Ext2 on Singularity

Scott Finley
CS 736 Semester Project Report – Spring 2008
University of Wisconsin – Madison

Abstract

Singularity is a new operating system produced by Microsoft Research. The design of Singularity is a marked departure from current commodity operating systems such as Windows or Linux.

This paper presents and implementation of the Linux ext2 file system for use on Singularity. This project allows the design of Singularity to be evaluated.

The development and testing described in this paper show that the Singularity design goals of high dependability and "good enough" performance are largely met. The protection provided by the micro kernel design prevented the Singularity kernel from crashing in the face of errors in the ext2 code. Performance testing showed that while not as fast as ext2 on Linux performance is not prohibitively low and the overhead of garbage collection is minimal.

1 Introduction

Singularity is a new operating system released by Microsoft Research. It is intended as a platform to investigate new ideas in operating system structure. Many of the design decision in Singularity represent a dramatic departure from main stream operating systems such as Windows or Linux [1].

Evaluating Singularity's design presents a challenge because existing applications and libraries must be re-written to be ported. One result of this issue is that FAT32 is the only file system that is supported. This leaves Singularity without a file system which can meet the IO demands of a benchmark such as SpecWeb99. Singularity provides better response time than Windows in this benchmark, but cannot match the total simultaneous users because of the bottleneck of the FAT file system[1].

This paper describes the implementation and testing of the ext2 file system for use on Singularity. The work so far has focused fully supporting reads from ext2. Caching of file system metadata and raw disk blocks makes reads speeds comparable to ext2 reads on Linux. Limited write support was completed, primarily to aid in testing. Write support has not progressed to a point that makes a performance analysis worthwhile.

The goal of this work is to evaluate the design of Singularity. Integration of a well known file system provides the opportunity to gauge the development experience when doing system programming in Singularity. Because the result is a file system that is compatible with ext2 running on Linux, a direct performance comparison is meaningful.

The development and testing described in this paper show that the Singularity design goals of high dependability and "good enough" performance are largely met. The protection provided by the micro kernel design prevented the Singularity kernel from crashing in the face of errors in the ext2 code. Performance testing showed that while not as fast as ext2 on Linux performance is not prohibitively low and the overhead of garbage collection is minimal.

The remainder of this paper is organized as follows: Section two is an overview of the important points of the Singularity design. Section three covers the details of my ext2 implementation on Singularity. Section four evaluates ext2 on Singularity as well as Singularity as a whole. Section five concludes the paper.

2 The Singularity Architecture

The information in this section is a summary compiled from references [1] through [9].

Singularity is a new operating system design by Microsoft's Research department. It is not intended for practical use, but rather as an environment in which to explore new ideas in operating system research. The design of Singularity explicitly targets dependability as the primary goal. Performance is a secondary goal only to the extent that it must be good enough to make the system practically usable. Despite the de-emphasis of performance, Microsoft has published micro-benchmarks showing that the Singularity design results in improved performance of many basic operating system operations such as a kernel API call, thread yield or starting a process.

2.1 The programming Environment

Programs for Singularity are written in Sing#, an extension of Spec#, which is itself an extension of C#. Almost all of the Singularity kernel is written in Sing#, with a small portion (around 5%) written in assembly and C++.

Despite the basis in C#, programs in Singularity do not run on a virtual machine such as the Common Language Runtime in Windows. Instead the Microsoft Intermediate Language (MSIL) code generated by the Sing# compiler is compiled to native machine code by the Bartok compiler. This process includes aggressive optimizations to further improve performance.

This optimization process is improved by taking advantage of the fact that processes in Singularity are sealed, meaning that code cannot be changed or linked in dynamically. Because the compiler can count on seeing all of a program’s code at compile time, many global optimization are possible which are not available in other systems.

Finally, all Singularity processes, as well as the kernel, are garbage collected. Each garbage collector runs independently which allows them to be configured differently depending on the needs of the application.

2.2 The Process Model

Singularity uses a micro kernel architecture. Outside of this micro kernel Singularity uses a single process model called a Software Isolated Process (SIP). As the name implies, SIPs do not rely on memory management hardware for address space protection as is in most modern operating system. Instead each SIP has a software protected “object space”. Static analysis, type safety and other language features are used to guarantee at compile time that code within a SIP cannot access memory that does not belong to it.

Software protection of processes removes much of the cost associated with context switches in a hardware based system. Historically, micro kernels have suffered from poor performance, due in large part to the overhead of doing many hardware based context switches. Software based protection makes Singularity feasible from a performance perspective.

The use of static analysis to protect process memory implies that Singularity must not run code that is not known to be trusted. Binaries can be signed to show they come from a trusted source and a compliant compiler. It would also be possible to distribute programs as MSIL code and the local system could do the required analysis on that before compiling and installing it. As mentioned in section 2.1, processes are sealed which prevents incorrect code from being loaded or generated at run time.

