When malloc() Never Returns NULL -- Reliability as an Illusion

Kudrjavets, Gunnar; Thomas, Jeffrey; Kumar, Aditya; Nagappan, Nachiappan ; Rastogi, Ayushi

Published in:
Proceedings of 2022 IEEE 33nd International Symposium on Software Reliability Engineering (ISSRE)

When malloc() Never Returns NULL—Reliability as an Illusion

Gunnar Kudrjavets
University of Groningen
9712 CP Groningen, Netherlands
g.kudrjavets@rug.nl

Jeff Thomas
Meta Platforms, Inc.
Menlo Park, CA 94025, USA
jeffdthomas@fb.com

Aditya Kumar
Snap, Inc.
Santa Monica, CA 90405, USA
adityak@snap.com

Nachiappan Nagappan
Meta Platforms, Inc.
Menlo Park, CA 94025, USA
nnachi@fb.com

Ayushi Rastogi
University of Groningen
9712 CP Groningen, Netherlands
a.rastogi@rug.nl

***Abstract*—For decades, the guidance given to software engineers has been to check the memory allocation results. This validation step is necessary to avoid crashes. However, in user mode, in modern operating systems (OS), such as Android, FreeBSD, iOS, and macOS, the caller does not have an opportunity to handle the memory allocation failures. This behavioral trait results from the actions of a system component called an out-of-memory (OOM) killer. We identify that the only mainstream OS that, by default, lets applications detect memory allocation failures is Microsoft Windows. The false expectation that an application can handle OOM errors can negatively impact its design. The presence of error-handling code creates an illusion of reliability and is wasteful in terms of lines of code and code size. We describe the current behavior of a sample of popular OSs during low-memory conditions and provide recommendations for engineering practices going forward.**

***Index Terms*—Allocator, memory, OOM, OOM killer.**

I. Introduction

This paper is prompted by observations about how various OSs behave under low memory conditions. Based on our industry experience, we notice that the application's OOM error-handling code never executes on specific OSs. We do not observe specific pre-programmed actions such as recovery, retrying allocations, or the presence of relevant log messages. Instead, applications are just being terminated. However, recommended software engineering practices encourage engineers to diligently write error-handling code to verify the success of each explicit memory allocation [1], [2], [3]. Handling the OOM errors is supposedly necessary to avoid crashes and ensure that an application continues functioning in a stable state.

We argue that for popular kernels and OSs based on them, such as Android, Linux, iOS, or FreeBSD, the core assumption that an application can reliably handle the OOM conditions is outdated. In most mainstream OSs, *a user mode application will never have an opportunity to handle a failure to allocate memory*. This deviation from a traditional assumption about how to design robust applications is caused by modern OSs using a system component called an OOM *killer* (see Section II-B). Suppose the amount of memory allocated by an application exceeds a certain quota (e.g., a per-process limit) or the amount of free memory for an entire OS drops below a certain threshold. In that case, the OOM killer will start terminating processes to free memory. Processes that will be killed can be an application that fails to allocate memory or any other processes that match the heuristic that the OOM killer uses. The presence of an OOM killer has implications on how reliable applications should be designed and what assumptions they can make about their ability to handle and recover from errors.

We conduct experiments on Android, FreeBSD, iOS, Linux, macOS, and Windows to investigate how an application behaves under low-memory conditions. We also investigate how high memory consumption impacts other applications executing in parallel. Based on the sample of OSs we investigate; we find that when using default settings: (a) only on Windows, an application can reliably know if OS does not have enough memory to satisfy the allocation request and react appropriately, (b) applications consuming the most of memory will be terminated before other applications executing in parallel experience the consequences of low-memory conditions, and (c) though not always a suitable technique, using an OOM killer is a practical approach to maintain OS's stability. We offer suggestions on how applications should be designed given these limitations and how the presence of an OOM killer changes the prevailing long-held beliefs about memory management.

