIBUTED DISK PAGES USING A CONFIGURABLE SOFTWARE BUS

Charles Falkenberg, Paul Hagger and Steve Kelley

Institute for Advanced Computer Studies and
The Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

As network latency drops below disk latency, access time to a remote disk will begin to approach local disk access time. The performance of I/O may then be improved by spreading disk pages across several remote disk servers and accessing disk pages in parallel. To research this we have prototyped a data page server called a Page File. This persistent data type provides a set of methods to access disk pages stored on a cluster of remote machines acting as disk servers. The goal is to improve the throughput of database management system or other I/O intensive application by accessing pages from remote disks and incurring disk latency in parallel. This report describes the conceptual foundation and the methods of access for our prototype.

With oversight by Office of Naval Research, this research is supported by ARPA/SISTO in conjunction with the Domain Specific Software Architectures project.

Contents

1 INTRODUCTION

With the goal of achieving parallel I/O on a large data space we have created a persistent data type called a Page File. The methods are designed to be used by data intensive applications such as a database management system to read and write disk pages. The page size can vary for each Page File and pages can be stored on a local disk or spread across the disks of a cluster of remote processors. The Page File abstraction allows page reads and writes to go on without any knowledge of the number of remote disks or the remote allocation of pages. Our prototype exploits the parallelism available when pages are accessed on multiple remote disks simultaneously. In addition, we hope to increase the available disk space and minimize disk contention.

Network communication in the prototype is based upon *software bus* organization using the POLYLITH software interconnection system [Purt9X]. Software bus organization provides a single communication interface for applications, written in different languages and distributed across a network of diverse computers and operating systems. Because of these benefits, the prototype we built on the POLYLITH system may be easily reconfigured for purposes of experimentation.

The access methods have been tailored to meet the needs of the database management system ADMS [Rous9X]. ADMS utilizes incremental access methods and caching to improve the performance of large distributed databases. The access methods of our prototype are designed to fulfill the I/O requirements of ADMS. Existing I/O access methods can easily be replaced by the methods of our prototype. A sample ADMS work load has been generated and is being used to test the performance of our persistent Page File objects.

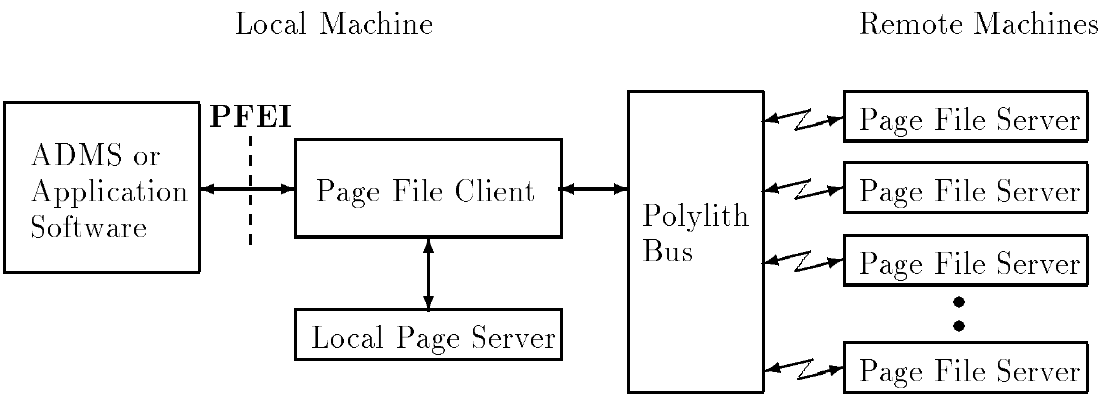

Figure 1: Overview of the components of the Page File data type.

Figure 1 shows the logical components of the prototype. An application can use the prototype by linking to the functions which make up the Page File External Interface (PFEI). Collectively these access methods make up the Page File client. For remote Page Files the client requests services from the POLYLITH bus which passes messages over a network to the Page File servers located on separate machines. Data and confirmations are passed back through POLYLITH to the Page File client and then returned to the requesting application. For local Page Files, page requests are fulfilled with a simple calls to local page server. When a Page File is opened the page size and the local or remote allocation must be supplied. All subsequent access is made with individual page numbers without referencing the size of the pages or the type of allocation.

