BUFFER STORAGE

A *buffer* is a contiguous piece of memory address space used to transfer data between different execution threads of a computer system. The threads might be executions of user programs (e.g., cooperative parallelism), or the threads might be executing on behalf of a system service (e.g., the software component of a storage system). Three general domains of buffer use are (1) communication between processes (e.g., a browser downloading a web page), (2) data transfer between levels of the memory hierarchy (e.g., reading a file), and (3) database processing.

In many cases, when data are communicated between execution threads, the data pass through protection boundaries. For example, a client process may request service from a server process or from the operating system kernel. The communication buffer limits size and location of the memory space that the communicating threads will share. Typically, memory protection is ensured by the operating system's kernel, explicitly copying the buffer from the sender to receiver, or by limited-memory space sharing implemented through virtual memory.

Effective use of buffering greatly increases the efficiency of a computer system. Transmitting memory between processes generally incurs significant central processing unit (CPU) and memory overhead for various types of bookkeeping. Transmitting large blocks of data amortizes the fixed bookkeeping costs across all the bytes in the buffer. Buffering can also transform synchronous communication into asynchronous communication. Also, computer systems often encounter "bursty" behavior (i.e., a brief but intense period of resource use). For example, a process might write a large block of data to a file. While the file system transfers the data to the file, the process can compute the next block of data. The use of buffers enables *caching,* or storing popular data high in the memory hierarchy to save on expensive data-fetch operations.

APPLICATIONS

Buffer storage and management is used in many disparate areas of a computer system. We discuss issues related to three main application areas: communications, memory hierarchy, and database management.

Communications

In many applications, one thread of execution needs to transmit a stream of data to another thread of execution. Examples include network communications (e.g., communications using TCP/IP) and cooperative parallelism (e.g., processes obtaining work from a work queue). The primary issues related to buffer management for communication are transferring data between memory spaces and using buffers to match data production rates with data consumption rates.

Separate Memory Space Domains. In some applications, the communicating execution threads share the same memory space. For example, one thread might fetch data from a data source into a buffer, and the other thread might process the data in the buffer. Because the threads share their memory space, no data copying is required for the sharing. In other applications, the threads do not share a memory space. For example, if the operating system does not support multiple threads of execution in the same address space, the example of asynchronous input/output (I/O) must be implemented with two separate processes. In another common case, one of the communicating processes is the operating system kernel.

There are three primary methods for communicating buffers between processes with separate memory spaces. In the first method, the operating system kernel copies the buffer from the sender to the receiver. This method is safe, easy to use, and general purpose, but incurs a memory copying overhead that can be substantial in data-intensive applications (e.g., file servers).

The second method uses the virtual memory system to share a buffer. The communicating processes ask the virtual memory system to map portions of their virtual address space to the same physical memory block. Data are transferred from the sender to the receiver by writing to and reading from the shared memory block. This method of communication avoids a data copy. However, the virtual memory system places restrictions on the possible sizes of the shared memory block, and the allocation of the shared block might be cumbersome. In addition, the communicating processes must follow a protocol that restricts access to the shared memory block, or else the sender might overwrite a message before the receiver has finished reading it.

The third method for communicating data between execution threads also uses the virtual memory system. When the sender sends a message to the receiver, it tells the kernel the address range to be transferred. Instead of copying the data in the address range, the kernel maps an equal-size address range in the receiver process to the physical memory pages in the sender. To ensure safety, if the sender writes to a virtual memory page shared in this manner, the kernel will first copy the physical memory page and remap the receiver's virtual memory to point to the copied page (this is *copy on write*). If the receiver cannot access the sender's physical memory (e.g., they reside on different computers), then whenever the receiver reads a first byte from a shared page, the kernel copies the page from the sender into the local physical memory space (*copy on read*). This method of communication can substantially reduce the amount of copying. Because the sender cannot overwrite the message it sends, the sender and receiver do not need to transfer data to and from a shared buffer. If the message is a large structured data object (e.g., a matrix or a database), the receiver might read only a small part of the message and will only perform a small physical transfer.

Asynchronous Data Transfer. A great improvement in throughput can often be obtained by substituting asynchronous I/O for synchronous I/O. Let us consider an example of a process that transfers data from a disk-resident file to the network. If the process uses synchronous I/O, it requests the data from the disk drive, then transfers the data to the network. With asynchronous I/O, the process reads a block of data from the disk drive and simultaneously writes the previous block to the network. The time to do one block transfer with synchronous I/O is the sum of the disk-drive service time and the network service time. Using asynchronous I/O, the total time reduces to the maximum of these two service times. Thus, asynchronous I/O usually improves transfer rates significantly.

