# Software Aging Analysis of the Linux Operating System

**Sneha sunny[1], Sona Binoy[2], Amitha Joseph[3]**

[1]Santhigiri College of Computer Sciences,
Vazhithala, Thodupuzha
bcab19_2233@santhigiricollege.com

[2]Santhigiri College of Computer Sciences,
Vazhithala, Thodupuzha
bcab19_2234@santhigiricollege.com

[3]Assistant Professor, Computer Science Department,
Santhigiri College of Computer Sciences ,Vazhithala, Thodupuzha
amithajoseph@santhigiricollege.com

***Abstract- Software systems running continuously for a long time tend to show degrading performance and an increasing failure occurrence rate, due to error conditions that accrue over time and eventually lead the system to failure. This phenomenon is usually referred to as software aging. Several long-running mission and safety critical application have been reported to experience catastrophic aging-related failures. Software aging sources that is, aging-related bugs may be hidden in several layers of a complex software system, ranging from the operating system (OS) to the user application level. This paper presents a software aging analysis at the operating system level, investigating software aging sources inside the Linux kernel. The majority of the research efforts in studying software aging have focused on understanding its effects theoretically and empirically.***

**Keywords-** Software aging, Linux kernel, trend analysis

## 1. Introduction

Software Aging can be defined as a continued and growing degradation of software's internal state during its operational life. This problem leads to progressive performance degradation, occasionally causing system crashing. Due to its cumulative property, it occurs more intensively in continuously running systems that execute over a long period of time. It is typically caused by accrued error conditions, such as round-off errors, data corruption, storage space fragmentation, or unreleased memory regions. Detecting and removing the sources of software aging (i.e., the so-called aging-related bugs [1]) is very difficult at testing time, since aging becomes evident only after a long operational time. For this reason, software aging represents one of the most subtle dependability threats in today's business- and safety critical software systems. Past research work reported software aging phenomena that manifested as the increasing consumption of Operating System (OS) resources, such as free memory and swap space exhaustion [2]–[4]. Subsequent studies found software aging sources in several software applications, such as web servers [4], telecommunication systems [5], and SOAP servers. Therefore, several approaches were developed to predict the time to failure at operational time, in order to plan proper actions (that are usually referred to as software rejuvenation) with an optimal schedule (i.e., neither too early, because it would be expensive, nor too late, because a failure may occur before rejuvenation).

Although relevant, the analysis of software aging sources at application level represents only a partial view of the issue. In fact, the OS itself can be a source of software aging phenomena, since it is a large and bug-prone part of complex software systems. Being able to detect and isolate the aging contribution of the OS would yield insights about aging trends for a wide number of applications based on it. Moreover, these insights can be exploited for planning software rejuvenation strategies tailored to the OS , as well as for identifying aging related bugs in the OS code. In this work we carry out an experimental campaign to analyze software aging inside the Linux OS kernel. First, the study tests the presence of aging sources at the OS level. The goal of this phase is to statistically confirm if and in what extent the Linux kernel is actually affected by aging-related bugs. A deeper analysis is then carried out, with the goal of figuring out how the usage of each internal subsystem impacts on aging trends. By means of a kernel tracing tool specifically developed for this study, we collected usage information about several subsystems, such as memory management and the filesystem. Usage information has been related with the observed aging trends, by means of

multiple linear regression and principal components analysis; these relationships were exploited to find out kernel subsystems responsible for aging phenomena.