2.3 Communication Channels

All inter-process communication in Singularity is done via communication channels. These channels are a first class abstraction which is managed by the kernel and supported explicitly by the Sing# language.

Because dependability is the primary goal of Singularity, shared memory between processes is not supported. Shared memory is notorious for being difficult to program correctly. Even when done correctly, shared memory leads to decreased dependability when a process fails. In this case, any shared memory to which the process had access is left in an unknown state, essentially forcing the failure of all processes which also shared that memory.

Communication channels allow (and require) that the interactions between processes follow an explicit contract that can be understood and verified by the compiler. Furthermore, when a process fails, all processes in communication with it can be notified and handle the situation gracefully.

All channel messages and data are passed via a special kernel managed heap called the Exchange Heap. Objects in the exchange heap are explicitly allocated and deleted in a similar way to memory allocation in C++. Each object in the exchange heap may only be accessed by one process at a time, and that process may only hold a single pointer to it. These restrictions allow the compiler to statically determine that exchange heap memory is not leaked nor accessed when it is no longer owned. Pointers to exchange heap object may be passed between processes in a message, which also transfers ownership of the object. This allows for zero copy semantics when copying data buffers or other large objects in a message.

3 Ext2 Implementation

The focus of the ext2 implementation on Singularity was to fully support reads and to use caching to obtain good performance. The implementation consists of a command line control application, a system service, the core file system process and communication contracts. The details of these will be discussed in the remainder of this section.

3.1 Ext2ClientManager

The Ext2ClientManager is a system service which is launched during the system boot process. The purpose of this process is to handle requests for operations which are specific to ext2 volumes. The current version support the “mount” and “unmount” commands. A fully featured ext2 implementation would also include at least an additional “format”.

Ext2ClientManager is accessible at a known path in the “/dev” directory of the Singularity namespace. Clients wishing to perform ext2 volume operations send requests to the Ext2ClientManager at this location. The manager tracks volume and mount point status to prevent volumes being mounted more than once or a single mount point being used more than once. When a volume is mounted the Ext2ClientManager creates a new instance of an Ext2Fs process (described in section 3.3) to service requests for that mount point.

The Ext2ClientManager consists of about 300 lines of code.

3.2 Ext2Control

Ext2Control provides a command line interface to the Ext2ClientManager. The only purpose is to make the ext2 volume operations provided by the Ext2ClientManager accessible from the command line.

Other applications are free to contact the Ext2ClientManager directly in the same way that Ext2Control does, so Ext2Control is only used by interactive users via the shell.

The Ext2Control application consists of about 500 lines of code.

3.3 Ext2Contracts

The Ext2Contracts module is a project that defines the custom communications channel contracts used in ext2. These contracts are used by the Ext2ClientManager to communicate with an Ext2Fs instance, and by clients of the Ext2ClientManager to send mount and unmount commands. The contracts used by Ext2Fs to communicate with clients accessing files and directories within the file system are already defined by Singularity so that all file systems export the same interface.

This module also defines the data object (a struct in the exchange heap) which holds the settings to be used when mounting an ext2 volume. This includes the volume and mount point paths and maximum cache sizes.

The Ext2Contracts module consists of about 200 lines of code.

3.4 Ext2Fs

Ext2Fs is the module responsible for handling client accesses to the file system. An instance of the Ext2Fs process is started by Ext2ClientManager for each mount point. This Ext2Fs instance then handles all client requests to the file system at that mount point.

3.4.1 Directory Interfaces

As with all IO in Singularity, clients access files and directories through communication channels. Singularity provides a DirectoryServiceContract which defines the operations available for directories. These include creating and deleting entries, listing the entries and getting attributes. Similarly, there is a FileServiceContract governing the operations on files, such as reading and writing.

Opening a file or directory in Singularity is equivalent to binding a channel at to the desired object. Objects are identified by use of a path name string relative to the directory channel through which the request comes. For example a client might issue a bind request for the directory “/a/b” through a channel connected to the file system root. Assuming this succeeds it might issue a bind request for “/c/d.txt” through the new channel. The result would be a file channel bound to “/a/b/c/d.txt” through which it could then read and write the file.

The Singularity file system interface is designed to be idempotent. The only client state the file system must hold is which object (file or directory) a specific channel is accessing. In Ext2Fs the channel endpoints are held in a map structure which pairs them with an inode number. When a request is received over a channel it is satisfied by operating on that inode.

Directories are not expected to have files named "." and ".." to refer to themselves and their parent. The Singularity shell (nor any other application) has no notion of a "current directory" or changing directories. Ext2Fs complies with this design by suppresses the "." and ".." files when responding to a directory list request. Although it is clearly possible to display these files and correctly parse paths containing them, doing so could cause problems for a Singularity application attempting to walk the file system and not expecting these files to exist. A depth first search could easily result in a path containing just "./" for example. It would be possible to make the suppression of these files a mount option so that if a shell were developed that could make use of them they would be accessible.