This example is a motivation for the *bounded buffer* problem. To implement asynchronous I/O, we use two threads. One thread reads a block of data from the disk drive and puts the block into a shared buffer. The second thread gets a block from the buffer and gives it to the network. Since one of the devices is likely to be faster than the other, we need to synchronize the two threads. The sender should wait until there is room in the buffer before storing its block, and the receiver should wait until there is a block of data in the buffer before retrieving one. Operating systems typically provide support for writing an object with the following interface:

`bounded_buffer(N,K)`	Create a buffer with N blocks of data, each K bytes in size, initially all empty.
`void put_block(char *dat)`	Wait until there is an empty block in the buffer, then transfer the data pointed to by `dat` into the empty block, making it full.
`void get_block(char *dat)`	Wait until there is a full block in the buffer, then transfer its contents to the memory location pointed to by `dat`, and make the buffer empty.

A program built using this interface can implement asynchronous I/O as long as $N \geq 2$. However, there is a great deal of unnecessary copying. The solution that is preferred in practice returns pointers to data blocks instead of the data blocks themselves. Data-block consumption works in both directions. The sender consumes empty blocks and produces full blocks, while the receiver consumes full blocks and produces empty blocks. The buffer maintains separate lists of full and empty blocks. Because of the bidirectional flow of buffer blocks, this type of buffering is referred to as *double buffering*. The interface for a double buffer is:

```
double_buffer(N,K)
char *get_empty_block()
char *get_full_block()
void put_empty_block(char *dat)
void put_full_block(char *dat)
```

A double-buffering program would be implemented as follows:

```
double_buffer dbuf(N,K)

producer()                  consumer()
  char *dat                   char *dat
while(1)                    while(1)
  dat=                        dat=get_full_block()
    dbuf.get_
    empty_block()
  read_from_disk(dat)         write_to_net(dat)
  put_full_block(dat)         put_empty_block(dat)
```

In some applications (e.g., compressed multimedia) the sender or the receiver might have significant variation in its cycle time. These variations in speed can be smoothed out by using large numbers of blocks but at the cost of significantly more storage overhead.

Memory Hierarchies

Buffers are commonly used to communicate data between memory hierarchies. *Primary storage* is fast silicon memory. This type of memory is expensive and *volatile* (the contents are erased when power is turned off). Primary storage can have many levels of hierarchy also (registers, on-chip cache, off-chip cache, local memory, remote memory, etc.), but we do not explore these issues here. *Secondary storage* is slow, but large, inexpensive, and nonvolatile. Secondary storage is usually implemented with magnetic disk drives, but other media can be used (nonvolatile silicon storage, optical disks, tapes, etc.). A *tertiary storage* system is composed of a robot arm that can serve removable media (e.g., tapes) to read/write drives. Tertiary storage has long file access latencies (10 s to 1000 s), but is 1 to 2 orders of magnitude less expensive than secondary disk storage.

Buffer space is used to provide transparent access to secondary and tertiary storage, and to improve performance. Typically, the global buffer space is divided into buffer blocks. We discuss three examples of access to a memory hierarchy and the issues related to each type of access.

File Systems. In general usage, a file is an ordered collection of uninterpreted bits residing on secondary storage. One must *open* a file before accessing the data in it. Typically, a file system provides a hierarchical *directory* for convenient selection of the desired file. Once a file is opened, one can read from or write to arbitrary positions in the file. If one writes past the end of the file, the file length is automatically extended. Modern operating systems typically provide additional operations on their file systems, but such a discussion is beyond the scope of the present article.

To read (write) a block of data, one typically issues a positioning system call to indicate the desired read (write) location (a *seek* command), and then one issues a read (write) system call with a pointer to the data buffer. The mechanisms of the buffer data transfer have been discussed earlier. The main issues with file systems relate to caching file blocks and storing the file on secondary storage so that it can later be read again.

Typical data access patterns on a file system show a great deal of locality. For example, at any given time one typically works on a small collection of the documents that are stored in the file system. Some files (popular programs, for example) are frequently read. These are examples of *temporal locality*—If you access a data block now, you are likely to do so again in the near future. Files are usually read from beginning to end in sequential order. Therefore they exhibit a great deal of *spatial* locality—if you read (write) a block now, you are likely to read (write) the next block in the file in the near future. We can greatly improve the performance of the file system by taking advantage of these types of locality.