## 2. What is Software Aging

Software aging is a phenomenon that occurs in all types of software, regardless of who created it or how powerful the software was when first purchased. Software aging is the gradual degradation of coding in which fragmentation begins manifesting itself, leading to slower performance and decreased output. Software rejuvenation is aimed at correcting the errors of aging, but it only offers a limited fix to the problem. Continual upgrades to the software also exacerbate the aging effects and the original coding becomes warped. Most users find it easier, and less time consuming, to buy new software instead of trying to maintain aging software. Software being used for the first time is fresh and has not been introduced to any degrading code, nor has its information been fragmented. This means the program can run quickly, without problems. As software aging progresses, the operating system will be able to feed fewer resources into the program. With fewer resources and degrading code, the software starts to cause lags or may automatically shutdown. Upgrades, while seemingly good, can have devastating effects on aging software. An upgrade introduces new code. This new code can further the effects, or visibility, of fragmented code. The upgrade also introduces more code, which increases the size of the program. This means even more resources are needed to produce the same output as before the upgrade was added. Software rejuvenation has been employed to correct the damaging effects of software aging. There are many types of software rejuvenation techniques but, overall, they aim to ease fragmentation and return the software back to its original coding. Software rejuvenation offers a limited fix, because it cannot correct all the errors, and is best used on software that shows from low to medium signs of aging. Years after getting a piece of software, the effects of software aging will become unavoidable. The amount of time is not set, because it depends on how well the program was made, but 10 years is usually the upper range of when the effects of aging make the program nearly unusable. Software rejuvenation can correct some of the errors when the software gets to this point, but the aging effects will still make it difficult to use the program. When the software aging effects are unavoidable, most users opt to purchase new software. The new software will not need the same upkeep until later in its life and will be able to produce a better output than the aged software. Purchasing newer software, especially for businesses, frees up human resources to work on other tasks or projects. This paper, we test this hypothesis by analyzing aging sources inside the OS. Our work relies on a workload- and measurement based approach, in that we monitor the OS's health under different controlled workloads, collect data characterizing its behaviour, and then analyze them to identify aging sources inside the Linux kernel.

## 3. Reasons of Software Aging

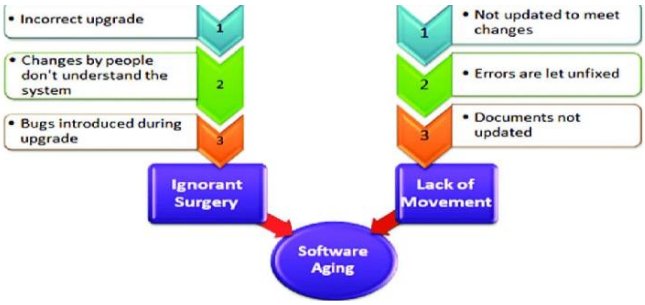


Figure 1: flow chart reasons for software aging

### 3.1 Ignorant Surgery

Although it is essential to upgrade software to preventaging, changing software can cause a differentform of aging. The designer of a piece of softwareusually had a simple concept in mind when writingthe program. If the program is large, understandingthat concept allows one to find those sections of theprogram that must be altered when an update or correction is needed. Understanding that concept alsoimplies understanding the interfaces used within thesystem and between the system and its environment.Changes are made by people who do not understand the original design concept almost always causethe structure of the program to degrade. Under thosecircumstances, changes will be inconsistent with theoriginal concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one mustknow both the original design rules, and the newly introduced exceptions to the rules, to understand theproduct. After many such changes, the original designers no longer understand the product. Those whomade the changes, never did. In other words, nobodyunderstands the modified product. Software that hasbeen repeatedly modified (maintained) in this waybecomes very expensive to update. Changes takelonger and are more likely to introduce new "bugs".Change induced aging is often exacerbated by thefact that the maintainers feel that they do not havetime to update the documentation. The documentation becomes increasingly inaccurate thereby makingfuture changes even more difficult.

- Incorrect Upgrade
- Change by people don't understand the system
- Bugs introduced during the upgrade

### 3.2 Lack of Movement

Over the last three decade, our expectations aboutsoftware products has changed greatly. I can recallthe days when a programmer would "patch" a program stored on paper tape by using glue and paper.We were all willing to submit large decks of cardsand to wait hours or days for the job to compile andrun.

When interactive programming first came in, wewere willing to use cryptic command languages. Today, everyone takes on-line access, "instant" response, and menu-driven interfaces for granted. Weexpect communications capabilities, mass onlinestorage, etc. The first software product that I built (in1960) would do its job perfectly today (if I could finda Bendix computer), but nobody would use it. Thatsoftware has aged even though nobody has touched it.Although users in the early 60's were enthusiasticabout the product, today's users expect more. My oldsoftware could, at best, be the kernel of a more convenient system on today's market. Unless software isfrequently updated, it's user's will become dissatisfied and they will change to a new product as soon asthe benefits outweigh the costs of retraining and converting. They will refer to that software as old andoutdated.

- Not updated to meet changes
- Errors are let unfixed
- Documents not updated

## 4. The Cost of Software Aging

1. Newer versions of software are frequently released. As drivers become out of date, compatibility issues can occur with newer software and operating system updates. Troubleshooting these issues can be time consuming.