A remote Page File is spread across the remote disks using an allocation strategy. We have currently implemented a round robin strategy in which each successive page will be stored on the next server in order. Other potential strategies include an adaptive strategy in which pages are shuttled between servers to minimize disk contention. If a Page File is locally allocated the pages are stored sequentially on the local disk. In any of these cases, pages are returned to the application through a single high level interface which is designed to utilize the potential for parallel I/O of remote pages.

This paper describes the conceptual foundation of the Page File type and the access methods which define it. Section 2 is a definition of the requirements which motivated the creation of the prototype. This includes the future research interest in this prototype. Section 3 contains a detailed description of the access methods which make up the Page File External Interface. This is intended to be a manual for developers interested in utilizing objects of this type.

2 MOTIVATION AND REQUIREMENTS

This section provides some background into the design and implementation of the prototype. The conceptual design called a The Tower of Pizzas is presented first. This is followed by a discussion of how the design has been implemented in order to achieve parallelism and scalability. Finally, the current status and future potential of the project is presented.

2.1 A Tower of Pizzas

Each remote server is an independent machine which contains all of the components found in the pizza box of one workstation: cpu, disk and operating system. This makes the remote cluster a Tower of Pizzas on which data can be stored and retrieved in parallel [Rous92]. Results from experiments using ADMS indicate that local disk latency is 2-3 times greater then the network latency. Therefore, if the disk latency for several different pages can be incurred remotely, in parallel, and the pages delivered to the client over the network, the average time required to access each page will be closer to the network latency. If all remote servers read pages simultaneously then the client can receive those pages from the network faster than if each page had been read independently from the local disk. In a database management system which where I/O is the bottle neck a significant improvement in throughput may be possible.

This improvement in access time may be assumes that several pages are to be read from the disk on the remote server instead of from the local disk. An additional gain will be achieved if the pages can be cached in the memory of the remote server. Each server is dedicated and so the cluster provides a large memory space used exclusively for disk caching. The access to a single page in a remote memory has the potential to be quicker the access to a page on the local disk.

Two levels of parallelism can be achieved with this design. First, disk pages can be requested by a single client from several servers simultaneously in order to incur disk latency in parallel. A second level of parallelism will be achieved by adding multiple clients to the same remote cluster of servers. When different clients request pages from separate servers the requests may be fulfilled without any disk contention. Figure 2 illustrates the configuration of multiple clients and servers as a fully connected bipartite graph.

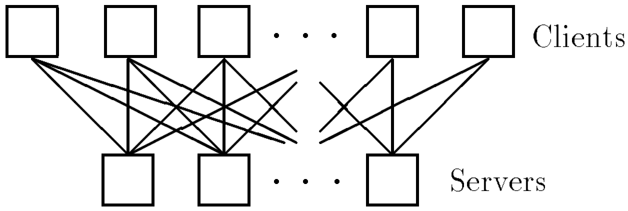

Figure 2: Bipartite graph of clients and servers.

The configuration in figure 2 a high potential for scalability of storage and throughput. When servers are added the total storage is increased and the first level of parallelism is increased. When clients are added more of the servers are kept active and the throughput is increased. As this second level of parallelism is increased greater advantage is taken of the page caching at each server. This scalability is an important advantage of this design.

Finally, using a cluster of disk servers allows the disk load to be balanced across all servers. Since each file is spread across the servers, each server can be equally loaded. This keeps the impact of very large files to a minimum and provides for the scalability of storage.

2.2 Page File Implementation

The first level of parallelism requires a new implementation of I/O. In order to be fulfilling multiple page requests simultaneously, reads and writes to a Page File must be done in two steps. The first step is a non-blocking request, the second a confirm. Several requests can be made to read or write pages activating the majority of the servers. After the first request is confirmed subsequent confirm operations may be completed incurring only the network overhead.

As an example several reads can be requested and if possible processing can continue. Each request is then confirmed and if the page is available it can be used. This presents some new problems but it is useful if several pages are needed before the application can continue or if a system is prefetching pages based on prior paging behavior. This prefetching can be achieved in a database management system where paging behavior is somewhat predictable. In addition, if some processing is to be performed on each page, it can be done to the first page that arrives for a single request allowing time for the other pages to arrive.