II. Background and motivation

A. Memory management

An OS is responsible for managing access to various resources exposed to applications. These resources include CPU time allocation, memory, storage, and direct hardware access. Accessing any of these resources is based on an application's *exact* demands. For example, an application can create a file, allocate a fixed size of memory, or request the creation of a thread. Good programming practices for writing reliable

code [1], [2], [3] suggest that an application must always check the return value from a mechanism (e.g., a syscall) used to manipulate the resources managed by an OS. In theory, if allocating resources fails, then an application can decide how to handle this error according to its design principles. An application can continue to execute in a way that inflicts the least damage to the users or the system's overall stability.

This paper focuses on handling a failure to allocate one category of resources—*memory*. The *kernel* manages the entirety of the memory in an OS. The kernel is defined as "the part of the system that runs in protected mode and mediates access by all user programs to the underlying hardware (e.g., CPU, keyboard, monitor, disks, network links) and software constructs (e.g., filesystem, network protocols)" [4, p. 22]. *User mode* is the counterpart to the software running inside the kernel (*kernel mode*). The scope of the user mode is limited to "programs running outside the kernel" [5, p. 298]. *This paper focuses only on user mode applications*. Applications such as a browser, a compiler, or a text editors run in user mode. All applications distributed by application stores like Google Play for Android or App Store for iOS and iPadOS execute in user mode. From a typical user's point of view, user mode applications are mainly what they visibly interact with.

The kernel manages memory in chunks of fixed size called *pages* or *page frames* [5], [6]. The memory manager in the kernel is responsible for tasks such as accounting for memory usage, managing physical pages, paging, and swapping. All user mode applications eventually end up requesting memory from the kernel allocator.

One of the kernel's responsibilities is to ensure the system's overall stability The kernel needs to track the per-process memory allocations as part of this requirement. If the amount of free physical pages becomes too small, then this threatens the stability of an entire OS. To alleviate this situation, the kernel can terminate user mode processes to release some of the pages.

The consequences of errors between kernel mode and user mode are different. The damage caused by incorrect memory management in user mode is, in most cases, limited to the scope of a single process. However, a similar mistake in kernel mode (e.g., in a device driver) has dire consequences, resulting in a kernel panic [7, p. 18] or a bug check [8] (more commonly known as BSoD or Blue Screen of Death). As a result, an entire system becomes unusable.

B. The purpose of the OOM killer

The OOM killer is a built-in facility in some of the OSs responsible for preemptively monitoring the memory usage for an entire OS.[1,2,3] The logic to decide what processes to terminate is based on heuristics such as the type of application, its priority, its memory usage, and a variety of other implementation-specific details. The OOM killer uses approaches like monitoring specific watermarks for the number of available and used physical pages of memory, triggering paging in the kernel, flushing internal caches if needed, and killing processes as a last resort. The OOM killer operates within the scope of an individual process. The necessity to employ an OOM killer is dictated by the fact that an OS cannot assume that each user mode application manages memory efficiently and does not have memory leaks. Utilizing the OOM killer enables an OS to continue providing services for all the processes while sacrificing a few.

[1]https://lwn.net/Kernel/Index/#OOM_killer

[2]https://engineering.fb.com/2018/07/19/production-engineering/oomd/

[3]https://source.android.com/devices/tech/perf/lmkd

C. Behavioral differences between OSs

The general principles of memory management in modern OSs are well-document in literature [8], [9], [10], [11], [12], [13]. However, different OSs may use conceptually contrasting approaches to memory management that influence the application's design. For example, we enumerate critical differences between how kernels in Linux (foundation for Android), iOS, and Windows approach memory management.

- Linux uses a concept called *overcommitting* [2]. Overcommitting means that when memory is allegedly allocated, a particular memory region is not yet reserved for the application's use. The actual allocation of memory happens when the allocated pages are modified. During the write operation, a page access fault is triggered, and only then will kernel attempt to allocate memory. Linux utilizes the OOM killer by default.
- iOS does not use either *paging* or *swapping* [14]. Paging is an optimization technique to temporarily store the contents of memory on a disk and reload it as needed. As a result, iOS can only utilize as much memory as physically available. Memory is one of the most precious resources on iOS. That design decision forces the iOS OOM killer to aggressively terminate applications that use too much memory.
- Windows, on the other hand, does not use overcommitting and does not have an OOM killer. The NT kernel was designed [8], [15] to be robust against user mode applications requesting "too much memory." An application can assume that when a pointer to a region of memory is returned, memory will be available.

