Evaluate the Fragmentation Effect of Different Heap Allocation Algorithms in Linux

Dimitrios Rentas

LIU-IDA/LITH-EX-A--15/069--SE

2015-12-18
Evaluate the Fragmentation Effect of Different Heap Allocation Algorithms in Linux

av

Dimitrios Rentas

LIU-IDA/LITH-EX-A--15/069--SE

2015-12-18

Handledare: Petru Eles
Examinator: Petru Eles
ABSTRACT

Modern application are becoming more complex and demanding in terms of resource utilization. LTE network is part of those applications. Efficient memory utilization poses a great challenge to developers. The dynamic memory allocations and de allocations over the program execution time leads to a problem called memory fragmentation, which can eventually lead the system out of memory. Currently there are many allocators that are specifically designed for dynamic memory management. This thesis contains the study and analysis of three different allocators, ptmalloc2, tcmalloc and tlsf. The goal of the thesis is the evaluation of their performance in terms of memory fragmentation and cpu execution time. The allocators are tested against a real program tracing file, which contains a sequence of allocations and deallocations captured from an executing process.
## Contents

ABSTRACT................................................................................................................................. 1

Figures .......................................................................................................................................... 7

Tables........................................................................................................................................... 9

List of Abbreviations......................................................................................................... 10

Chapter 1 Introduction ................................................................................................................. 12

Chapter 2 LTE .............................................................................................................................. 13

  2.1 Requirements and analysis of Long Term Evolution ............................................................ 13

  2.2 EPS Architecture .................................................................................................................. 14

    2.2.1 Evolved Packet Core Network ...................................................................................... 16

    2.2.2 EUTRAN, Access Network ............................................................................................ 18

  2.3 Protocol Overview ................................................................................................................ 19

    2.3.1 User Plane ..................................................................................................................... 19

    2.3.2 Control Plane ................................................................................................................ 19

    2.3.3 UE Control Signaling ................................................................................................. 20

  2.4 EnodeB Memory Usage ....................................................................................................... 20

Chapter 3 Understanding Memory Management ......................................................................... 22

  3.1 History .................................................................................................................................... 22

  3.2 Memory Hierarchy ............................................................................................................... 24

  3.3 Virtual Memory .................................................................................................................... 26

    3.3.1 Memory Access Process .............................................................................................. 26

    3.3.2 Sharing ......................................................................................................................... 27

    3.3.3 Copy on Write .............................................................................................................. 27

  3.3.4 Translation Lookaside Buffer ....................................................................................... 27

    3.3.5 Advantages and Disadvantages of Virtual Memory .................................................. 29

  3.4 Virtual Address Space Segmentation ............................................................................... 30

  3.5 Memory Allocation .............................................................................................................. 31

    3.5.1 Dynamic Memory Allocation ...................................................................................... 32

  3.6 Dynamic Memory Allocator ............................................................................................... 34

    3.6.1 Policy, Strategy and Mechanism ............................................................................... 35

    3.6.2 Allocation Mechanisms .............................................................................................. 35

  3.7 Fragmentation ........................................................................................................................ 39

    3.7.1 Causes of Fragmentation ............................................................................................ 40