A *cache* is a temporary copy of data that resides on slower memory devices. If a user program requests to read a data block which is cached, a *cache hit* occurs (respectively, a *cache miss*). On a cache hit, the operating system can return the cached data instead of making a slow and expensive request to secondary storage. Caching can also speed up user level write requests. Instead of writing the data block immediately, the operating system returns control to the user program immediately and schedules the physical write to secondary storage to occur later (*write-back* caching). This policy implements a form of asynchronous I/O. Write-back caching can also help improve file layout, as we discuss later. When the cache is full, some of the blocks in the cache must be removed to make space for the newly referenced blocks. We discuss *cache-replacement* policies in more detail later.

If a user program requests to read a file block now, it is likely to request the next file block in the near future because of spatial locality. By *prefetching* the next few file blocks and transferring them into the cache, we can exploit spatial locality. If the file block prefetch is performed asynchronously, the user program will not experience delays when it reads the prefetched blocks. In addition, it is likely that the prefetched blocks are physically close to the requested block and therefore are inexpensive to read immediately after reading the requested block. Concurrent requests for data from the magnetic disk might make access to the next file block slow when the user program actually makes the request.

The way that a file is written to typical nonvolatile memory devices such as magnetic disk drives can affect the speed with which the file can be accessed again. Magnetic disk drives have moving parts, so a file layout that minimizes aggregate disk drive movement allows the file to be read faster than a layout that does not minimize movement. In addition, information about the file (the *metadata*), including its physical locations on secondary storage, must also reside on secondary storage. We discuss these issues in more detail later.

Virtual Memory. Most modern operating systems provide *virtual memory*. With virtual memory, a process' address space is not mapped directly to any physical address space. Instead the logical memory addresses accessed by a process are translated into physical addresses by a combination of memory management hardware and operating system software. The unit of address translation is typically a fixed size block called a *page*. The address translation mechanism can indicate that a page does reside on main memory, but instead resides on slower secondary memory. If a process references a memory location on a page that is not resident on main memory, a *page fault* occurs. The process is suspended, the page is fetched from secondary memory (the *backing store*), and then the process is resumed. The backing store is typically one or more files managed by the file system.

The details of virtual memory management are far beyond the scope of this article. However, we discuss it because of its close relationship to buffer management. We have already seen that virtual memory can be used as a mechanism for buffer communication. In many operating systems, virtual memory is used for file access by opening a *memory mapped file*. The backing store for a portion of a process' address space is defined to be the memory mapped file. Read and write access is performed by reads and writes to logical memory.

Another connection between virtual memory and file buffer management is the treatment of caching. The main-memory-resident portion of a processes virtual memory space is equivalent to a file system cache. Similar cache-management considerations apply. One significant difference between file cache management and virtual memory management, however, is the mechanism by which data blocks are requested. A request for a file block is made explicitly through a system call, and the operating system can spend nontrivial resources recording cache hits. A request for data on a virtual memory page is made at the level of the processor's instructions. To avoid slowing down the processor, only very simple operations can be made to record cache hits. The typical mechanism is to associate a *reference bit* with each page. On any access to data in the page, the reference bit is set. The rate at which the page is accessed can be determined by periodically clearing the reference bit and measuring the time until the reference bit is set again. We will discuss virtual memory management algorithms in more detail later. In addition to the reference bit, virtual memory management hardware typically provides a *dirty bit,* which is set whenever a memory location in a page is written to.

Explicit Buffer Access. In some application areas, it is preferable to use explicit buffer management. A typical example is an index structure that is designed for efficient search while only a portion of the index resides on main memory. More generally, database applications often bypass file system buffering and perform their own.

An application that performs explicit buffer management typically makes use of a *buffer manager.* A simple interface for a buffer manager is the following:

`buffer_manager(N,K)`	Create a buffer with N blocks of data, each K bytes in size, initially all unpinned.
`char *get_block(int address)`	Load K bytes of data from the location indicated by `address` on secondary storage into an unpinned buffer. Pin the buffer and return its address.
`void unpin(char *dat)`	Unpin the block pointed to by `dat`.

A block of data that is fetched into the buffer is *pinned* into the buffer. No pinned block will be overwritten, so it is safe for the application to access the pinned block directly. When the application is no longer needed, it is unpinned. An unpinned block retains its association with the data at location `address` to take advantage of possible future cache hits. However, the block becomes a candidate for replacement.

The user application might know of a good strategy for performing block replacement. For example, in an index structure an unpinned leaf node is a good candidate for replacement. These issues will be discussed in more detail later.

The explicit buffer management might be hidden from the application programmer by using a *persistent object store.* Persistent objects are data objects (e.g., C++ objects) that reside on secondary storage and can be accessed in main memory through an implicit cache. A common method of access is *pointer swizzling,* which uses virtual memory to detect a reference to a noncached persistent object (1). Another common access method is through object container classes.