The servers may also perform some housekeeping in parallel while not filling client requests. Potential activities include flushing pages to disk and prefetching pages into the servers disk cache. These activities can be done by the each server after the write or read request has been fulfilled.

In order to achieve the second level of parallelism the number of both clients and servers will be varied leading to a great many possible configurations. The Polylith software bus will allow us to experiment with these different configurations with little or no modification of the application programs. In addition, Polylith allows us to use a variety of architectures to implement the network of distributed systems without any modification.

The prototype is modular by design to provide a solid foundation for a wide range of modifications which will suggest themselves during testing and reconfiguration. As part of this modularization two internal interfaces have been defined. The Page File Remote Interface (PFRI) is made up of functions calls for processing of remote Page Files. These are the functions which make use of Polylith to implement communication. The Page File Local Interface (PFLI) is a set of function calls to access the local disk. The local interface is used by the client if the Page is locally allocated or by the each remote server if the Page File is remotely allocated. Figure 3 shows the individual software modules and the important internal interfaces.

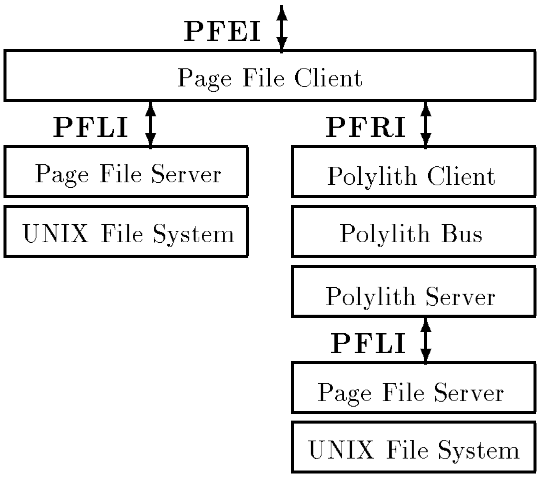

Figure 3: Software modules and interfaces.

This modularity allows us to isolate the components which are only used during remote processing. This is necessary in order to establish a benchmark of local vs. remote Page File processing. In addition, the Page File Remote Interface provides a high level view of the remote access through the Polylith bus.

2.3 Research Potential

This persistent data type will be used to research several aspects of distributed file processing. Various strategies for allocating and buffering pages at the server as well as at the client will be looked at and optimized for different configurations of clients and servers. The communication network will be optimized and different prefetching strategies will be tried in order to utilize the two stage read and write. The development supporting this research will be done in several phases.

The first phase of implementation distributes pages from a single client to multiple servers in a round robin fashion. This is the foundation of the prototype which will be modified to support the other research goals. This phase includes varying the number of servers and the buffering done at the client. In addition a work load processor has been built which executes a work load from an ADMS session. The Polylith configuration and client buffering will be optimized and the a benchmark of local vs. remote allocation will be established. This phase has been completed and the preliminary results indicate that for some work loads the remote allocation of pages has higher throughput than the allocation on local disk.

Subsequent phases will incorporate page buffering at the servers, page level locking, and multiple clients running on separate workstations. Buffering pages at the server is necessary to insure that the request for a remote page can be filled from memory as frequently as possible. Initially the MRU and LRU replacement policies will be tried but the distribution of pages across all servers may change the effectiveness of these traditional buffering strategies. Various page allocation strategies will also be tried to reduce disk contention at the servers.

Page level locking is a critical part of the transaction management in a database management system. Page locking in our prototype needs to be done at the server so that it can be seen by multiple client machines. Shared memory processes will be used at the server to support multiple clients while keeping lock information in memory.

Prototyping multiple clients and multiple servers will exploit a second level of parallelism and allow us to experiment with the scalability of the system. Polylith will provide a platform for easy reconfiguration of the prototype. As more clients are added the buffering strategies may need to be adjusted and several configurations will be attempted in order to quantify the scalability.

3 PAGE FILE EXTERNAL INTERFACE

The methods of access to the Page File can be broken into three groups. The highest level methods initialize and terminate the use of the type as a whole. At the middle level, methods provide services for a single Page File (eg. open, close). At the lowest level, accessors provide page services which includes reading and writing individual pages.

Type level methods are needed for Polylith as well as any other protocol to establish communications and allocate storage needed to administer the type. These methods are unique to the type and are similar to application initialization or housekeeping functions.