    3.7.2 Quantifying Fragmentation .......................................................................................... 40
Figures
Figure 2-1 Mobile Networks Evolution [3GPP] ................................................................. 13
Figure 2-2 EPS Architecture [3GPP] .................................................................................. 15
Figure 2-3 LTE interfaces ................................................................................................. 15
Figure 2-4 eNodeB Network ............................................................................................ 18
Figure 2-5 User Plane Protocol Stack ................................................................................ 19
Figure 2-6 Control Plane Protocol Stack ......................................................................... 19
Figure 3-1 Virtual Memory Allocation Concept ............................................................... 23
Figure 3-2 Memory Hierarchy ......................................................................................... 24
Figure 3-3 TLB Translation Process ............................................................................... 28
Figure 3-4 Memory Segment ........................................................................................... 31
Figure 3-5 Memory Allocation Methods ......................................................................... 31
Figure 3-6 Segregate List Example ............................................................................... 37
Figure 3-7 Buddy Systems ............................................................................................... 38
Figure 3-8 Fragmentation ................................................................................................. 39
Figure 3-9 Quantifying Fragmentation Live Memory to Heap Size ............................... 41
Figure 4-1 Allocated Chunk ......................................................................................... 45
Figure 4-2 Free Chunk .................................................................................................... 45
Figure 4-3 Fast Bins ......................................................................................................... 47
Figure 4-4 Normal Bins ..................................................................................................... 47
Figure 4-5 Multiple Arenas ............................................................................................. 48
Figure 4-6 Ptmalloc2 allocation ...................................................................................... 50
Figure 4-7 Ptmalloc2 free ............................................................................................... 51
Figure 4-8 Tcmalloc Architecture ............................................................................... 55
Figure 4-9 Tcmalloc Spans ............................................................................................ 56
Figure 4-10 Tcmalloc Array of Spans .......................................................................... 57
Figure 4-11 Size Classes ................................................................................................. 58
Figure 4-12 Tcmalloc Central Cache .......................................................................... 59
Figure 4-13 Tcmalloc Allocation Algorithm ................................................................. 61
Figure 4-14 Tcmalloc Free Algorithm ......................................................................... 62
Figure 4-15 TLSF Allocation Algorithm .................................................................. 65
Figure 4-16 TLSF Architecture .................................................................................. 67
Figure 4-17 TLSF Free Algorithm ........................................................................... 68
Figure 5-1 Synthetic Trace ............................................................................................. 70
Figure 5-2 Ramps Pattern .............................................................................................. 71
Figure 5-3 Peaks Pattern ................................................................................................. 72
Figure 5-4 Plateau Pattern ............................................................................................. 72
Figure 5-5 Real Program Tracing ................................................................................. 73
Figure 5-6 Driver Program Overview ............................................................................ 76
Figure 5-7 Trace File Live Memory ............................................................................... 78
Figure 5-8 Size Distribution .......................................................................................... 78
Figure 6-1 First Scenario Memory Usage Over Live Memory ....................................... 81
Figure 6-2 First Scenario Execution Time .................................................................... 82
Figure 6-3 Second Scenario Memory Usage Over Live Memory ................................ 84
Figure 6-4 Second Scenario Execution Time ................................................................ 85
Figure 6-5 Third Scenario Memory Usage Over Live Memory ........................................... 87
Figure 6-6 Third Scenario Execution Time ........................................................................ 88
Figure 6-7 Fourth Scenario Memory Usage Over Live Memory ....................................... 90
Figure 6-8 Fourth Scenario Execution Time ....................................................................... 91
Figure 6-9 Fifth Scenario Memory Usage Over Live Memory .......................................... 93
Figure 6-10 Fifth Scenario Execution Time ....................................................................... 94
Tables
Table 4-1 Bins Size .............................................................................................................. 48
Table 5-1 Simulation Scenarios ........................................................................................... 76
Table 5-2 Trace File Number of malloc/free calls ............................................................... 79
Table 6-1 First Scenario Maximum Size Values and Fragmentation .................................. 80
Table 6-2 First Execution Time ............................................................................................ 83
Table 6-3 First Scenario System Calls .................................................................................. 82
Table 6-4 Second Scenario Maximum Size Values and Fragmentation .............................. 83
Table 6-5 Second Scenario Execution Time ........................................................................... 84
Table 6-6 First to Second Scenario Execution Time ............................................................. 83
Table 6-7 Second Scenario System Calls .............................................................................. 83
Table 6-8 Third Scenario Maximum Memory Size and Fragmentation ............................... 83
Table 6-9 Third Scenario Execution Time ............................................................................ 83
Table 6-10 Third Scenario System Calls .............................................................................. 83
Table 6-11 Fourth Scenario Maximum Memory Size and Fragmentation ............................ 83
Table 6-12 Fourth Scenario Execution Time ......................................................................... 90
Table 6-13 Fourth Scenario System Calls ......................................................................... 90
Table 6-14 Fifth Scenario Maximum Memory Size and Fragmentation ............................... 92
Table 6-15 Fifth Scenario Execution Time ......................................................................... 92
Table 6-16 Fifth Scenario System Calls ............................................................................. 93
Table 6-17 Pmalloc2 Profile Parameters .............................................................................. 94
Table 6-18 Pmalloc2 Profiles Fragmentation Results .......................................................... 94
Table 6-19 Pmalloc2 Profiles Execution Time ...................................................................... 95
Table 6-20 Pmalloc2 Profiles MMAP Calls .......................................................................... 95
Table 6-21 Pmalloc2 Profiles SBRK Calls ......................................................................... 96
Table 6-22 Tmalloc Environment Variables Fragmentation ................................................. 97
Table 6-23 Tmalloc Environment Variables Execution Time .............................................. 97
Table 6-24 Tmalloc Environment Variables MMAP System Calls ..................................... 98
Table 6-25 Tmalloc Environment Variables SBRK System Calls ....................................... 98
Table 6-26 TLSF Sbrk Fragmentation ................................................................................... 99
Table 6-27 TLSF Sbrk Execution Time ................................................................................ 99
Table 6-28 TLSF Mmap System Calls ............................................................................... 100
Table 6-29 TLSF Sbrk System Calls ................................................................................... 100
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>Access Stratum</td>
</tr>
<tr>
<td>BCH</td>
<td>Broadcast Channel</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CQI</td>
<td>Channel Quality Indicator</td>
</tr>
<tr>
<td>DMM</td>
<td>Dynamic Memory Management</td>
</tr>
<tr>
<td>DTCH</td>
<td>Dedicated Traffic Channel</td>
</tr>
<tr>
<td>EPC</td>
<td>Evolved Packet Core</td>
</tr>
<tr>
<td>EPS</td>
<td>Evolved Packet System</td>
</tr>
<tr>
<td>EUTRAN</td>
<td>Extended Universal Terrestrial Radio Access Network</td>
</tr>
<tr>
<td>FLI</td>
<td>First Level Index</td>
</tr>
<tr>
<td>GBR</td>
<td>Minimum Guaranteed Bit Rate</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile</td>
</tr>
<tr>
<td>HSPA</td>
<td>High Speed Packet Access</td>
</tr>
<tr>
<td>HSS</td>
<td>Home Subscriber Server</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MBMS</td>
<td>Multimedia Broadcast and Multicast Services</td>
</tr>
<tr>
<td>MME</td>
<td>Mobility Management Entity</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>NAS</td>
<td>Non-Access Stratum</td>
</tr>
<tr>
<td>PCEF</td>
<td>Policy Control Enforcement Function</td>
</tr>
<tr>
<td>PCRF</td>
<td>Policy Control and Changing Rules Function</td>
</tr>
<tr>
<td>PDN</td>
<td>Packet Data Network</td>
</tr>
<tr>
<td>PGW</td>
<td>PDN Gateway</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RLC</td>
<td>Radio Link Control</td>
</tr>
<tr>
<td>RRC</td>
<td>Radio Resource Control</td>
</tr>
<tr>
<td>RSS</td>
<td>Resident Set Size</td>
</tr>
<tr>
<td>SAE</td>
<td>System Architecture Evolution</td>
</tr>
<tr>
<td>SGW</td>
<td>Serving Gateway</td>
</tr>
<tr>
<td>SLI</td>
<td>Secondary Level Index</td>
</tr>
<tr>
<td>TFT</td>
<td>Traffic Flow Templates</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>UE</td>
<td>User Equipment</td>
</tr>
<tr>
<td>UL-SCH</td>
<td>Uplink Shared Channel</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Memory</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