3.4.2 File Interface

File operations are similarly idempotent. Ext2Fs holds no client state such as a file pointer for an open file (other than the inode number as mentioned above). Instead, each read or write request must specify a file offset at which the operation is to take place.

File reads and writes must supply Ext2Fs with a data buffer, buffer offset and maximum transaction size along with the file offset mentioned above. This interface implies that the buffer provided may be bigger than needed for any particular request. Because the client passes exclusive access to the data buffer to Ext2Fs in the read or write message, it is important that the buffer be returned to the client. In other systems a file write or error result might not return the original data because the caller could presumably keep a reference to it if needed. In Singularity the response to file operations always returns the buffer, whether or not it was successful. Clearly the buffer returned should be the same one that originally came from the client (or at least an exact copy) so that the client does not lose data.

The passing of data buffer ownership also presents a unique possibility for data loss if Ext2Fs fails. If an application were to have a large data buffer, and wish to write only a small part of it to disk, it would be most efficient to pass the whole buffer and specify the portion to be written. If Ext2Fs where to fail during such an operation the application would be cleanly notified that the communication channel to Ext2Fs was closed. However the data buffer would be lost, potentially destroying irreplaceable data. Although Singularity allows for zero copy writes, applications dealing with critical data need to make a copy before writing if they wish to recover from as many faults as possible without data loss.

3.4.3 Supported Operations

Although reading of files and directories is fully supported by the current Ext2Fs implementation, writing has not been fully implemented. The current code is able to overwrite data in an existing file and add new data blocks to the file if needed to satisfy a write. The creation of new files and directories is not supported. When servicing a write the only metadata updated is the file size in the inode and the addition of new data blocks (which updates the inode data block number and the block bitmap) if needed. Other inode data such as file modification time is not updated, nor are the statistics in the super block and block descriptors.

This limited write support allows data generated during performance testing to be easily transferred off the running Singularity system. This was done by writing the test results to a file and then rebooting into Linux where the volume could be mounted and the results file accessed. Prior to implementing writing the only way to gather test results was to print them to the screen and then type them into another computer by hand.

Ext2Fs uses a simple single threaded design with no explicit disk scheduling. The Ext2Fs thread simply blocks waiting for a request to arrive on one of the open communication channels and services them in the order they are received. This approach was sufficient for running tests with only one client, but would clearly not provide optimal results with multiple clients. A better design would be to explicitly queue requests and then attempt to service them in an optimal order. Using a thread pool to service requests could also improve performance.

Ext2Fs contains about 2400 lines of code.

3.5 Caching

Three caches are used in Ext2Fs: the inode cache, the block number cache, and the data block cache.

The inode cache is an obvious and straight forward optimization. Any operation on a file or directory will need at least some of the data in the inode and reading the information off the disk is extremely costly in cases where the request would otherwise involve few (or no) disk accesses.

The block number cache is actually an extension of the inode cache, in that it holds the list of data block numbers containing the data for the file. The first 12 entries of this list come directly from the inode and the rest (if any) are contained in the inode indirect blocks. This cache is implemented separately from the inode cache because the design was cleaner, and it also allows the cache to be configured separately from the inode cache.

The block number cache was found to be most important when reading large files. Reading the block number entries for a large file with many indirect blocks involves tremendous overhead because each data block read might require several disk accesses to compute where the data is located. Caching the information between requests removes the overhead.

The raw data block cache is perhaps the most obvious and straight forward cache. The data block cache makes the inode cache and block number cache redundant because inode or block number data will be in two caches when it is initially read. This situation is not too bad because once inode and block number data is cached, requests for them will never reach the data block cache and the blocks will eventually be. Despite this redundancy they still improve performance by removing the overhead of parsing the inode and block number data from raw disk blocks.

All three caches are implemented identically and are instances of the same class. The data is stored in a hash table. The inode and block number caches used the inode number as the key. The data block cache uses the block number as the key. Cache replacement uses a least recently used policy.

LRU replacement implemented with a linked list. Every time an item in the cache is accessed, the entry for it is moved to the end of the linked list. In this way the list is sorted in order of time of last access. When an entry needs to be removed from the cache the entry at the front of the list will be the least recently used. List entries can be removed from the list and placed at the end in constant time. A reference to the list entry is held along with the cache data in the hash table. Figure 1 illustrates the cache structure.

4 Results

4.1 Test configuration

Testing was done on an AMD Athlon 64 3200+ machine with 1 GB of RAM. The hard disk used for testing was a Western Digital 120 GB disk with a 7200 RPM rotational speed and a 2 MB buffer. The disk was attached via a parallel ATA interface.