Database Systems

Buffer management for database applications has received intense scrutiny in the research community. Part of this effort relates to special buffer allocation and block replacement strategies for database systems. A typical database application answers queries through a composition of database operators (i.e., select, project, join, etc.). These operators are highly tuned, and a great deal is known about their data access patterns. Inexpensive performance improvements are possible by using explicit buffer management with a tuned buffer allocation policy. However, the same is true for other applications (e.g., scientific computing) so we will not address the issue further here.

Database systems typically implement a type of formal data sharing between concurrently executing programs. A *transaction* is a program that has atomicity, consistency, isolation, and durability (ACID) properties. The start of a transaction is indicated by a `begin_transaction` statement and is ended by a `end_transaction` statement. *Isolation* means that the execution of a transaction is not affected by concurrently executing transactions. *Atomicity* means that either the transaction executes to completion, or all traces of a partial execution are removed. *Durability* means that the effects of a completed transaction are not undone by system failures (though they can be undone or modified by subsequent transactions). *Consistency* means that a transaction takes a database from one consistent state to another consistent state.

The theory of transaction processing is extensive, and more than a cursory discussion is beyond the scope of this article. However, we note that guaranteeing ACID properties often reduces to a matter of deciding when a transaction is allowed to access a buffer and guaranteeing that modified buffers are written to stable storage before the transaction is allowed to complete. We return to this topic in more detail later.

BUFFER ALLOCATION POLICIES

The use of caching is of great benefit in making computer systems more efficient. When properly applied, a program can access a vast (but slow) memory at a speed close to that of a fast (but small) memory. In this section, we discuss cache-management policies and more generally buffer management policies.

There are several dimensions to the choice of a buffer allocation policy. One dimension is to look at local versus global allocation policies. A local allocation policy assumes a fixed-size buffer, and makes decisions about which block to replace on a cache miss. A global allocation policy allocates buffer space to a set of concurrently executing processes (each of which can make local allocation decisions). Some buffer allocation policies make a clear distinction between global and local allocation, while some let the concurrent processes compete for blocks in a global pool.

A second dimension is whether the blocks are of fixed or variable size. Fixed-size block management is more efficient than variable-size block management in all but a few special cases. Most of this discussion assumes fixed-size blocks, with a special section on variable-size block management. A third dimension is whether cache hits are explicitly (a system or procedure call is required to access the block) or implicitly recorded (i.e., virtual memory).

Local Allocation; Fixed-Size Blocks; Explicit Cache Hits

The most common local replacement algorithm for fixed-size buffers with explicit knowledge of cache misses is the *least recently used* (LRU) algorithm. If a cache miss occurs, then the unpinned block that holds the data requested furthest in the past is chosen for replacement (i.e., the least recently used block).

LRU has a simple implementation. Each main-memory cache block has a forwards and backwards pointer. The pointers are used to implement a doubly linked list, with the head of the list being the most recently requested block and the tail of the list being the least recently requested block (pinned blocks, if any, are managed separately). On a cache miss, the block at the tail of the list is removed from the list, the association of the block with its current contents is broken, the new data are loaded into the block (and an association of the data with the block is made), and the block is placed on the head of the list. On a cache hit, the referenced block is removed from the list and then placed at the head of the list. The forwards and backwards pointers associated with each cache block ensure that only a few instructions are required to manipulate the list.

Although it is a very simple algorithm, LRU has many nice properties. First, it is fast and easy to implement. Second, it has good theoretical optimality properties. LRU is an optimally *competitive* algorithm. Roughly, this means that LRU will not have significantly worse performance than other cache replacement algorithms even on worst-case patterns of data access. In addition, increasing the space allocated to a cache that is managed by LRU is guaranteed to reduce the cache-miss rate. Some cache-replacement algorithms, such as first in—first out (FIFO) do not have this property (this is known as *Belady's anomaly*).

Third, LRU tends to have good performance in practice. To see why, consider a common model of data reference behavior, the *independent reference model* (IRM). IRM assumes that every data item D_i that can be accessed has an associated probability of being referenced p_i. On each data access, the probability of referencing D_i is p_i, independent of all preceding data accesses. Although this model is simple, it approximates actual data access behavior to a reasonable degree. If the data reference pattern is IRM, then the optimal cache-replacement algorithm (called *OPT* or A^*) for N blocks is to pin in the cache the $N - 1$ data items with the highest probability of reference and use the remaining block to handle cache misses.

The OPT algorithm cannot be used in practice because the probabilities of reference are not known a priori. However, the LRU algorithm is a maximum likelihood estimator (MLE) of the OPT algorithm given that the only information that is available is the time since the last reference. In addition, LRU often performs well when IRM is violated. For example, the data item reference probabilities often change because of temporal locality. Because LRU stored only a small amount of information, it is able to react to changes in temporal locality quickly.