We can see that behavior at the conceptual level between different OSs varies significantly. These differences, in turn, can have implications on the design assumptions of applications that are intended to run cross-platform.

D. Causes for an OOM condition

An OS not having enough memory to complete the desired operation can either be a permanent or a temporary condition. For example, if a service has a sudden spike in the number of requests it must process, this condition can be transient. Once the number of requests decreases, the memory consumption will reduce as well. If during that time, either the service or other applications running in parallel use various mitigation to temporarily deal with memory pressure, then they can continue to execute.

If an application has a consistent memory leak (e.g., a cache that is not bounded or allocations that are never released), then without killing the process, the OS will eventually run out of memory. If the leak is in a user mode process, the memory can be eventually reclaimed by terminating the process(es). If the leak is in the kernel itself (e.g., in an I/O manager code), then the only solution to reclaim the memory is to reboot the system.

E. Strategies to handle the OOM *conditions*

The standard guidance for engineers is to check the result of each call to one of the functions related to memory allocation and assume that they can fail [1], [2], [3]. This rationale assumes that when memory is unavailable, the allocator will return a value that signifies that the memory request failed. If memory allocation fails, the engineer can do any of the following:

1) Clean up and return: The function needs to release the resources already allocated at its scope, roll back the changes in the global state, and return the appropriate error code or an exception to the caller. The central assumption here is that the caller will handle the error and perform similar actions. Eventually, the user or a top-level caller is communicated the reason why an operation failed. The primary intent behind this strategy is to return the system to a stable state and ensure continued execution [3, p. 49].

2) Log an error message and exit the current process: The basic pattern for this approach is displayed in Listing 1.

Listing 1
HANDLING AN OOM CONDITION IN C.

```
void *p = malloc(N);

if (!p) {
    perror("OOM");
    exit(EXIT_FAILURE);
}
```

The rationale for this behavior is that if no more memory can be allocated, further use of the application is undesirable. The best course of action is to record what happened and hope restarting the application will avoid the problem next time. The application can also try to flush the pending changes to the disk, save the current state, or perform other actions to help with future recovery. There are no guarantees, however, that any actions, including logging the error message itself (that may require allocating memory), will succeed.

3) Release and retry: The application can perform a last-minute attempt to free any resources available (e.g., release memory allocated for an internal cache) and then retry allocating memory. The reasoning is that if the application stores many objects in the memory, then the cache can be populated later, and the freed memory may help the application continue executing.

4) See no evil, hear no evil, speak no evil: Approaches like *ignoring the problems related to resource management* [3, p. 223], most famously applied to earlier versions of UNIX [16], are also possible. Some of the anecdotal reasoning we have heard in the industry is based on the cost of writing reliable code versus the probability and consequences of a crash in non-critical user mode applications.

F. Strategies to anticipate the OOM *conditions*

We have observed various techniques that applications use to avoid getting into the OOM situation in the first place. One of the strategies involves *periodically polling the available* OS *memory* (either a percentage or a fixed size). Based on the amount of available memory, an application will preemptively take various steps to reduce the possibility of the OOM. However, modern OSs utilize concepts such as *cgroups* in Linux [17] and *job* objects in Windows [8], [15]. Those constructs allow controlling one or more processes as a group. For example, a creator of a job object can specify how much memory a single process can allocate or how much I/O the process can perform. In that state, an application can no longer use data about the global state to make correct decisions about memory availability.

Subscribing to low-memory notifications is another way to be notified about memory pressure. Most modern OSs enable applications to react to a situation when the number of free pages falls under a certain threshold. An application can then try to release resources allocated by it and hope that the resulting impact on the amount of available memory is significant enough for the OOM killer not to terminate the process. However, there are no guarantees that any effort performed by an application at this stage will be sufficient to avoid termination.