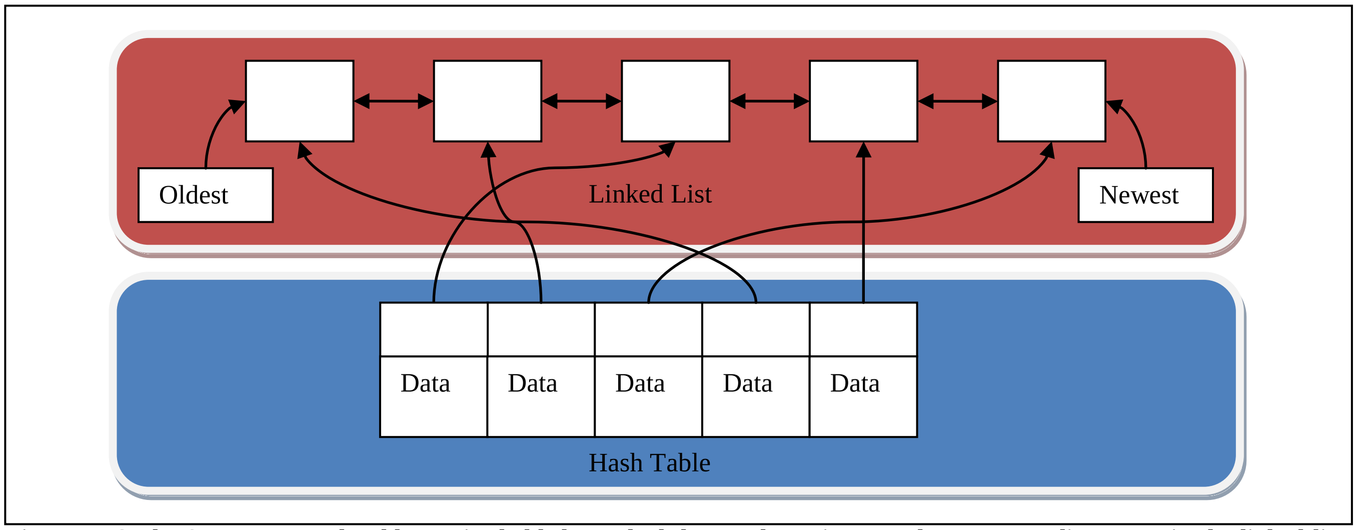

Figure 1: Cache Structure. Hash table entries hold the cached data and a pointer to the corresponding entry in the linked list. When an entry is accessed the list node is moved to the back of the list.