Long Term Evolution (LTE) Networks is the latest standard in the mobile networks technologies hierarchy GSM/EDGE and UMTS/HSPA. The requirements for fast mobile, broadband access were increased exponentially during the last decades and an even larger and even more significant development is expected in the future. It is guided from the unlimited features in services and internet applications and from the development of mobile equipment and portable devices. To fulfill these demands for the large data rates in mobile networks, the International Telecommunication Union (ITU) defined the primal levels for 4th Generation Networks evolution (4G).

The 4G network is comprised of several nodes. Each one of these nodes is itself a computer capable of making calculations and executing different processes. Nowadays, more and more people tend to utilize this network, which poses great challenges in terms of computational performance. In this thesis we will focus on the memory utilization.

The purpose of this thesis is the evaluation of the fragmentation effect and the execution time performance in applications such as the ones executing on the LTE network.

In chapter 2, the introduction to the LTE network is presented, followed by the architecture and the protocol stack analysis.

Chapter 3 introduces the memory subsystem of the modern computer architecture and analyzes the reasons why the efficient memory utilization is so important for all applications. Furthermore, a thorough analysis is made on the Dynamic Memory Management (DMM) unit while the fragmentation problem is presented.

Chapter 4 analyzes the 3 main algorithms used for dynamic memory allocation

In chapter 5, the methodology used to evaluate the fragmentation effect caused by LTE applications is introduced and analyzed.

Finally, in chapter 6 the results are presented in terms of fragmentation, execution time and system calls.
Chapter 2 LTE

During the past decades various mobile standards were introduced, from 2G to 4G. The main breakthrough of the 2G standards was the support of mobile telephony and voice applications. The 3G standards introduced the packet based data communication while the support of Internet applications had already begun. The 4G standards feature IP packet based networks and the focus is to support the demand for bandwidth demanding applications such as video on demand. Figure 2.1 illustrates the evolution of the different standards; we will not dive into details on the different standards. [1]

2.1 Requirements and analysis of Long Term Evolution

The most important requirements during the development of LTE were:

- Reduced delays, in terms of both communication establishment and transmission delays
- Increased peak data rates
- Reduced cost per bit
- Simplified network architecture
- Flexibility of spectrum usage
- Improved system capacity and coverage
- Seamless integration with existing networks
- Multi antenna support

Additionally, LTE was developed to utilize only the Packet Switched services, instead of the Circuit Switched used on earlier technologies. The main difference between the two is that Packet switching spreads the resources all over the network, while Circuit switching requires dedicated resources throughout the whole communication between two users. Additionally, Packet Switching uses the IP protocol to provide connectivity between the User Equipment (UE) and the Packet Data Network (PDN). The User Equipment is the device used by the user to access the network; it could be his mobile phone, his tablet. The Packet Data Network’s main functionality is to allocate IP address to that equipment.

The term LTE is mainly connected with the evolution of the radio access through the Evolved UTRAN. However, there are many improvements to the non Radio network aspects. These improvements introduce the System Architecture Evolution (SAE) which includes the Evolved Packet Core (EPC) network. SAE and LTE comprise the Evolved Packet System (EPS).

The communication between the two parts is made using the EPS bearers. A bearer is an IP packet flow with a predefined Quality of Service. The application requirements guide the EPC and the EUTRAN to establish those bearers. Improved Quality of Service management introduced the native support of VoIP calls.

**2.2 EPS Architecture**

EPS gives user the ability to access the Internet using IP connectivity. As already mentioned, bearers are used per QoS flow. Each user can establish several bearers, since he can have a VoIP call and at the same time download a file from an FTP server. Those two applications have different QoS requirements.