Page File level methods are similar in many respects to the related UNIX file system calls. Most do not return control to the application until the requested function completes successfully or terminates unsuccessfully.

Page level methods provide a new paradigm for accessing data. Data is retrieved one page at a time and the basic operations are not atomic. Pages are retrieved by first requesting a page number and then subsequently checking if the page has been returned. Pages are written by requesting a write and later checking to see if the write has been confirmed. This provides for a degree of parallelism during page reads and writes. Several disks can be in operation simultaneously as a result of several page read or write requests. These two operations are valid for locally allocated Page Files as well but no parallelism is gained.

The next sections describe the methods at each of the three levels. The function prototype and a brief description is given along with the error conditions and special parameters. The error conditions and the special values for any parameters are defined in `pfExternal.h`.

3.1 Error processing

Calls to the Page File functions can result in two types of errors. The first type are common UNIX file system errors. These include "file not found" or "invalid authorization". The second type are errors within the Page File system including "invalid allocation type" or "invalid file descriptor".

If an error is found all functions will return a negative value which matches matches `PF_ERROR` and the specific error number is in the variable `pfError`. This variable is defined in `pfExternal.h` and will contain both type of errors. UNIX system errors are positive and match the values specified for the particular UNIX system. The Page File errors are negative and match values defined for errors `pfExternal.h`.

3.2 Type level methods

In order to create Page File objects the type must be initialized. Initialization is required to activate the remote servers and set up communications. The remote servers must also be explicitly shut down and all open processing brought to a close which is done with the terminate type function.

3.2.1 Start Page File processing

`(int RtnVal)` ← `pfStart(int* argc, char*** argv, int NbrFil, int NbrRqs)`

This function initializes the data type by building the necessary run time structures and initiating communication. The argument count and the argument vector (`argc` and `argv`) passed into the application program are used and modified in this function. `NbrFil` is the maximum number of open Page Files at any one time and `NbrRqs` is the maximum number of unfulfilled read or write requests at any one time.

Pointers to `argc` and `argv` are used to extract any parameters needed by POLYLITH and then the parameter count and vector are modified to reflect the the parameters passed to the application only. If the application is started by POLYLITH this function must be called before any parameters are extracted from `argv`.

If start up is done successfully a positive value is return which matches `PF_SUCCESS`. If an error occurs a negative value is returned from which matches `PF_ERROR` and the error code can be found in `pfError`. The possible Page File errors are as follows:

`PF_LISTERR`	Error during creation of internal lists.
`PF_LOCALERR`	Error in local startup.
`PF_REMOTEERR`	Error in the remote startup.

3.2.2 Terminate Page File type

`(int RtnVal)` $\leftarrow$ `pfTerminate()`

This function closes all open Page Files and terminates communication with the remote servers. After it is called no new Page Files can be opened. Storage used for internal structures is freed and all remote servers are terminated. This function must be called before the program completes if the initialize function was called. No errors are returned by this function.

3.3 Page File level methods

The file level methods provide operations to open, close, and lock page files. Since most of these methods are blocking control is not returned to the requesting application until the Page File request succeeds or fails. If the Page File is stored remotely these functions contact all servers. If the Page File is stored locally these functions will call functions from the C library to perform the requested operation. These functions are generalizations of the same UNIX system calls.

3.3.1 Open Page File

`(int PagFilId)` $\leftarrow$ `pfOpen(char* filename, int flags, int mode, int AlcTyp, int PagSiz)`

Opens a Page File and returns a Page File id (positive int). The `filename` is a string and can be qualified with sub directories below the base directory. `PagSiz` is the size of the data page which must match the page size given on the call to `pfOpen` when the Page File was created. The allocation type (`AlcTyp`) designates how the pages are allocated. Possible values are:

`PF_LOCAL`	Page File is allocated to the local disk.
`PF_RROBIN`	Pages are allocated to remote disks by round robin.
`PF_ADAPTIVE`	Pages are allocated adaptively, currently set to `PF_LOCAL`.