Statistical Estimation Methods. The performance of the LRU cache-replacement algorithm can be improved by using more information to estimate the reference rate of a data item (2–4). The LRU/2 algorithm records the time of the last and the penultimate reference to a data item. The data item whose penultimate reference is furthest in the past is chosen for replacement.

Unfortunately, the LRU/2 algorithm is expensive to implement. Ordering on the time of the penultimate reference cannot be implemented with simple list operations. Instead a priority queue must be used, which is complex to implement and is (relatively) expensive to manipulate.

A fast approximation to the LRU/2 algorithm is the 2Q algorithm. The 2Q algorithm uses two lists ("two queues") to manage the cached blocks. The first list is a *probation queue,* and the second list is the main cache. The main cache has a desired maximum size. When a cache miss occurs, a block is removed from the tail of the main cache if the main cache exceeds its maximum size or otherwise from the tail of the probation queue. The block is loaded with the referenced data and is placed at the head of the probation queue. When a cache hit occurs and the block is in the probation queue, the block is placed at the head of the main cache. When a cache hit occurs and the block is in the main cache, the block is removed from its position in the list and placed at the head of the main cache. The effect is to place only "hot" data items into the main cache, where hot data items are detected by a short period between successive references.

To make the statistical estimation algorithms work well, some additional complications must be introduced. Temporal locality is particularly strong immediately after a data item has been referenced. Even cold data items are likely to be referenced again shortly after an initial reference. To filter out these *correlated references,* statistical estimation methods do not count repeat references to a data item during the *correlated-reference period* after a cache miss. In the 2Q algorithm, the correlated-reference period is implemented by placing a *correlated-reference queue* in front of the probation queue. On a cache miss, the block with the referenced data item is placed at the head of the correlated-reference queue, and the block at the tail of the correlated-reference queue is placed at the head of the probation queue. Cache hits on the block in the correlated-reference queue have no effect.

Special Purpose Algorithms. In some special applications, the user might know enough about the data access pattern to be able to use a special cache-replacement algorithm and reduce the cache-miss rate. For example, a common access pattern is to scan a large database repeatedly. If the database is larger than the cache, then replacing the most recently used (MRU) cache block minimizes the cache-miss rate. Special cache-replacement algorithms can also be used for index structures. However, both the LRU/2 and the 2Q cache-replacement algorithms have near-optimal performance on both scan and index structure data reference patterns.

Handling "Dirty" Blocks. If a process updates a cached block, the block becomes "dirty". Dirty blocks are more expensive to replace from the cache than clean blocks, because their contents must be written back to secondary storage. Many cache-management systems perform *data cleaning* periodically to avoid choosing a dirty block for replacement. If a dirty block is chosen, it can be cleaned immediately, or the cleaning can be deferred and another block chosen for replacement.

Local Allocation; Fixed-Size Blocks; Implicit Cache Hits

In virtual memory systems, cache hits are not explicitly registered so a LRU-type algorithm cannot be implemented. Most memory management hardware supports reference bits, so the time between references to the page can be approximated.

A large class of page-replacement algorithms are known as *clock* algorithms. With these algorithms, the metadata for the virtual memory pages are placed in an array. A clock algorithm keeps a pointer to the last page examined during the last page replacement invocation. On a page fault, the clock algorithm scans the virtual memory pages, starting where it left off on the last invocation. If the pointer reaches the end of the array, it starts again at the beginning of the array. The motion of the pointer through the virtual memory pages can be visualized as the hand of a clock scanning through the hours of a day. When enough pages have been selected for replacement (perhaps just 1), the clock algorithm stops.

The *second-chance* algorithm is the simplest nontrivial clock algorithm. When the second-chance algorithm examines a page, it looks at the reference bit of the page. If the reference bit is set, the second-chance algorithm clears the bit and proceeds to the next page. If the reference bit is not set, the page is chosen for replacement. Frequently referenced pages will tend to have their reference bits set, while infrequently referenced pages will not.

The second-chance algorithm can be extended with r history bits. When a page is examined, the reference bit is shifted into the history bits, then reset. If all of the history bits are zero, the page is chosen for replacement.

Virtual memory managers typically use a *free pool* or a set of pages already selected for replacement. On a page fault, a page in the free pool is chosen to store the referenced data. A *cleaning daemon* is periodically executed to run a clock algorithm until the free pool is sufficiently large. Writes are scheduled for the dirty pages that are added to the free pool.

Global Buffer Management

A global buffer management strategy can let all concurrent processes compete for the same set of buffers or can partition the buffer space among the processes. If one expects that different processes will tend to access different data, then partitioning is likely to be the best strategy. Otherwise, competition is a better strategy. For example, file cache tends to be equally accessed by all processes, while (nonshared) virtual memory tends to be allocated.