Someone can realize that the network being used is comprised by several parts. At a higher abstraction the EPC and the EUTRAN are the main elements. Each one of them consists of different elements. While the EPC consists of several nodes the EPC
is made up of only one node, the evolved NodeB, which connects to the UEs. Each one of the different network elements are connected to each other using standardized interfaces in order to allow different vendors to operate on them.

Figure 2.2 shows the EPS architecture and we can see that the overall architecture supports not only LTE but older technologies also.[14]

The interfaces connecting the different parts are presented on figure 2.3. We can notice that eNodeB is the frontier that connects a UE with the rest of the elements.
2.2.1 Evolved Packet Core Network

This network is responsible for controlling the UE and handling of the bearers. The main parts are:

- PDN Gateway (PGW)
- Serving Gateway (SGW)
- Mobility Management Entity (MME)

The PDN Gateway and Serving Gateway are parts of SAE GW. In addition to these elements the Network also contains other nodes and functions like the Home Subscriber Server (HSS) and the Policy Control and Changing Rules Function (PCRF). A better description of the different parts is given:[10]

- **MME.** MME is a control node which is in charge of controlling the signaling between the UE and the Core Network. The protocols running between the UE and the EPC are called Non-Access Stratum (NAS). The main functionalities the MME provides are:
  - Bearer management. This includes establishment, maintenance and release of the bearers. These functions are handled by the session protocol layer in the MME.
  - Connection management. This includes the establishment and security between the network and the UE. It is served by the NAS protocol layer.
  - Inter-networking with other networks. This includes handing over voice calls to legacy networks.

- **PDN Gateway.** The PGW is responsible for providing with IP address the UE. Additionally, this forces the QoS according to the PCRF. It is in charge of filtering the user’s IP packet to its corresponding QoS based bearer. This separation is based on Traffic Flow Templates (TFT), which uses IP header information to distinguish packets such as VoIP from web browsing.

- **PCRF.** The PCRF is responsible of providing policy control and controlling the data flow in the Policy Control Enforcement Function (PCEF), which is part of the PGW. PCRF provides the QoS class identifier that decides how a certain stream of data will be treated by the PCEF, making sure that this decision is made based on the user’s subscription profile.
• **Serving Gateway.** The SGW is in charge of transferring user’s IP packets and serves as the local mobility anchor for the already established bearers when the user moves between eNodeBs. It has also the responsibility to keep information about the bearers while and when the UE is in idle state. While in idle state it temporarily buffers downlink data and at the same time the MME issues a paging procedure to check whether the UE is still in idle state in order to reestablish the established bearers.

• **Home Subscriber Server.** The HSS contains user’s subscription data such as the QoS profile and any access restriction for roaming. It also contains information on which PDN the user can connect and on which MME the user is currently connected to.

### 2.2.1.1 NAS Procedures

NAS is a protocol layer in the MME protocol stack. When a UE turns on and attaches to the network, MME creates a UE context, assigning the UE with a temporary identity number. UE context holds information about the subscription of the user, based on the information stored in the HSS. This information is then locally stored to the MME to allow faster execution, rather than requesting the same information over and over again from the HSS.

In order to avoid congestion and to release wasted resources, when the UE is in idle state the allocated resources are released back to the eNodeB. Only the UE context and the established bearers are kept to the MME. In case the MME needs to reach the UE a paging procedure starts. A paging message is send from the MME to the eNodeBs and the eNodeBs informs the related UE, so it can resume its operation from the idle state. The stored bearers are reestablished.

Furthermore, the NAS layer is responsible for the security functions. An authentication procedure is performed between the UE and the MME, where the authentication keys are produced.
2.2.2 EUTRAN, Access Network

As already mentioned, the access network consists only of a network of eNodeBs as illustrated on figure 2.4. EnodeB is considered to be one of the most important elements in the whole network architecture, since this is in charge of connecting the users to the rest of the network.

The eNodeBs are connected to each other using the X2 interface and are connected to the MMEs using the S1 interface. The protocols run between the eNodeBs and the UE are called Access Stratum (AS) protocols.

The nodes are responsible for all radio related functions:

- Radio Resource Management. This covers all the functionalities responsible for radio bearers, such as control, mobility control etc.
- Header Compression. To improve efficient use of the radio, the IP packet header is compressed.
- Security. All information send over the radio is encrypted.
- Positioning. The eNodeB provides all the required information on how to calculate the UE’s position.
- Connectivity to the EPC. Provides the signaling towards the MME and the SGW.

All the above functionalities are integrated into the eNodeB.
2.3 Protocol Overview

In this subchapter a presentation of the protocols being used is made. There are two different protocol stacks based on whether user data are handled or radio resources are handled. The first one that is responsible for controlling user data is called **User Plane**, whether the second one is called **Control Plane**.

### 2.3.1 User Plane

A UE sends and receives IP packets encapsulated in an EPC specific protocol and tunneled between the PGW and the eNodeB. Different tunneling protocols are being used all over the different interfaces. Figure 2.5 shows the User Plane protocol stack.

![User Plane Protocol Stack](image)

**Figure 2-5 User Plane Protocol Stack [3GPP]**

### 2.3.2 Control Plane

The control plane handles radio specific functionality. Figure 2.6 illustrates the protocol stack.