The `flags` and `mode` parameters are used in the same way as the UNIX system call `open()`. The `flags` parameter designate how the file is opened and if the file is to be created. The `mode` parameter is only evaluated if the file is created and designates the authorities of the new file. The possible values for `flags` are:

`PF_RDONLY`	Page File opened for read access.
`PF_WRONLY`	Page File opened with write access.
`PF_RDWR`	Page File opened with read and write access.
`PF_CREAT`	Page File is created.

If this function successfully opens the Page File it returns a positive `int` which is a unique identifier for this open Page File. The same page file can be opened multiple times and a new file identifier will be returned. If an error occurs during the open a negative value is returned in `PagFilId` which matches `PF_ERROR` and the error code can be found in `pfError`. The possible Page File errors are as follows:

`PF_MAXOPEN`	Maximum number of Page File already open.
`PF_BADALCTYPE`	Invalid `AlcTyp` passed to open.
`PF_MAXREQUEST`	The Page File system is out of request ids.

If a UNIX error occurs a negative value is returned, which matches `PF_ERROR`. The error code can be found in `pfError`. Some of the possible UNIX file errors are as follows:

`[EACCES]`	Error in file or directory permissions.
`[EDQUOT]`	Disk quota error.
`[ENOENT]`	File does not exist and PF_CREAT not specified.

3.3.2 Close Page File

`(int RtnVal) ← pfClose(int PagFilId, int WaitFlg)`

Closes a Page File and cancels any outstanding read requests. The `PagFilId` must be the id of file opened with the `pfOpen` function. This close operation reduces the number open files and allows the file identifier to be reused. This function can wait for confirmation or not. If `WaitFlg` is set to `PF_WAIT` then control will not be returned until the close has been confirmed by all servers. If `WaitFlg` is set to `PF_NOWAIT` then control is returned immediately and the close proceeds without confirmation. If this call completes successfully a positive value is returned which matches `PF_SUCCESS`. If an error occurs a negative value is returned, which matches one of the following errors:

`PF_BADFILE`	An invalid Page File id was passed on the call.
`PF_MAXREQUEST`	The Page File system is out of request ids.
`PF_RQSCNL`	Unconfirmed requests were canceled.

If a bad file id is given no file is closed. If open requests exist for the file all requests are canceled, the file close proceeds and `PF_RQSCNL` is returned. If blocking is requested, each server responds before the function returns.

3.3.3 Drop Page File

`(int RtnVal) ← pfDrop(char* filename, int AlcTyp)`

Drops (unlinks) any Page File. This removes a Page File from all servers on which it is stored. The allocation type (`AlcTyp`) designates how the pages in the Page File to be deleted are allocated. Possible values are:

`PF_LOCAL`	Page File is allocated to the local disk.
`PF_RROBIN`	Pages are allocated to remote disks by round robin.
`PF_ADAPTIVE`	Pages are allocated adaptively.

If the Page File is successfully dropped a positive value is returned which matches `PF_SUCCESS`. If an error occurs a negative value is returned from which matches `PF_ERROR` and `PF_pfError` matches one of the following:

`PF_MAXREQUEST`	The Page File system is out of request ids.
`[ENOTDIR]`	Path contains invalid directory.
`[ENOENT]`	Invalid file name.
`[EACCES]`	Error in file or directory permissions.

3.3.4 Lock Page File

`(int RtnVal) ← pfLock(int PagFilId, int LckTyp)`

Places a UNIX lock on an open Page File. The `PagFilId` must be the result of and `pfOpen` operation. If this page is file is allocated to remote disks each remote file is locked. All locks are non-blocking and if the lock cannot be achieved an error is returned. The type of lock is designated by `LckTyp` and the possible values are:

`PF_SHARE`	Shared file lock.
`PF_EXCL`	Exclusive file lock.
`PF_UNLOCK`	Release lock

If the Page File is successfully locked a positive value is returned which matches `PF_SUCCESS`. If an error occurs a negative value is returned which matches `PF_ERROR` and one of the following error conditions:

`PF_BADFILE`	An invalid Page File id was passed on the call.
`PF_MAXREQUEST`	The Page File system is out of request ids.

If the lock cannot be made without blocking a negative value is returned from which matches `PF_ERROR` and the error code in `pfError` is as follows:

`[EWOULDBLOCK]`	The lock cannot be achieved without blocking.