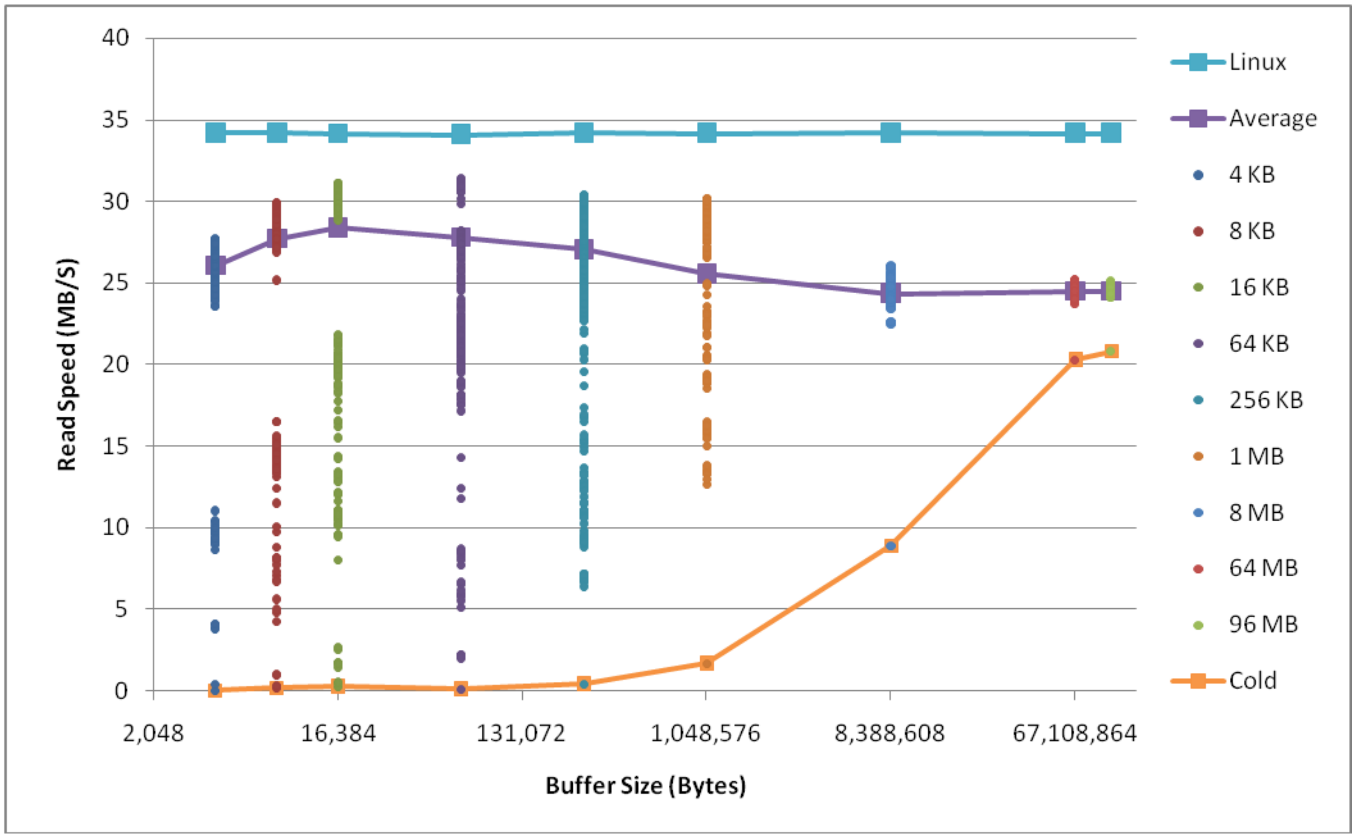

Figure 2: Sequential reading of a 350 MB file.

Performance evaluation of ext2 on Singularity focused on sequential read speed. Sequential reads were performed using a variety of read buffer sizes ranging from 4 KB to 96 MB. Figure 2 plots the read speed of reads using 350 MB file. Each request was timed using the CPU cycle counter and the result plotted over the range of buffer sizes. Tests were run immediately after mounting the volume so that the caches would be cold. The volume was remounted between the runs for each buffer size to clear the caches.

4.2 Test Results

The baseline for comparison in this test was the read performance of the same 350 MB file in Linux. The top line in the graph shows this to be just over 34 MB/s. Charting the average read speed in Singularity shows that the maximum performance of about 28 MB/s is achieved by using 16 KB buffers.

Testing file smaller than 350 MB showed very similar results.

An interesting feature of this chart is the significant gap in between clusters of read results for the four smallest buffer sizes. One set of measurements cluster near the average and others are distributed more widely at slower read speeds. Figure 3 shows a time series of 200 sequential reads from the 350 MB file using 16 KB buffers. In this graph the majority of the reads can be seen near 30 MB/s as expected from Figure 2. The slower reads are distributed in a periodic fashion. No direct evidence of the source of these periodic slower reads was collected, but it is consistent with what one would expect from a disk hardware effect such as the head seeking from one track to the next. The reads using buffers greater than 256 KB don't show this clear separation. As the buffer size rises, the individual read performance begins to converge to the average.

The line labeled "cold" in figure 2 is a plot of the very first read request at each buffer size. Because this is the first request, the inode and block number caches are empty and all meta data must be read from the disk. The performance of this read is equivalent to what the overall performance would be with no caching at all. When using 4 KB buffers the measured speed was 0.005 MB/s. This does not rise above 1 MB/s until 1 MB buffers are used.