![Control Plane Protocol Stack](image)

**Figure 2-6 Control Plane Protocol Stack [3GPP]**
Before we continue and analyze the UE states we need to clarify that the RRC protocol has the following responsibilities:

- Handling the information being broadcast over the S1 interface.
- All procedures from the establishment to the release of an RRC connection, including security, paging etc.
- Security activation and transfer of UE context.

### 2.3.3 UE Control Signaling

Without diving into details the UE can have the following RRC states [1]:

- **RRC_IDLE.** In this state the UE has an IP address but is not connected to the e-UTRAN/eNodeB. It is controlling a Paging Channel to detect incoming calls and based on the upper layers it may enable several power saving features.

- **RRC_CONNECTED.** In this state the UE is connected to the eNodeB. The UE can now support mobility, where it is possible to move between cells using the handover procedure. Additionally, the UE can now receive and transmit data.

### 2.4 EnodeB Memory Usage

In LTE eNodeB is one of the most important elements. It has to handle thousands of UEs at the same time at a fast pace.

The continuous allocation and deallocation of UE resources is a memory demanding task. Since the number of allocation and deallocation requests cannot be identified beforehand, *dynamic memory allocation* is used to control them.

As it will further be explained in the following chapters, this kind of memory allocation can cause problems.
Especially the continuous allocation and deallocation of small sizes stresses the memory subsystem and can leave holes in the contiguous memory space. Eventually, the node will not have contiguous free memory to serve future allocations.

This problem is called *memory fragmentation* and the rest of the thesis focuses on analyzing this problem and how to mitigate it.
Chapter 3 Understanding Memory Management

Memory is among the most basic and most essential resources available to a process i.e. a program that is being executed by a CPU in a computer. In this chapter the management of this resource is covered.

Before further analysis the concepts of physical and virtual memory should be defined.

An address generated by the CPU is a virtual memory address whereas an address that is actually available on the memory unit is a physical address. A virtual address is translated to a physical address through a memory mapping hardware unit.

3.1 History

From the early days of computing and programming the developers acknowledged that fast access to large amount of storage is hard and costly. Back in the 1950s computers had a two level storage hierarchy, the main memory and the disks. The CPU could only address the main memory in terms of physical memory addresses. Since program’s code and data need to reside in the main memory in order to be executed, the problem was that main memory was sometimes not large enough to accommodate the needs of a running process.

In order to overcome this problem, programmers had to divide their programs into blocks, called segments, and find an efficient way to schedule the exchange between the two levels. Usually, at the beginning of a program’s execution the first segment was transferred into main memory, while the rest of them were left stored on the disk and they were then sequentially transferred to the main memory replacing the ones that were no longer needed.

Is it easy to see that this task required additional planning and it was time consuming, while the programmers’ jobs were significantly affected by the correct usage of the memory. Furthermore, this task was prone to errors, since the programmer was solely responsible for it. In 1961 Fotheringham introduced a method of assigning that task to the Operating System[12] (the program that resides between the user and the
hardware and is responsible to correctly allocate resources based on user’s needs) without the help of any control mechanism. This method is now known as *Virtual Memory*. The main goal and innovation of that method was the distinction between addresses (virtual) and memory allocation (physical). This allowed developers to refer only to virtual addresses while the Operating System along with the hardware was tasked to dynamically translate a virtual address to its physical location in main memory. Figure 3.1 presents an abstract view of this concept.

![Figure 3-1 Virtual Memory Allocation Concept](image.png)

To achieve this functionality these three inventions were introduced:

a) Development of a hardware that can automatically translate each virtual address generated by the CPU to its current physical memory location. This hardware is nowadays called *Memory Management Unit* (MMU).

b) An interrupt handler mechanism that could determine when to move a missing segment of data from the secondary storage to the main memory. This handler is controlled by the MMU and is called *demand paging*.

c) A replacement algorithm to detect and move the least useful segments back to the secondary memory.

Using the Virtual Memory, concept part of the program is located to the Hard Drive, while the currently essential parts are moved to the Main Memory for execution. This
not only relieved the programmers of the complicated task of memory managements, it also allowed them to start writing larger programs.

### 3.2 Memory Hierarchy

The invention of virtual memory increased the degree of multiprogramming, since several programs could be simultaneously loaded into the main memory.

Having additional available hard disk storage gave programmers the freedom to write larger programs, without worrying about the main memory size, since most of the segments of their program reside on the disk and only the essential ones are fed to the main memory.

It is now worth mentioning that one of the challenges of computing is to provide CPU with necessary data as quickly as possible. The mapping between virtual and physical address is relatively expensive in terms of performance. Accessing segments in disk and swapping them to main memory has significant performance penalty.