3.3.5 Access Page File

`(int RtnVal)` ← `pfAccess(char* filename, int AlcTyp, int AccTyp)`

This function returns the accessibility of a Page File. This is used to check for the existence of a file or to see if the file can be read, written to or exclusively locked. It can be used before a lock is requested to indicate if access to the Page Files is possible. The Page File name can be qualified with subdirectories below the base directory. The allocation type (`AlcTyp`) designates how the pages of the file are allocated. Possible values are:

`PF_LOCAL`	Page File is allocated to the local disk.
`PF_RROBIN`	Pages are allocated to remote disks by round robin.
`PF_ADAPTIVE`	Pages are allocated adaptively.

The type of access needed can be specified in `AccTyp` and the possibilities are:

`PF_READ`	The Page File can be opened for reading.
`PF_WRITE`	The Page File can be opened for writing.
`PF_LOCKEX`	An exclusive lock can be made.
`PF_EXIST`	Does the named file exist.

If the Page File can be successfully accessed in the specified way, a positive value is returned which matches `PF_SUCCESS`. If access is not possible a negative value is returned which matches `PF_ERROR` and one of the following error conditions:

`PF_BADALCTYPE`	An invalid `AlcTyp` was passed on the call.
`PF_MAXREQUEST`	The Page File system is out of request ids.

If a UNIX error occurs a negative value is returned from which matches `PF_ERROR` and the error code in `pfError` is positive and could be one of the following follows:

`[EACCES]`	Permission bits do not allow access to some part of Path.
`[ENOTDIR]`	Path contains invalid directory.
`[ENOENT]`	Invalid file name.

3.3.6 Status of Page File

`(int RtnVal)` ← `pfStatus(PagFilId)`

This function returns a pointer to a structure which contains the status of an open Page File. This information includes the number of unconfirmed reads and writes.

3.4 Page level methods

The page level methods provide a logical departure from the existing file access functions in three ways. First, reading and writing data is done one page at a time. Second, reads and writes are not atomic, they are split into a request and a confirmation. Finally, the read and write request calls do not block and the confirm blocks only if Polylith is run with the direct connect option.

In order to retrieve a data page, memory must be allocated for the page and a request for that page made. When the page is needed the `pfConfirmRead` function is called and the status of the page retrieval is returned. The page has either been written to the memory location provided or the request is still being serviced. This separation allows multiple pages to be requested and retrieved in parallel while the application is processing those pages which have been returned.

Writing a page is similar. The write is requested and is non-blocking. Processing can continue until the confirmation of the page actually being written is required. At this time the `pfConfirmWrite` can be called until it returns a successful result.

These routines are valid for local or remotely allocated Page Files. If the pages are allocated locally only one call will be needed to the confirm functions. This will always return successfully and the page will be written to or read from disk.

3.4.1 Request Page Read

`(int RqsId) ← pfRequestRead(int PagFilId, int PageId, void* BuffPtr)`

Initiate page retrieval. The page is identified with a number `PageId` which must be an existing page which must be less or equal to the last page in the file and greater than 0. The Page File id `PagFilId` must be a valid open Page File. The memory which into which the page will be written is pointed to by `BuffPtr`. Memory must be available to accommodate an entire page of data for the Page File identified by `PagFilId`. If the request is successful a positive request id is returned. This is used to check the status of the read request or to cancel the read request. If an error occurs a negative value is returned matching `PF_ERROR` and one of the following negative error codes can be found in `pfError`.

`PF_MAXREQUEST`	The maximum number of requests have been made.
`PF_BADFILE`	The Page File id is invalid.
`PF_BADPAGE`	The page number is invalid.
`PF_BUFFERERR`	Error occurred during remote buffer allocation.

3.4.2 Confirm Read Request

`(int RtnVal) ← pfConfirmRead(int RqsId)`

Return the status of a page request. `RqsId` must be a open read request. If the page has been copied into the memory location this function returns a positive value equal to `PF_SUCCESS`, read request is complete and the request id will be reused. If the page has not yet been received a positive value matching `PF_WAITING` will be returned. If an error occurs a negative value is returned matching `PF_ERROR` will be returned and the value of `pfError` will match one of the following:

`PF_BADREQUEST`	The request id in invalid.
`PF_RQSCANCEL`	The read request has been canceled.
`PF_NOTREADRQS`	The request id is not a read request.