Anticipatory Memory Allocation [18] is a technique used to make kernel robust to memory-allocation failures. Similar logic can be applied to the user mode applications. Applications can preemptively estimate how much memory they need and attempt to pre-allocate this amount during the startup. For non-critical applications, this approach is very wasteful because most of the memory will be hoarded and not used. Similar methods use memory allocation rate as a predictor [19].

Attempt to *outsmart the* OOM *killer* is another possibility. The source code for kernels of OSs such as various distributions of Linux or FreeBSD is public. The inner workings of the OOM killer can be inferred from that source code. Even though iOS is a closed-source OS, the internals of how Darwin and Mach kernels are implemented are also documented to some degree [12], [14]. *An application can reverse engineer the algorithm the* OOM *killer uses to avoid being terminated*. However, each OS can change both the design and implementation of the OOM killer with each new release or an update, rendering all preemptive measures an application has put in place obsolete.

III. EMPIRICAL FINDINGS

Listing 2 contains the essential part of a program[4] we use on different OSs to gather empirical data about what happens

[4]https://figshare.com/s/8ed507efe7d4ed0f6480

when an application tries to continue allocating memory without releasing it. The algorithm allocates a fixed size of memory, dirties the allocated pages by explicitly writing into them post-allocation, and returns an error when memory no longer can be allocated. The extra step of dirtying the pages is necessary to prevent the optimizations related to overcommitting and, in some cases, compilers optimizing away the allocation code.

Listing 2
ALGORITHM TO CONTINOUSLY CONSUME MEMORY.

```
/* Any size can be used here. */
unsigned alloc_size = 1024 * 1024 * 10;

while (1) {
    void *p = malloc(alloc_size);

    if (!p) {
        perror("OOM!");
        exit(EXIT_FAILURE);
    }

    /* Dirty the allocated pages. */
    memset(p, 'X', alloc_size);
}
```

We execute this application on Android, FreeBSD, iOS, Linux, macOS, and Windows. Both as a single instance and multiple copies in parallel with varying allocation sizes. We observe distinct types of behaviors: (a) OOM killer terminates only the offending application because it consumes most of the memory, (b) OOM killer terminates a set of processes (not necessarily including the offender) based on its heuristic (e.g., background versus foreground, priority, recent usage), (c) global alerts from an OS indicating lack of memory that result in a user being presented with a choice of what process to manually terminate, and (d) *malloc()* returns `NULL` when an OOM condition happens.

This behavior matches what we expect based on inspecting the source code for various implementations of an OOM killer and existing documentation. Samples of the output from various OSs are displayed in Listing 3.

Listing 3
SAMPLES OF BEHAVIOR ON VARIOUS OSS.

```
# Microsoft Windows 10.0.22000.675
C:\Temp>oom.exe
OOM!: Not enough space

# Ubuntu 20.04.3 LTS (Focal Fossa)
$ ./oom
Killed

# macOS Monterey Version 12.3.1
$ ./oom
zsh: killed     ./oom
```

Our experiments are run with default settings. We do not modify any parameters specific to overcommitting, process-related quotas, or use custom user mode memory managers. Default kernels are used for all the OSs.

Out of all the OSs we experiment with, in case of an OOM condition, *malloc()* returns `NULL` only on Windows. On all the other OSs, a set of processes consuming most of the memory (or meeting the criteria a particular OOM killer uses) are terminated by the OOM killer first.

IV. DISCUSSION AND IMPLICATIONS

A. Implications of current state in memory management

The problem of "malloc() never returns NULL," and shortcomings of the OOM killer have been known since it was introduced to Linux [20], [21]. Current historical assumptions about handling the OOM conditions have a variety of adverse side-effects when it comes to application reliability.