Over the years, many methods have been invented to overcome this problem. The most important method is the addition of new memory hierarchy levels to the two old ones. Fast cache storages are placed between the CPU and the main memory. They store the data of memory locations that the CPU uses repeatedly or is expected to use very soon. The modern hierarchy is shown in figure 3.2

![Figure 3-2 Memory Hierarchy](image)
Closest to the CPU are the general purpose registers. The CPU can access those registers within 1 CPU clock cycle. Modern CPUs contain at least two levels of caches, while there are some high end models containing three levels. The L1 cache is usually very high speed static RAM (RAM that retains its data as long as power is supplied) with access latency of up to 5 ns and size between 16 and 32 Kbytes. L2 cache is made of dynamic RAM (the data are lost after the power supply cut) with latency up to 10 ns and size up to 2 Mbytes. Dynamic RAM requires being refreshed periodically as opposed to static RAM. All caches are located within the CPU. The next level is the Main Memory or RAM with access latency of about 50 ns, the size in modern systems can be several Gbytes while the throughput can be up to 10GB/s. We can point out that so far the hierarchy is balanced with single order of magnitude increased latency in each level down, while providing several orders of magnitude in terms of capacity. However, there is a large gap in both latency and performance between RAM and Hard drive. Modern Hard Drives can be several hundred Gbytes large but require hundreds of milli-seconds seek time to access the required data while the throughput is about 100 MB/s.

Additionally, it is worth noting that the faster the access, the more expensive it is to implement it. This is the reason why large capacity, slow hard drive is not so expensive compared to smaller capacity faster RAM.

It is now clear, that accessing the hard drive affects the overall performance of the system. This is the reason that many methods and techniques were developed to avoid excessive disk access. For instance, there are sophisticated branch prediction techniques that try to predict which data will be used in the future and swap them to the caches. So, even though the usage of Virtual Memory allowed the programmers to write larger programmers, the problem of excessive disk accesses needed to move data back and forth from main memory to disk, still has to be addressed. However, the main goal of Virtual Memory is to hide all these details from the programmer.
3.3 Virtual Memory

The main concept of Virtual Memory is that each program has its own address space assigned by the operating system’s kernel divided into pages providing the basic abstraction between the physical memory and the user. The Kernel’s pages are of variable or fixed size while each page has linear address range. Those pages are mapped to the main memory by the MMU; physical addresses are also divided into page frames. However, not all pages of a program need to be mapped for their execution or their mapping is postponed until they are actually needed. Pages and frames usually have the same size, 4kB for 32bit architecture and 8kB for 64bit architecture. Pages are either in a valid or invalid state. A page in a valid state indicates a page that is currently associated with a page frame in main memory or in a secondary backing storage as the hard drive. A page in an invalid state is not associated with anything and represents an unallocated piece of the address space. A program accessing an invalid page leads to segmentation fault.[12]

3.3.1 Memory Access Process

Every time a program needs to access a virtual address, the MMU checks if the requested page is mapped to the main memory. This is achieved by referencing the page table. This table is created and maintained by the operating system and is responsible for storing the mappings of virtual addresses to the physical addresses. The page table is organized into page table entries, where each entry maintains information for only one page at a time. From each page table entry the operating system must be able to get the following information:

- The virtual page number
- Whether the page is valid or invalid.
- The page’s translation information, meaning its location in the physical memory.
- Whether the page was recently written or accessed.

If the page exists to the page table (mapped) the contents of the address is returned to the program. If it is not mapped, the CPU raises an interruption called page fault, the kernel intervenes and transparently transfers pages from the disk to the main memory.
Since, the amount of virtual address space is larger than the physical address space it is up to the kernel to page out unused pages to the secondary storage.

### 3.3.2 Sharing

It is possible that different virtual address pages from different processes are mapped to the same physical address. This allows different virtual address spaces to share physical pages. The shared pages can be readable, writable, or read only. Sharing has also improved the multithreading environments, since the parts that are the same for the programs need only to be placed in one place of the memory, e.g. shared libraries.

### 3.3.3 Copy on Write

When a process needs to write to a sharable and writable physical page, two things can happen.

- The kernel allows the changes and all processes that share this page get notified and updated.
- The MMU may raise an exception allowing the kernel to create a new copy of the sharable page for the writing process. The process performs the writing function to that newly create page. This is called Copy on Write.

In general, processes are allowed to read from shared pages, without any restriction. However, when a process needs to write to that shared page, it receives a unique copy of that page.

### 3.3.4 Translation Lookaside Buffer

During the last years address space have grown larger and larger. The usage of a single page table is not efficient anymore, because the larger size of it would make it hard to manage leading to increased access latency. In order to address this problem, the Translation Lookaside Buffer (TLB) was developed. In general, it is a cached table, containing recently used entries of the page table. It is used to avoid the necessity of accessing the main memory every time a virtual address is mapped and as a result to improve the virtual address translation process. The most recent virtual – to
physical mappings are stored and most likely the next virtual address to be mapped will need the same page entry. Every time a process needs to reference the memory:

- First search the TLB if the page is stored. If it is, the reference proceeds really fast.
- If the TLB has no info about the page, it must go through the page table to get info and eventually to the main memory. The return page is stored to the TLB for future reference. In order to decide which page to remove to accommodate the newly created entry, different replacement policies have been developed e.g. least recently used, which tries to take advantage of memory locality (used memory addresses are most likely to request access for addresses that are located near to them).

3.3.4.1 Translation
Each address generated by the process is a virtual address and consists of at least two parts. The first part indicates the page number and the second part provides an offset, to show the exact address in that page. An abstraction of the translation process is shown in figure 3.3
We can note that first the TLB is searched for a mapping, which is small in size and really fast to search. If that mapping is found, it is translated to physical address; this is called TLB Hit.