If the buffer space is not partitioned, then the global buffer allocation strategy is to have each process execute a local replacement algorithm on the global set of buffers. If the buffer space is partitioned, then each process executes a local replacement algorithm on its allocated buffer space. The final problem is to determine an optimal allocation of buffer space to processes.

Because the data access patterns of most processes exhibit temporal locality, a process will have a low page-fault rate as long as its *working set* is resides on main memory. The working set of a process is the collection of pages referenced by the process in the last T seconds. An optimal buffer allocation algorithm will allocate to each process the number of pages in its working set.

Implementing an optimal buffer allocation strategy is not feasible. Instead, the *buffer pressure* of each buffer partition can be measured and used to estimate the optimal allocation. If a process has a buffer that is too small, it will incur many misses. If the buffer is too large, it will incur few misses. The allocation can be balanced by taking pages from a process with a low miss rate and giving them to a process with a high miss rate. In clock algorithms, the imbalance in buffer allocation can be measured by observing the difference in the rates at which the clock scans through the virtual memory pages.

VARIABLE-SIZE BLOCKS

In some applications, variable-size buffers are preferable to fixed-size buffers. One example is tertiary storage management. To improve performance, a region of magnetic disk is used to cache tertiary storage-resident files that have been accessed. The data access, and thus the caching, is typically on a per-file basis. The files sizes can range through 6 orders of magnitude.

The wide variety of file sizes opens a new dimension in determining an optimal buffer allocation. A large file is expensive to store, but caching it can return a large benefit if it is frequently accessed. Reducing the miss rate requires that one cache the files whose per-byte amortized access rate is the highest.

An algorithm (called GOPT or ST-LRU) (5) for determining which cached file to replace is the following. For each cached file F_i, compute the weight of the file w_i to be the product of its size and the time of its last access. On a cache miss, choose for replacement the file with the largest weight until enough free space exists to load the requested file. The GOPT algorithm has been proved optimal under certain conditions.

The GOPT (or ST-LRU) algorithm is expensive to execute because each cached file must be evaluated on every cache miss. A less-expensive alternative is the ST-bin algorithm (6). Files are hashed into buckets based on their size, where each bucket represents a contiguous range of file sizes. The files in each bucket are organized into an LRU queue. On a cache miss, the weight of the file at the head of each bucket's queue is evaluated, and the heaviest file is chosen for replacement until enough space exists to load the requested file. The ST-bin algorithm has performance close to that of the ST-LRU algorithm.

BUFFER STORAGE

Data buffers are typically stored on secondary storage in *files* using a *file system.* A file system provides support of identifying the desired file through the use of *directories.* A directory entry for a file contains a file name and metadata about the file. Some of the metadata indicates the locations of the file on secondary storage. In this discussion, we assume that the secondary storage is a magnetic disk drive, and that the file system is similar to the well-documented Fast File System of 4.3 BSD UNIX. While other file systems differ in many details, the fundamental problems and solutions are similar.

Disk-Drive Performance Characteristics

Magnetic disk drives consist of one or more spinning stacked platters coated with magnetic media and a disk arm that positions a read/write head on each active surface of the platters. Each platter is divided into a number of tracks, and each track is divided into a number of sectors. A magnetic disk drive will transfer data (read or write) in the unit of a single sector. File systems often make their block sizes equal to an integral number of sector sizes.

The physical characteristics of a magnetic disk drive are the cause of its performance characteristics. Disk sectors tend to be large. So while a disk drive can transfer data at a high rate, it can read from or write to only a few sectors per second. Obtaining data from a magnetic disk drive requires mechanical movement. Accessing a sector on the same track incurs a *rotational latency* for the platter to spin and make the desired track come under the read/write head. Requests for sectors on the same track of a different platter incur a rotational latency and a head calibration delay. Accessing a sector on a different track requires the movement of the disk arm. Moving the disk arm a long distance requires considerably more time than moving it a short distance. However, moving the disk arm a single track requires a substantial fraction of the time to perform a full platter seek because of start up and calibration delays.

Obtaining high performance from magnetic disk drives requires a careful data layout that minimizes the movement of disk drive components to read a file. This requires that a file's metadata are stored close together, that the metadata are stored close to the data, and that contiguous data blocks are stored close together (because files are usually accessed sequentially).

Metadata

The secondary storage media typically contains the metadata, which allows data blocks to be stored and accessed. One part of the metadata are the directories which contain file names and pointers to the corresponding file metadata. By making directories a special type of file, directories can be hierarchical.