3.4.3 Cancel Read Request

`(int RtnVal) ← pfCancelRead(int RqsId)`

Cancel a request to read a page. If several pages were requested and they are no longer needed this function will free the request and allow the memory allocated for the page to be reused. If the request is successfully canceled a positive value equal to PF_SUCCESS is returned and the request id may be reused. If an error occurs a negative value is returned matching PF_ERROR is returned and pfError will match one of the following:

PF_BADREQUEST	The request id in invalid.
PF_RQSCANCEL	The read request has been canceled.
PF_NOTREADRQS	The request id is not a read request.

3.4.4 Request Page Write

`(int RqsId) ← pfRequestWrite(int PagFilId, int PageId, void* BuffPtr)`

Request a page be written to disk. The data to be written is located in the memory location pointed to by BuffPtr and extends for a length of the page. The page id PageId must be greater or equal to 1 and less than or equal to the last page plus 1. This page write request cannot be canceled. If this function completes successfully a positive request id is returned which can be used to check on the status of the write request. If an error occurs a negative value is returned matching PF_ERROR and one of the following negative error code can be found in pfError.

PF_MAXREQUEST	The maximum number of requests have been made.
PF_BADFILE	The Page File id is invalid.
PF_BADPAGE	The page number is invalid.
PF_BUFFERERR	Error occurred during remote buffer allocation.

3.4.5 Confirm Write

`(int RtnVal) ← pfConfirmWrite(int RqsId)`

Return the status of a page write request. RqsId must be a open write request. If the page has been written this function returns a positive value equal to PF_SUCCESS, the write request is complete and the request id will be reused. If the page has not yet been written a positive value matching PF_WAITING will be returned. If an error occurs a negative value is returned matching PF_ERROR will be returned and the value of pfError will match one of the following:

PF_BADREQUEST	The request id in invalid.
PF_RQSCANCEL	The read request has been canceled.
PF_NOTWRITERQS	The request id is not a write request.

It is possible for a time out error to occur after the page has been successfully written by the server but before the confirmation is sent to the Page File client and therefore returned to the application. Therefore the application should not assume that page has or has not been written if a time out occurs.

BIBLIOGRAPHY

[PuJa91] An environment for developing fault tolerant software. J. Purtilo and P. Jalote. **IEEE Transactions on Software Engineering**, vol. 17, (1991), pp. 153-159.

[Purt9X] The Polylith Software Bus. J. Purtilo. To appear, **ACM Transactions on Programming Languages and Systems**.

[Rous9X] ADMS: A Testbed for Incremental Access Methods, N. Roussopoulos, N. Economou, and A. Stamenas, To appear, **IEEE Trans. on Knowledge and Data Engineering**.

[Rous92] The Tower of Pizzas, N. Roussopoulos, **Internal Memo**.

SUMMARY OF METHODS

- Start Page File processing
  `(int RtnVal)` ← `pfStart(int* argc, char*** argv, int NbrFil, int NbrRqs)`

- Terminate Page File type
  `(int RtnVal)` ← `pfTerminate()`

- Open Page File
  `(int PagFilId)` ← `pfOpen(char* filename, int flags, int mode, int AlcTyp, int PagSiz)`

- Close Page File
  `(int RtnVal)` ← `pfClose(int PagFilId, int WaitFlg)`

- Drop Page File
  `(int RtnVal)` ← `pfDrop(char* filename, int AlcTyp)`

- Lock Page File
  `(int RtnVal)` ← `pfLock(int PagFilId, int LckTyp)`

- Access Page File
  `(int RtnVal)` ← `pfAccess(char* filename, int AlcTyp, int AccTyp)`

- Status of Page File
  `(int RtnVal)` ← `pfStatus(PagFilId)`

- Request Page Read
  `(int RqsId)` ← `pfRequestRead(int PagFilId, int PageId, void* BuffPtr)`

- Confirm Read Request
  `(int RtnVal)` ← `pfConfirmRead(int RqsId)`

- Cancel Read Request
  `(int RtnVal)` ← `pfCancelRead(int RqsId)`

- Request Page Write
  `(int RqsId)` ← `pfRequestWrite(int PagFilId, int PageId, void* BuffPtr)`

- Confirm Write
  `(int RtnVal)` ← `pfConfirmWrite(int RqsId)`