On the other hand, TLB Miss is when the page table is searched to find the mapping. Page table searching is slower, since it requires memory access, which translates to additional latency.

An improvement of the previous process is increasing the size of the pages, thus the amount of stored information into the TLB. This is called Huge Pages.

### 3.3.5 Advantages and Disadvantages of Virtual Memory

There are many benefits of using Virtual Memory. Some of them are:

- **Automatic Storage Allocation.** Virtual Memory makes it possible to run a program whose memory exceeds the physical memory. This relieves the programmer from worrying about the size of a program and the size of the available physical memory. Secondary storage is used to store the program and only the currently essential parts are moved to the main memory, this is called swapping.

- **Protection.** Virtual Memory hardware can enforce protection by introducing access privileges to the pages. It can mark pages as read only, to prevent unwanted changes.

- **Memory Sharing.** Physical pages can be shared among several processes, saving space. This is especially useful in sharing libraries.

- **Demand Paging.** The segment needed to be executed is only moved to main memory when it is actually referenced.

On the other hand the disadvantages are:

- **Overhead.** The translation between virtual and physical address adds overhead to the system.

- **To implement it,** additional complex hardware support is needed.

- **Program execution time is less predictable.** In real time application this is a major problem.
3.4 Virtual Address Space Segmentation

In modern operating systems, the kernel arranges process memory space into variable size segments instead of fixed size pages, which share certain properties such as permissions. Those segments are mapped to physical memory by the process page table. Furthermore, the kernel preserves a segment for its own needs. As already explained, not all parts of the segments are mapped to physical memory. In order to protect the kernel segment from unauthorized access it is flagged as privileged code to the page table. This segment is constantly present and is mapped to the same physical memory in all processes (shared).

In this thesis our main environment is the Linux Operating System 32bit version and from now the discussion will be limited to it. Certain types of memory segments are found in every process:

- Text segment, containing the binary code currently running by the process. This segment is marked as read only.
- Initialized Data segment, containing the static and global variables initialized by the programmer.
- Uninitialized Data Segment called Block Started By Symbol (BSS), containing uninitialized static variables filled with zeros.
- The Heap segment; it has variable size and is used to store the dynamically generated data. It is added to the end of BSS and its size is controlled by the `brk` address.
- Memory Mappings segment, containing dynamic libraries and anonymous mappings.
- Stack Segment, containing the process execution stack, which grows and shrinks dynamically.
The layout is better illustrated in figure 3.4

For 32bit architecture (equating to 4GB of addressable memory), the kernel segment is 1 GB, while the rest 3GB are occupied by the rest of the segments.

3.5 Memory Allocation
There are two methods a program can use to request memory from the operating system: static and dynamic. Those methods are shown in figure 3.5
In static allocation the memory is allocated at compile time and is served by the stack segment. The address and the size of those allocations are fixed and do not vary through the program’s lifetime. Typically, stores global variables and statically allocated fixed size arrays. The lifetime of such an allocation is fixed to the scope of the program that created it.

On the other hand, in dynamic allocation the memory is allocated at run time using the `malloc` family functions and is served by the heap or the memory mappings segment. The size of the allocations can vary through the lifetime of the program e.g. if it uses dynamically created lists.

Dynamic allocation is either implicit, where the whole allocation method is managed by the programmer since he is responsible to `free` the allocated memory, in case he forgets about it there is a `memory leak`, or explicit where the program’s garbage collector is responsible for freeing the no longer used allocations. During this thesis the Dynamic Implicit allocations are examined.

### 3.5.1 Dynamic Memory Allocation

Dynamic memory allocation should provide the programmer with the means to dynamically create and destroy objects and structures. The means are provided by C language’s library interfaces:

- **malloc.** The `malloc` function allocates a block of memory of a requested size. The signature of the function is:

  ```c
  void *malloc(size_t size);
  ```

  This call will allocate a block of at least `size` bytes. In case the call is successful it will return a void pointer to a newly created memory area, if it is not successful it will request additional memory through the kernel. If it fails it will return `NULL`. The returned pointer is always void, which can be cast to the desired type of data. There are different implementations of the `malloc` call, each one targeting to different goals like fast execution time or minimize memory waste.
- **free.** The *free* function frees the allocated space. The signature of the function is:

  ```c
  void free(void *ptr);
  ```

  This call will mark the area of *ptr* as *free*, thus enabling its reuse. However, the *ptr* must be a previously allocated pointer, otherwise problems will occur.

- **calloc.** Same as *malloc*, but the returned block of memory is already initialized with zeroes. The signature is:

  ```c
  void * calloc(size_t num, size_t size);
  ```

  The *num* argument indicates the number of elements to allocate.

- **realloc.** This function reallocates memory. The signature is:

  ```c
  void * realloc(void *ptr, size_t newsize);
  ```

  In case the size is NULL, the function has the same functionality as the *malloc* function and a new block will be allocated. In case the size is smaller, the function will try to shrink the block allocated by *ptr*.