1) Lack of ability to handle errors: When writing code, engineers assume the possibility of recovering and executing of error-handling code. For most of the popular OSs, this assumption is incorrect. In low-memory conditions, an application may be killed by the OOM killer using `SIGKILL`. `SIGKILL` (colloquially known as `kill -9`) is a signal that causes an application's immediate termination. Because the application cannot handle, intercept, or block `SIGKILL`, *an application must assume that it can be terminated at any moment, with or without a cause*. An application may never have a chance to handle an OOM condition in its code. As a result of sudden termination, an application may leave behind various residues. For example, an application could fail not to clean up the temporary cache of files on a disk, not release cross-process synchronization primitives, or even corrupt data. Therefore, if an application is designed under the assumption that it will have a chance to handle an OOM condition, then it needs to be redesigned to match the current reality.

2) Lack of control over the execution environment: In general, most applications do not fully control their execution environment. That is especially true for mobile applications on Android or iOS, where OS "sandboxes" the application. The OS will prevent an application from making changes to the device and even querying the data about other applications. Potential workarounds to avoid being terminated, such as modifying the OOM killer settings, are not possible by design.

3) Lack of fairness: Ensuring a system's stability is not an entirely fair process. An application must not be the most significant memory consumer to be killed. Another process of a higher priority may be using more memory, and the application may end up on the kill-list regardless. Alternatively, it may be one of the many applications that are terminated to ensure that the OS can continue providing services. Even if an application releases all the memory possible as a response to a low-memory notification and behaves as a "good citizen" in a specific ecosystem, it can still be terminated.

4) Maintenance costs: An application's code base will contain error-handling code that is never executed. That extra code needs to be maintained and tested. The presence of extra error detection and recovery code decreases the clarity of the code base and its readability [22]. The error-handling code

is shown to be a "substantial source of faults in systems code" [23]. Other negative side-effects of extra code include increases in the application's size. Application size, in turn, is one of the factors that need to be tightly controlled on mobile OSs such as Android or iOS [24]. From a testing point of view, additional execution paths will require developing more test cases and effort to reach higher code coverage.

B. Discussion points

1) Overcommitting: For server OSs, such as different distributions of Linux and BSD, there are several options to control an OS's behavior. An organization that deploys applications internally can either build a custom kernel or control different settings related to overcommitting. For example, the Linux kernel can be configured to disable overcommitting. One of the risk factors with this approach is that even if a particular application is designed to handle the OOM events properly, the behavior of all the other applications is unknown.

2) Multiple platforms: If an application's code is intended to run cross-platform on OSs without the OOM killer, it must handle the possible failures to allocate memory. Ideally, a version compiled for each platform will be optimal, i.e., no unnecessary checks for allocation failures if an OOM killer is present on a target OS. However, there is a downside to initially omitting the error-handling code. An application that is originally designed to run only on one OS and later ported to a different OS will require revisiting all the instances in the code base where memory allocation is performed.

3) Exclusion from OOM *killer:* There is a possibility that applications can exclude themselves from being killed by the OOM killer. However, that option is not available for applications installed through an official mobile application deployment platform such as Apple Store or Google Play Store. For example, in Linux, an ability to exclude a specific process from being killed may be available only for privileged processes depending on the kernel version. Enabling this approach also introduces the "what if everyone did that" dimension to the OOM management issue, rendering the OOM killer ineffective.

4) Silent OOM *killer makes debugging time-consuming:* Our observations from the industry indicate that users have a poor experience when the OOM killer does its job. We had to debug a multitude of issues when under low-memory conditions, a process experienced a sudden termination. The actions of the OOM killer result in a sudden "application death" caused by `SIGKILL`. However, for the user, there is no clear indication of why the application was terminated. This problem is well-known in the Linux community.[5] Anecdotal evidence from users after enabling *systemd-oomd* by default on Ubuntu Desktop supports this observation.[6]

C. Recommendations

We recommend that engineers adopt the following guiding principles when designing their applications:

1) *On* OS*s that utilize an* OOM *killer, design applications with an assumption that they can be terminated at any time without having a chance to react*. This concept is not new. The desire for an application to gracefully recover after a crash has been advocated for in the past [25]. Some applications use patterns such as a global exception handler for unhandled exceptions. The code that is part of a global exception handler can perform any necessary actions before termination. However, global exception handlers will not execute when the OOM killer terminates a process.
2) *On* OS*s where an application can subscribe to low-memory notification events, an application should assume that its termination is imminent if notified*. The best-case scenario in such situations is to treat this notification as an opportunity to either trigger the controlled shutdown sequence or execute a best effort to avoid conditions that would prevent the application's restart. An application should try to commit pending changes, synchronize its in-memory data structures to the disk, and possibly, as a last-effort attempt, try to release as much memory as possible.
3) *If an application is intended to be deployed on* OS*s where the application can handle an* OOM *condition, then it should use a consistent design pattern such as "clean up and return" everywhere*. Utilizing a consistent design pattern where each function is responsible for error-handling and cleanup is a design strategy that requires engineers to exercise this practice meticulously. However, unless all the dependencies follow a similar approach, this solves only a part of the problem. Given the engineering cost, we recommend this approach for critical user mode applications such as daemons or services, depending on the nomenclature a particular OS uses.
4) *If the design of an application supports recovery in case of sudden termination, then we recommend that for memory management, the application switch over to functions such as xmalloc(),*[7] *xrealloc(), and xfree() or their equivalents*. These functions are guaranteed to either return successfully or terminate the application with an appropriate error message. Using the "x-family memory allocation functions" enables the application to omit the error-handling code and simplify the error-handling strategy [22].
5) *If an application is developed in C++, the standard library enables interception of a situation when the* `new` *operator cannot allocate memory*. If *malloc()* will return `NULL`, an application can choose how to act under the OOM conditions in a central location by setting a custom *new-handler*.[8] The handler can try to release some memory and retry the allocation, terminate, or perform a different action depending on the application's design.

[5]https://lwn.net/Articles/894546/

[6]https://lists.ubuntu.com/archives/ubuntu-devel/2022-June/042116.html

[7]https://www.freebsd.org/cgi/man.cgi?query=xmalloc

[8]https://en.cppreference.com/w/cpp/memory/new/set_new_handler

V. Conclusions and future work

Universally checking the result of a request to allocate memory has been a standard practice for decades. Our recommendation to ignore that guidance on a subset of OSs is clearly contrarian. However, software development practices need to adapt to a new reality. That new reality means, for example, in the case of popular mobile OSs such as Android and iOS, an application is not in control of what happens in case of an OOM event. The typical desktop applications that execute in non-administrative mode have the same limitations. They cannot change the OS settings, query the details about the memory usage of other applications, and cannot circumvent an official OOM killer to prolong their existence. *As a result, all the code that is supposed to execute when an* OOM *condition happens will never run. Therefore, there is no reason for that code to be present*.

One topic for the future work we intend to pursue is the effectiveness of low-memory notifications on OSs that enable them. We want to study (a) how many and what types of applications use those mechanisms, (b) what actions do they perform (e.g., is the memory being released or an event is just logged), (c) how efficient those actions are (e.g., what percentage of memory that an application is responsible for is released), and (d) what is the impact of those actions (e.g., in what percentage of cases the OOM killer will let an application continue to execute).

Another subject we are interested in studying is engineers' belief systems about memory management. Based on our observations, the beliefs depend on the abstraction level the engineers work at. Engineers working lower in the stack (e.g., compilers, kernel, systems software in general) tend to be more cognizant of the consequences of memory allocation failures. They believe in doing everything possible to prolong the application's lifetime. We are interested in studying if there is validity to our observations across the industry.

VI. Threats to validity

Like any other study, the results we present in this paper are subject to specific categories of threats [26, p. 222–223].

Threats to *external validity* are related to application of our findings in a different context. There are a variety of OSs in existence. Our experiments were run only on a subset of OSs, albeit the most popular ones. The behavior of various kernels and OOM killers is constantly evolving. Our conclusions are drawn only from a sample of available data. For commercial OSs, we rely on publicly accessible information [8], [12], [14], [15] about how their kernel behaves. However, that logic can change during any subsequent releases.