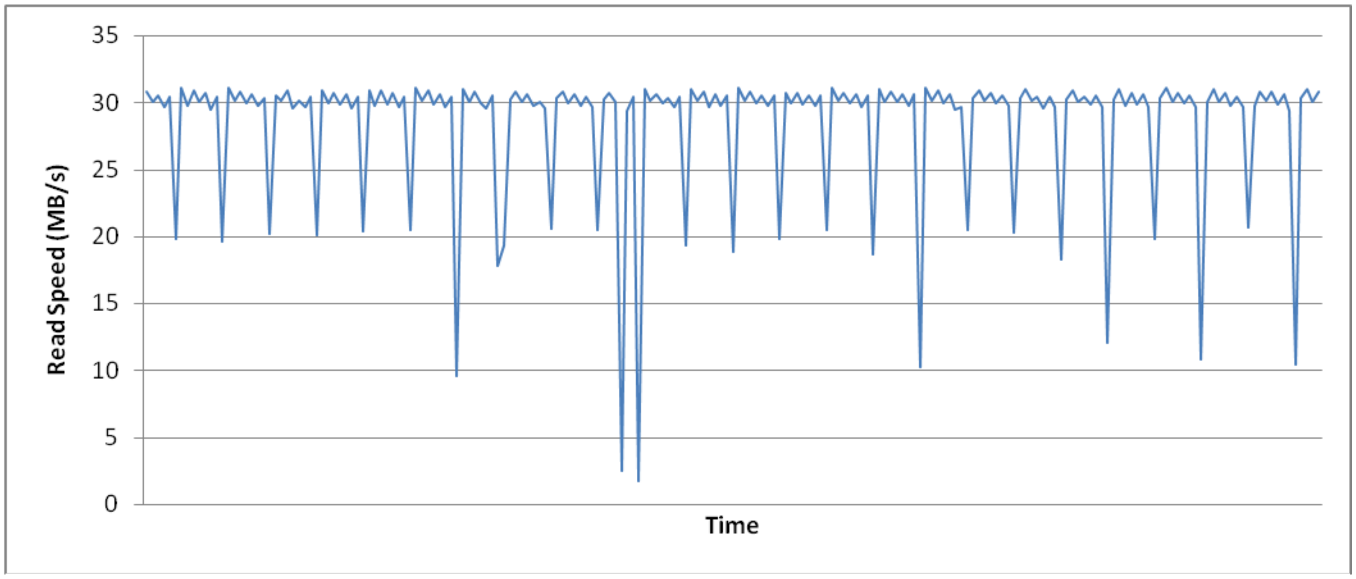

Figure 3: 16 KB sequential reads of the 350 MB file.

Figure 3 shows large “double spike” performance reduction that does not appear to be caused by the same periodic source as the other spikes. This type of interruption is seen on a semi regular basis when viewing the data at a longer time scale and is likely the result of Ext2Fs being preempted by another processes.

4.3 Garbage collection

Garbage collection is often the source of controversy when discussing the merits programming languages and runtime systems. One goal of the ext2 performance evaluation was to quantify the effect of garbage collection on ext2 performance. Figure 4 plots the response times for each of 2000 read requests to the 350 MB file using 16 KB buffers. The large spikes of around 30 ms are reads which have been interrupted by garbage collector runs. This can be verified by matching this data with the garbage collector information given by the Singularity debugger. Once the reads affected by the garbage collection were identified, an average read speed was computed without considering those requests. The difference between the average read speed with garbage collection and without was less that 0.1%.

Although garbage collection has very little effect on average read speeds, it does introduce a large variance in the latency of read requests. This is a potential problem applications requiring real time performance, such as video playback or embedded control. One solution to this is to issue read requests using large buffers so that delay due to garbage collection is a small part of the total request time.

Garbage collection run time is proportional to heap size, regardless of the amount of data actually collected. This is true because the heap must be scanned in order to determine if any collection is needed. As a result, applications such as Ext2Fs which potentially contain a large cache in the heap will have a larger garbage collection overhead.

The exchange heap in Singularity provides an opportunity to minimize garbage collection activity. No garbage collection is run on the exchange heap because exchange heap objects are explicitly allocated and deleted by the owner. Ext2Fs could allocate its caches in the exchange heap, and thus minimize the size of the local garbage collected heap.

Another advantage to holding the data block cache in the exchange heap would be that data blocks which were speculatively read from the disk (in anticipation of use due to a sequential file read for example) could be inserted directly into the cache without the overhead of copying from the exchange heap into the local process heap as is done in the current implementation.

A best case read latency could be realized by immediately returning a data block held in an exchange heap cache in response to a client read request. This could be done by simply removing it from the cache and passing it back to the user with zero copying. This has the clear drawback of

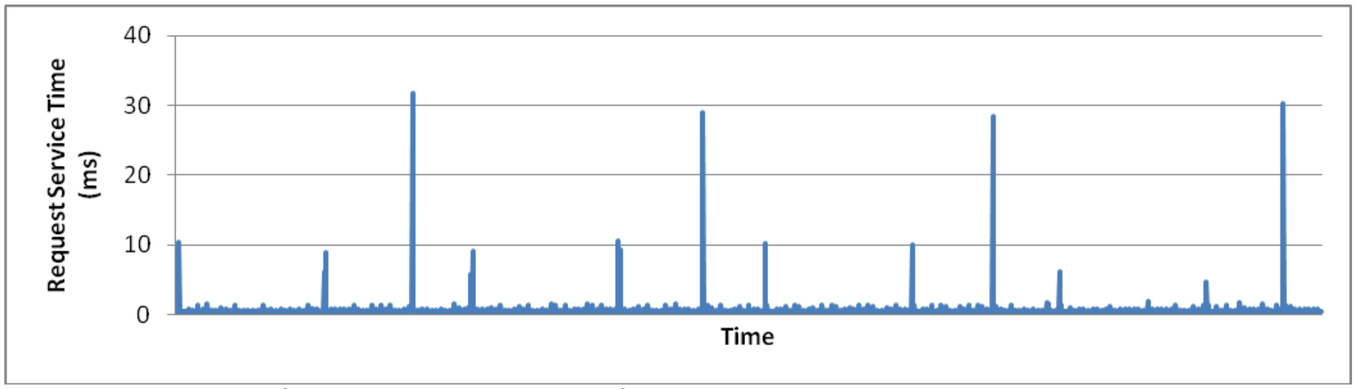

Figure 4: Service latency of sequential reads to the 350 MB file.

being equivalent to a "most recently used" cache removal policy. Once data is passed to a client in this way, the only way to recover it is to read it from the disk. This strategy would also be possible only if the buffer passed by the client in the read request matches the size of the cached buffer exactly.