2. Hardware failures on the aging technology. Can be faced with the option of waiting for a replacement machine to arrive and be configured before the user can get back to work or of potentially replacing the failed component at a high cost just to keep the aging machine available. A common response to this issue is, "If the computer fails, I will just run out to my nearest store and purchase a replacement." While you may get a replacement machine quickly, the costs associated often can be double the cost of a planned replacement. This is due to the fact that retail store computers tend to be geared towards home users and are not correctly configured for business networks.

3. Loss of employee productivity when using aging hardware. A current-generation computer takes between 1-3 minutes to complete the login process from a power-off state. Older hardware can take upwards of 5 minutes, with some reaching the 10-minute mark. As employee mailboxes grow in size, machines that have lower amounts of memory and slower hard drives can take longer to open email and to switch between folders.

4. Costs associated with aging servers. In the case of a server, sometimes the cost is not as evident. Aging servers can result in higher support call volume for things such as email database fragmentation or lack of drive space. As warranty support for servers expires, there is the potential that failure can result in lack of email and network access for an entire company. A failed motherboard on an out-of-warranty server can take several days to source parts for and repair.

During this time, the company network will be severely impaired, if not completely unusable. The cost of such a failure can be in the thousands of dollars.

Additionally, server operating systems give priority to background and system processes, such as the mail server or file server functions. Due to this, as a server ages and becomes more overloaded, console-based actions are the first to exhibit slowness. The act of adding an additional user account on a 1-to-2-year-old server takes on average 5-15 minutes. A 5-year-old server running a fully patched version of its original operating system, hosting email, and file storage can take 10-20 minutes on average to log on to and add a user. As the required action on the server gets more complex, the time required to complete the action is also extended on the aging hardware. This results in higher support costs for day-to-day operations and maintenance on the server.

### 4.1 How often should replace technology

Based on this information, the recommended replacement schedule for a server is 4.5-5 years. This will provide the greatest discount on the manufacturer's warranty contracts, along with replacing the server before support costs and productivity loss become too great. Desktops should be replaced at most every five years, with power users replacing on a more frequent schedule. Due to the additional abuse that laptops experience, along with slower performance rating, a laptop replacement schedule should be every four years, with power users replacing more frequently.

## 5. Prevent Medicine

### 5.1 Design for success

This principle is known by various names:

- information hiding
- abstraction
- separation of concerns
- data hiding
- object-orientation

To apply this principle one begins by trying to characterize the changes that are likely to occur over the "lifetime" of a product.

Since actual changes cannot be predicted, predictions will be about classes of changes:changes in the UI, change to a new windowing system, changes to data representation, porting to a new operating system. Since it is impossible to make everything equally easy to change, it is important to:estimate the probabilities of each type of change.Organize the software

so that the items that are most likely to change are “confined” to a small amount of code.

### 5.2 Keeping Records

Even when software is well designed, it is often not documented.

When documentation is present it is often:

- poorly organized
- inconsistent
- Incomplete
- written by people who do not understand the system

Hence, documentation is ignored by maintainers. Worse, documentation is ignored by managers because it does not speed up the initial release.

### 5.3 Second opinion

In engineering, as in medicine, the need for reviews by other professionals is never questioned.In designing a building, ship, aircraft, there is always a series of design documents that are carefully reviewed by others.

## 6 Effect Aging Linux Os

Aging sources actually exist in the Linux kernel; they manifested as a statistically significant aging trend of memory consumption. This result is of practical importance for final users, which can benefit from a rejuvenation schedule that individually takes into account aging at the OSand the application layer. Moreover, the analysis of internal subsystems identified a set of potential aging sources in the filesystem and process management subsystem, which manifested a significant contribution to the overall aging trend. A further experiment allowed us also to quantify a non-negligible contribution of the filesystem to the memory consumption trend, which impacts the overall aging effects in a large, complex software system.software aging inside the Linux OS kernel and affected by aging-related bugs. Usage of each internal subsystem impacts on aging trends.