The metadata for a file contains information about the file, for example, the nature of the file, the creation time, and security information. A significant part of the file metadata is a mapping between addresses in the file space and physical sector locations on the disk drive that contain the corresponding data. Since this mapping might be very large, it is often stored separately from the directory. An I-node is a component in this mapping. The map from addresses to sectors might be laid out as a list, a bit map, or a hierarchy.

File systems support allocation and deallocation of file data. The *free list* is a secondary storage-resident description of the unallocated sectors remaining on secondary storage. The free list can have many representations, including lists of free regions and bitmaps. Allocating a block of secondary storage for a file entails finding an appropriate block on the free list, marking it allocated, then incorporating the block into the file's address map. Deallocating space (e.g., deleting the file) is accomplished by deleting blocks from the file's address map and returning them to the free list.

File Layout and Access

The efficiency of the file system can be improved by placing file data and metadata in spatially local areas. Given the online nature of file creation and deletion, optimal algorithms are difficult to obtain. However a number of heuristics have been developed:

- Attempt to locate file I-nodes near the file's directory, and the file's data near the file's I-nodes. Attempt to allocate contiguous regions of file address space as contiguous or nearby disk blocks.
- Use write-back caching to obtain a large sequential collection of file data blocks before secondary storage allocation is performed. Prefer to allocate secondary storage data blocks in units of multiple sectors.
- Create several zones on the disk, where each zone is a contiguous region of tracks. Prefer to allocate directories, I-nodes, and file data blocks all in the same zone.
- If write-back caching is used, sort the disk blocks to write by their location to minimize disk movement.
- Prefetch nearby file blocks when performing a file read (prefetching is often implemented by the disk drive controller also).

After a long series of file allocations and deallocations, file blocks of most files tend to be scattered across the disk and distant from the file metadata. Such a file system is *fragmented.* Most file systems provide *defragmentation* utilities to restore spatial locality to the files on the disk.

Crash Recovery

If the computer system fails while file update operations are pending, then the file system is likely to be inconsistent when the computer system is restarted. If write-back caching is used, then some user-level writes might not have been propagated to secondary storage. If the crash occurs while a file update is in progress, then the metadata might be inconsistent (e.g., a block might have been allocated to a file but not yet removed from the free list).

To detect possible inconsistencies, a file system can store a sequence number in two different disk blocks. Before any updates are performed on the file system, the sequence number in one of the blocks is updated. When the file system is shut down, all pending update operations are performed and then the sequence number in the other block is incremented. If the computer initializes the file system and discovers that the blocks contain different sequence numbers, the file system metadata might be in an inconsistent state. In this case the computer will attempt to repair the file system before allowing any further access. For example, the computer will ensure that each block on the disk drive is either on the free list or is allocated to exactly one file.

To reduce file system repair time, some file systems use a *log* of all updates. Before any write (to file data or metadata) is performed, a description of the action is first written to the log, and the update propagated to secondary storage. For recovery, the computer only needs to ensure that all updates described in the log have been propagated to secondary storage. *Log-structured* file systems perform updates (almost) entirely in the log and (mostly) avoid the additional step of writing data in the areas pointed to by the I-node.

Tertiary Storage

Because a tertiary storage system uses removable media, the metadata for a tertiary storage system is typically stored in a database on secondary storage. A tertiary storage system typically uses a portion of secondary storage to improve performance. A *staging* area is used to store files pending migration to tertiary storage. Staging serves the same purpose as a write-back cache for secondary storage. A cache area is used for files accessed by user applications (and serves the same purpose as a regular cache). Often the staging area and the cache area are merged and treated uniformly.

Cache and staging space is managed in largely the same way as for secondary storage. One exception is that variable-size block cache management is used. Another exception is that purging buffers from the (secondary-storage) cache is an expensive, and writes to tertiary storage must make use of spatial locality in writes whenever possible. Here, a *watermark* algorithm is often used. When the free space in the cache reaches a low watermark, files are selected for purging from the cache. Dirty or staged files are written to tertiary storage. The purging stops when the free space reaches a high watermark.

Tertiary storage media is often append-only (WORM optical disks or tapes). If a file is fetched, modified, and migrated back to tertiary storage, the old version cannot be overwritten. Instead it is marked as deleted. If an excessive amount of tertiary storage space is consumed by deleted files, a *compaction* process is run to reclaim the space.

TRANSACTIONS

A transaction is a program that has ACID properties. Implementing ACID properties imposes restrictions on when buffers can be read, when buffers can be written to secondary storage, and when a transaction can be allowed to commit. Transactions that access the same buffers must have their entire executions ordered with respect to each other, logically if not physically (i.e., the transaction executions must be *serialized*). For example, consider an execution in which transaction T_1 writes to buffer x before transaction T_2 reads x, and transaction T_2 writes to buffer y before transaction T_1 reads y. This execution violates atomicity; in user applications it can lead to difficult-to-trace program errors, such as money vanishing (or suddenly appearing) in bank accounts.