A strategy of short term caching in the exchange heap should be very effective in the case where a client does a one-time sequential read of a file. If Ext2Fs were able to detect that such an access was beginning it could speculatively read ahead of the client stream request and hold the data blocks in the exchange heap. The data would then be returned very quickly to the client resulting in a very low latency on read requests from the client's perspective. This strategy would only be effective if the time between client requests was great enough to allow Ext2Fs to fetch the next block from disk before it was requested.

An extension of this scheme would improve latencies on warm cache reads. Data cached normally in the Ext2 heap could be copied to the exchange heap speculatively. As before this would be done when a streaming read was expected.

4.4 Singularity design

One of the goals of this project was to evaluate how the Singularity design effects the development of system software such as a file system.

4.4.1 Language Features

One clear advantage is the use of a modern, type safe, object oriented language. Kernel level development in current monolithic systems such as Linux or Windows is typically done in C, and can often impose a restricted execution environment, such as the lack of floating point support in the Linux kernel. Singularity's micro kernel design means that file systems, drivers and other code that is traditionally a kernel extension is instead implemented a SIP, just like a user application. System programming benefits from the use of all the normal system libraries, IO for debugging and full support for the normal user level debugger.

The system module designer is free to utilize software engineering techniques and practices such as abstraction, code hiding and code reuse through the use of classes. The systematic use of exceptions for error handling prevents the widespread suppression of errors due to C error code returns being ignored. This is known to be wide spread in file systems on Linux [10]. Furthermore, Spec# statically checks for the possible uses of null pointers. The programmer is forced to either declare objects using a non-nullable types or else insert run-time null pointer checks before pointers are used. This removes the possibility for a large class of bugs related to incorrect or inconsistent null-pointer checks. These bugs have also been shown to be widespread in the Linux and FreeBSD kernels [11].

4.4.2 Trusted Code

The Singularity design appears to have effectively reduced the amount of "unsafe code" in the system to a manageable level. All the unsafe code dealing with raw pointers and memory is encapsulated in the Singularity micro kernel and associated trusted libraries. This small critical code base (less than 5% of kernel code) can then be extensively tested and carefully controlled by the small Singularity kernel development team [1,4].

The code running in SIPs is known to be "safe" because the Sing# language and compiler prevent the creation of code that accesses memory outside the local object space.

Type safety guarantees that object references and pointers cannot be assigned from arbitrary values such as integers. At worst a pointer may be null, and the compile time and run time check prevent null pointer use [1].

4.4.3 Dependability

It is hard to formally measure the dependability of the Singularity design. As an anecdotal measure, it can be said that Sing# compiler checks prevented several errors during ext2 development that would have resulted in memory leaks in the exchange heap. Such leaks are particularly hard to detect and avoid in a traditional system using C or C++ because seeming minor code changes can easily change the logical flow of object or memory buffer ownership and result in the object not being deleted. Errors of this nature are particularly hard to detect because they often manifest only as slow memory leaks which do not affect system performance in the short term.

Another advantage of the micro kernel design is that the Ext2Fs interface to the rest of the system is well defined and very simple. Implementation of Ext2Fs did not require the knowledge of or use of any extensive kernel APIs or kernel data structures. Instead Ext2Fs is required to interact with clients using the directory and file service contracts, and with the disk driver use the disk driver contract. Each of these contracts defines a small number of possible messages which can be generated and handled in a consistent and straight forward way. In contrast, a file system in a monolithic kernel such as Linux must use a very large API and must directly manipulate and interpret data structures shared with other parts of the kernel. Errors in such code can have catastrophic results.

Perhaps the best testament to Singularity's dependability was the extremely good system stability during the development of ext2. In the author's experience, development of system level code in a monolithic kernel often results in system instability (note that this is highly dependent on programmer skill, so there is some hope that this experience was worse than is typical). Many errors related to shared data structures or incorrect use of memory and pointers can result in the failure of the kernel. The experience of developing ext2 on Singularity was much more like normal user level application development. There were many bugs and errors that needed to be corrected, but they only resulted in the failure of the SIP in which ext2 was running. If the ext2 process terminated as the result of a failure, it only resulted in open channels closing. Other processes could (and did) recover gracefully. In the case of a shell-based application which was using the file system unexpected failures would typically result in an error message on the screen and the process terminating. Such a program generally cannot proceed without a file system. In the case of other system services such as the Singularity Name Service which maintains the Singularity name space, such a failure would result in the ext2 mount point disappearing from the visible name space.