One threat to *conclusion validity* is related to the fact that our reasoning is drawn mainly from our experiences with the development of commercial system software. That implies a bias towards optimizing certain characteristics of software (e.g., performance cost, presence of unnecessary lines of code) and making trade-offs differently than, for example, in the case of open-source software.

References

[1] S. McConnell, *Code Complete*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2004.

[2] R. Love, *Linux System Programming*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, May 2013.

[3] J. Noble and C. Weir, *Small Memory Software*, ser. Software Patterns Series. Boston, MA, USA: Addison Wesley, Nov. 2000.

[4] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Upper Saddle River, NJ, USA: Addison Wesley, 2015.

[5] A. S. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

[6] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. Sebastopol, CA, USA: O'Reilly, 2003.

[7] M. Beck, Ed., *Linux Kernel Internals*, 2nd ed. Boston, MA, USA: Addison-Wesley, 1998.

[8] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals*, 6th ed. Redmond, WA, USA: Microsoft Press, 2012, OCLC: ocn753301527.

[9] T. Anderson and M. Dahlin, *Operating Systems: Principles and Practice*, 2nd ed. Austin, TX, USA: Recursive Books, 2014.

[10] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

[11] W. Stallings, *Operating Systems: Internals and Design Principles*, 6th ed. Upper Saddle River, NJ, USA: Pearson/Prentice Hall, 2009.

[12] A. Singh, *Mac OS X Internals: a Systems Approach*. Boston, MA, USA: Addison-Wesley Professional, 2016, OCLC: 1005337597.

[13] R. Love, *Linux Kernel Development*, 2nd ed. Indianapolis, IN, USA: Novell Press, 2005.

[14] J. Levin, **OS internals. Volume 1: User space*, 2nd ed. Edison, NJ, USA: Technologeeks.com, 2017.

[15] J. Richter and C. Nasarre, *Windows via C/C++*, 5th ed. Redmond, WA, USA: Microsoft Press, Nov. 2007.

[16] T. V. Vleck. (1993, Mar) Unix and Multics. [Online]. Available: https://www.multicians.org/unix.html

[17] S. M. Jain, *Linux Containers and Virtualization: A Kernel Perspective*. Berkeley, CA, USA: Apress, 2020. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-6283-2

[18] S. Sundararaman, Y. Zhang, S. Subramanian, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Making the common case the only case with anticipatory memory allocation," *ACM Transactions on Storage*, vol. 7, no. 4, pp. 13:1–13:24, Feb. 2012. [Online]. Available: https://doi.org/10.1145/2078861.2078863

[19] G. Nakagawa, H. Kawata, and S. Oikawa, "Out of memory prevention based on memory allocation rate," in *2015 Third International Symposium on Computing and Networking (CANDAR)*, 2015, pp. 566–570. [Online]. Available: https://doi.org/10.1109/CANDAR.2015.41

[20] Y. Jang, "Avoiding OOM on Embedded Linux," Mountain View, CA, USA, Apr. 2008, CELF Embedded Linux Conference. [Online]. Available: https://elinux.org/images/a/a3/CELF_AvoidOOM.pdf

[21] P. Patare and V. K. Govindan, "Efficient handling of low memory situations in Linux," *International Journal of Engineering Research and Technology*, vol. 4, 2015. [Online]. Available: https://www.ijert.org/research/efficient-handling-of-low-memory-situations-in-linux-IJERTV4IS020176.pdf

[22] G. J. Holzmann, "Code evasion," *IEEE Software*, vol. 32, no. 5, pp. 77–80, 2015. [Online]. Available: https://doi.org/10.1109/ms.2015.112

[23] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1109/DSN.2013.6575307

[24] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A survey of performance optimization for mobile applications," *IEEE Transactions on Software Engineering*, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2021.3071193

[25] G. Candea and A. Fox, "Crash-Only software," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, HI, USA: USENIX Association, May 2003. [Online]. Available: https://www.usenix.org/conference/hotos-ix/crash-only-software

[26] F. Shull, J. Singer, and D. I. K. Sjøberg, *Guide to Advanced Empirical Software Engineering*. London: Springer, 2008.