System-wide and application-specific. System-wide provides information related to subsystems that are shared and therefore influenced by other system elements. Examples of shared subsystems are OS, VM, and VMM levels. Indicators in this category are often used to evaluate the aging effects in the system as a whole and not for specific application process, since their shared nature may cause noise in the measured data. Examples of aging indicators in this category are free physical memory, swap spaces size, among others. Application-specific indicators provide specific information about an individual application process and therefore give more accurate information about the process than the system-wide indicators. When the application process is running under a VM (e.g., java programs) then indicators applied to the VM can also be used as reference for the application being executed under the VM. Examples of aging indicators in this category are resident size of the process (RSS), JVM heap size, response time, the system was stressed by means of a load generator,The load generator stresses several subsystems (e.g., process management, memory management) by using system calls provided by the OS (e.g., by allocating memory and by writing to the disk). The Process Management subsystem was also related to the aging trend memory could be leaked when a new process is started, or when a process exits. performance and resource usage. Software aging manifests itself as resource depletion and performance degradation. Data about the usage of OS subsystems’ functionalities (i.e., the workload parameters, such as number of interrupts or disk operations processed in a time period).workload parameter correlated to aging phenomena can be a symptom of aging bugs in a specific subsystem, therefore these relationships can be exploited to diagnose aging phenomena in OS subsystems. Aging indicators and workload parameters

### 6.1 memory consumption (MC)

Memory consumption is given by: MC = TM −FM −PC (1) where TM, FM, and PC are total memory, free memory, and page cache size respectively. This metric is provided bythe standard Linux kernel; it can be queried by means of the free utility. The page cache contains a copy of recently accessed files in kernel memory. Since the page cache can get all the free memory not allocated by the kernel or user processes, its memory consumption is quite large therefore, it is subtracted from MC. The MC is periodically sampled and stored in a trace.

### 6.2 system call latency (SCL)

The SCL indicator is considered since bugs that affect the system performance, such as aging-related bugs, may affect performance of services at the OS interface. Moreover, performance degradation is a common symptom of aging phenomena due to the accumulation of errors and stale resources

## 7 Conclusion

(1) We cannot assume that the old stuff is known anddidn’t work. If it didn’t work, we have to find out why.Often it is because it wasn’t tried.
(2) We cannot assume that the old stuff will work.Sometimes widely held beliefs are wrong.

(3) We cannot ignore the splinter softwareengineering groups. Together they outnumber thepeople who will read our papers or come to ourconferences.
(4) Model products are a must. If we cannot illustratea principle with a real product, there may well besomething wrong with the principle, Even if theprinciple is right, without real models, the technologywon't transfer. Practitioners imitate what they see inother products. If we want our ideas tocatch on, we have to put them into products. There is a legitimate, honorable and important place for researchers who don't invent new ideas but, instead, apply, demonstrate, and evaluate old ones.
(5) We need to make the phrase "software engineer" mean something. Until we have professional standards, reasonably standardised educational requirements, and a professional identity, we have no right to use the phrase, "Software Engineering".

## References

[1] HESTER, S.D., PARNAS, D.L., UTTER, D. F.,
"Using Documentation as a Software Design Medium", Bell System Technical Journal, 60, 8, October 1981, pp. 1941-1977,
[2] PARNAS, D. L., WEISS, D. M., "Active Design Reviews: Principles and Practices", Proceedings of the 8th International Conference on Software Engineering, London, August 1985. Also published in Journal of Systems and Software, December 1987.
[3] VAN SCHOUWEN, A. J., PARNAS, D. L., MADEY, J., "Documentation of Requirements for Computer Systems", presented at RE '93 IEEE International Symposium on Requirements Engineering, San Diego, CA, 4-6 January, 1993.
[4] PARNAS, D. L., MADEY, J., "Functional Documentation for Computer Systems Engineering (Version 2)", CRL Report 237, CRL-TRIO McMaster University, September 1991, 14 pgs. (to be published in Science of Computer Programming)
[5] PARNAS, D, L., "Tabular Representation of Relations", CRLReport260,CRL.

## Author Profile

**Sona Binoy** Pursuing Bachelor of Computer Application from Santhigiri college of Computer Sciences, Vazhithala in 2019-2022

**Sneha Sunny** Pursuing Bachelor of Computer Application from Santhigiri college of Computer Sciences, Vazhithala in 2019-2022

**Amitha Joseph** received the MCA professional degree and MPhil in Computer Science. She is currently working as an assistant professor in Santhigiri College of Computer Sciences, Vazhithala.