Ensuring atomicity and durability for a transaction that writes data is a seeming contradiction. Atomicity means that dirty buffers cannot be written to secondary storage before the transaction commit, but durability requires that they be written to secondary storage before the commit. The solution is to use a *log* file. If a dirty buffer is written to secondary storage before a commit point, a *log record* describing the previous contents of the disk block must first be written to the log (an *undo* log record). If a transaction has written to a buffer and its contents has not been written to secondary storage at the commit time, then a log record containing the dirty buffer must first be written to the log (a *redo* log record). When a transaction commits, a *commit record* is written to the log. If a failure occurs, a database recovery program scans the log and performs writes for all redo log records of committed transactions and for all undo records of uncommitted transactions.

If two transactions overlap in the data that they access, then a *commit* dependency can occur. For example, suppose that transaction T_1 writes to buffer x, and then transaction T_2 reads from buffer x and writes to buffer y. Suppose further that transaction T_2 commits while T_1 is still executing. If a failure occurs before T_1 can commit, or if the execution of T_1 is aborted (terminated before the commit for reasons of consistency, system resources, or user action), then consistency is violated. In user applications this problem can cause difficult-to-trace bugs. Therefore, transaction T_1 must commit before T_2 can commit, and a commit dependency exists between them.

A full discussion of transaction processing is beyond the scope of this article. However, we can outline a simple buffer management policy that ensures ACID properties:

1. Before a transaction T can access a data item, it must place a *lock* on the data item. If the data are locked by another transaction, T is blocked until the lock is released.
2. If a transaction updates a data buffer, the data buffer cannot be written to secondary storage until after the transaction commits.
3. A transaction cannot commit until a redo record is written to the log for every buffer it has written to.
4. A transaction commits by writing a commit record to the log. After the commit, the transaction releases all locks.

This transaction processing technique is *two-phase locking,* using exclusive locks and redo-only logging. Most database systems use more sophisticated techniques to improve concurrency, allow the execution of very large transactions, reduce logging overhead, and reduce database recovery time.

FURTHER INFORMATION

For more information on buffer management in centralized and distributed operating systems, see Refs. 7 and 8. An in-depth discussion of the implementation of the 4.4 BSD variant of UNIX is given in Ref. 9. An in-depth discussion of serializability theory is given in Ref. 10. The details of implementing a transaction processing system are discussed in Ref. 11.

BIBLIOGRAPHY

1. J. E. B. Moss, Working with persistent objects: To swizzle or not to swizzle, *IEEE Trans. Softw. Eng.,* **18**: 657–673, 1992.
2. E. J. O'Neil, P. E. O'Neil, and G. Weikum, The LRU-k page replacement algorithm for database disk buffering, *Proc. ACM SIGMOD Conf.,* Washington, DC, 1993, pp. 297–306.
3. J. T. Robinson and M. V. Devarakonda, Data cache management using frequency-based replacement, *Proc. ACM SIGMETRICS Conf.,* Boulder, CO, 1990, pp. 134–142.
4. T. Johnson and D. Shasha, 2Q: A low overhead high performance buffer management replacement algorithm, *Proc. Very Large Database Conf.,* Santiago, Chile, 1994, pp. 439–450.
5. P. J. Denning and D. R. Slutz, Generalized working sets for segment reference strings, *Commun. ACM,* **21**: 750–759, 1978.
6. T. Johnson, Analysis of the request patterns to the NSSDC online archive, *NASA Goddard Conf. Mass Storage Syst. Technol.,* pp. 225–240, 1995.
7. A. Silberschatz and P. Galvin, *Operating System Concepts,* Reading, MA: Addison-Wesley, 1994.
8. R. Chow and T. Johnson, *Distributed Operating Systems and Algorithms,* Reading, MA: Addison-Wesley, 1997.
9. M. K. McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System,* Reading, MA: Addison-Wesley, 1996.
10. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading, MA: Addison-Wesley, 1987.
11. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques,* San Francisco: Morgan Kaufmann, 1993.

THEODORE JOHNSON
AT&T Labs—Research

BUGS IN SOFTWARE. See SOFTWARE BUGS.

BULK ACOUSTIC WAVE SENSORS. See ULTRASONIC SENSORS.

BULK POWER SYSTEM RESTORATION. See POWER SYSTEM RESTORATION.

BURN-IN. See BURN-IN AND SCREENING.