In the case of a failure in ext2 that did not result in the process terminating (such as a deadlock) it was possible to also deadlock client processes by forcing them to block forever waiting for a message reply. This could be prevented by client applications implementing a timeout mechanism. In no case did this cause problems for the system as a whole or to unrelated processes.

During the whole development of ext2, the Singularity kernel or critical system services where never observed to crash or require a reset to recover from an error. The only exception to this is when the system runs out of physical memory. It appears to be possible to exhaust system memory by attempting to allocate more memory than is physically available. Singularity does not support paging of memory to disk, and apparently does not attempt to forcibly terminate a process or take some other action to recover memory. When physical memory was full, the kernel was unable to make progress because memory allocations kept being requested and failing. It is unclear if the offending allocation was in the kernel and the kernel itself was no longer able to operate, or if a SIP was continually retrying a memory allocation that was being denied. In any case this is not so much as issue of Singularity's design as it is a question of policy when the system runs low on memory.

5 Conclusion

This paper describes the implementation of the Linux ext2 file system on Singularity operating system. The purpose of this project was to investigate Singularity's design and performance.

Quantitative performance measurements are one of the most straight forward ways to compare Singularity to other systems. Microsoft's Singularity team explicitly states that performance is a secondary goal to dependability in the Singularity design. Performance measurements of this initial implementation of ext2 are in line with this goal. Best case average read speeds lag behind those for Linux, but do not show that the Singularity design is fundamentally unable to produce usable performance. It is possible that further design work and optimizations would be able to produce performance that matches or exceeds the Linux implementation.

The primary goal of dependability was not tested in a formal or quantitative way. The experience writing and testing ext2 on Singularity provides reasons for optimism about overall dependability. Singularity did not suffer any crashes during the development and testing process. The micro kernel architecture successfully isolates components such as a file system. In a monolithic system such a module would be run in kernel space and thus able to induce serious system wide errors.

While the isolation of a micro kernel has clear benefits for dependability, it could prove to be a hindrance for the integration of system services, and therefore for performance. A monolithic kernel is able to easily share resources between components. Caches could be combined and shared between different sub-system such as a file system or virtual memory system. Singularity's process isolation model would prohibit such sharing, or would at least prevent concurrent access to a shared resource. It is not clear at this point if such restrictions will have a significant negative effect on performance.

Finally, the use of a modern garbage collected programming language for system level code appears to be quite feasible. An aggressive, globally optimizing compiler which produces machine code from Microsoft Intermediate Language code removes the massive overhead that would normally make a language like C# completely unsuitable for system programming. Static analysis at compile time and advanced language features such as non-nullable types and tracking of object ownership prevent many of the most common errors that plague kernels written in C.

6 References

[1] **An Overview of the Singularity Project**, Galen C. Hunt, et al. *Microsoft Research Technical Report MSR-TR-2005-135*, Microsoft Corporation, Redmond, WA, October 2005.

[2] **Singularity: Rethinking the Software Stack**, Galen Hunt and James Larus. *Operating Systems Review*, Vol. 41, Iss. 2, pp. 37-49, April 2007. ACM SIGOPS.

[3] **Sealing OS Processes to Improve Dependability and Safety**, Galen Hunt, Mark Aiken, Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. *Proceedings of EuroSys2007*, pp. 341-354, Lisbon, Portugal, March 2007. ACM SIGOPS.

[4] **Authorizing Applications in Singularity**, Wobber, Ted, Aydan Yumerefendi. Martín Abadi, Andrew Birrell, and Dan Simon. *Proceedings of EuroSys2007*, pp. 355-368, Lisbon, Portugal, March 2007. ACM SIGOPS.

[5] **Language Support for Fast and Reliable Message-based Communication in Singularity OS**, Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. *Proceedings of EuroSys2006*, pp. 177-190. Leuven, Belgium, April 2006. ACM SIGOPS.

[6] **Solving the Starting Problem: Device Drivers as Self-Describing Artifacts**, Michael Spear, Tom Roeder, Orion Hodson, Galen Hunt, and Steven Levi. *Proceedings of EuroSys2006*, pp. 45-58. Leuven, Belgium, April 2006. ACM SIGOPS.

[7] **Access Control in a World of Software Diversity.** Martín Abadi, Andrew Birrell, and Ted Wobber. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pp. 127-132. Santa Fe, NM, June 2005. USENIX.

[8] **Broad New OS Research: Challenges and Opportunities.** Galen Hunt, James Larus, David Tarditi, and Ted Wobber. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pp. 85-90. Santa Fe, NM, June 2005. USENIX.

[9] **Making system configuration more declarative.** John DeTreville. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pp. 61-66. Santa Fe, NM, June 2005. USENIX.

[10] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACIDUSSEAU, R. H. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 206.220

[11] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57--72, 2001.