- **mmap.** This function creates an anonymous mapping. The signature is:

  ```c
  void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
  ```

  In general it is slower than a *malloc* call, because those mappings are zero filled by the kernel. Furthermore, the allocated space is always page-sized, so there might be memory waste. However, some allocators use this call for large requests. Once the allocation is freed the whole page is returned to the operating system. Typically, the programmer should not directly use this call and let the allocator manage the usage of it.

Under the hood the *sbrk* system call is used to manage the *malloc* family requests. This call shrinks and expands the size of the heap segment, which starts at the end of the BSS segment up to the *brk* address (end of heap segment introduced in figure 3-
4) In recent kernels the dynamic allocations are also served by the anonymous mappings segment, using the underlying `mmap` interface, creating anonymous memory regions.

The frequent use of system calls for variable size allocations is not beneficial, since it involves swapping between user and kernel mode which is CPU cycles expensive. Hence, the `malloc` underlying mechanism usually requests a size larger than the requested size and is responsible to handle this area only. Only when this area is out of space, an additional system call is made to request extra space.

In general, there should be a mechanism that keeps track of free memory blocks and which memory areas are allocated. This mechanism is named dynamic memory manager or dynamic memory allocator. The implementation details of different allocation algorithms are discussed in the next chapter.

### 3.6 Dynamic Memory Allocator

The dynamic memory allocator receives allocation requests to create memory blocks, and `free` requests to free those blocks. In general this software must keep information about which blocks of memory are free and which are not. The reduction of system calls is done by reusing the blocks. Furthermore, the memory allocator may join two adjacent blocks (coalesce) or split a larger block into two smaller ones for improved memory space utilization. Coalescing may happen:

- Immediate, after each `free` call tries to coalesce the block with its neighbors.
- Deferred, merge while the list is scanned or when a threshold of wasted memory is reached. Do not coalesce each time a `free` request is made.

In an abstract analysis, once the allocator receives a `free` request it pushes the memory block back to a cache so it can reuse it later on. In the event of receiving an allocation request it serves it by searching for the suitable free block from that list. In case it cannot find a free block, it requests more memory through the kernel.
3.6.1 Policy, Strategy and Mechanism

In [7], a three level model for allocator analysis was introduced, policy, strategy and mechanism. The main idea of this model is that the allocator implements a placement policy following the strategy to minimize excess wasted memory space. The placement policy instructs the allocator where in the free memory to put a requested block. The strategy takes into account the regularities in the program behavior. It basically defines the rules on which the allocation policy shall trigger. Finally, mechanism is a set of algorithms and structures used to implement the policy. The mechanism should be chosen wisely to reduce the amount of time needed by the allocator and minimize the wasted memory space. For example, a best fit policy can be implemented by searching a double linked list of available blocks and find he most suitable block size to allocate.

3.6.2 Allocation Mechanisms

There are different ways of managing and storing the free memory blocks and different allocators implementing different strategies to achieve this. However, they all follow some basic principles, lists that keep information about the free blocks. There are several algorithms that show how the lists are organized and implemented. The following sub chapters present those algorithms. [6,7]

3.6.2.1 Sequential Fit

This is the simplest of all algorithms. All the free memory blocks are kept into a single or double linked list. Usually each block is organized using the boundary tag technique introduced by Donald Knuth [4] where a block of memory keeps two value fields about its size, one at the beginning of the block and one at the end of it. This allows to easily coalescing two unused bordering blocks into a larger one.

On allocation request the list is searched to find the appropriate free block. Searching for that block is done using one of the following policies:

- First Fit. This algorithm searches through the list from the beginning until a block large enough to serve the allocation request is found. If the remaining part of the block is large enough it is placed back to the list. The problem
using this algorithm is that the large blocks in the beginning of the list tend to split into smaller ones. Those small blocks increase the list traverse time, since the algorithm has to find a large enough block to satisfy the request.

- **Next Fit.** This algorithm is identical to the previous one, with the sole difference that the search for an appropriate free block starts from the point where the previous search was satisfied. In case the search wraps around the point where it first started and have not served the request, then additional memory is requested.

- **Best Fit.** This algorithm traverses the entire list to get the smallest free block that is able to satisfy the request. This algorithm is the best in terms of memory usage, but one can imagine that the exhaustive list search has the worst time performance.

- **Worst Fit.** This algorithm searches the entire list for the largest block possible. The idea is that when the largest block is divided, the remaining parts of the block will be able to satisfy a request.

On free request the blocks are placed either on the front of the list or on the back of it. Additionally, address oriented placement policy may be used, where the freed blocks are placed sorted on their address.

### 3.6.2.2 Segregated Free Lists

In this algorithm the allocator uses an array of free lists, where each list holds blocks of a particular size. When a block is freed it is pushed back to the appropriate list. When a request is served the appropriate list is used to satisfy this request. The figure 3-6 shows an example of this algorithm. There are many variations of this algorithm based on splitting and coalescing. Additionally, the use of size classes to group similar object sizes creates an extra variation. Some of the variations are presented:

- **Simple Segregated Storage.** In this one, larger blocks of memory are not split to smaller ones while smaller sizes do not coalesce to satisfy larger sizes requests. When the list of a requested size class is empty additional memory is
requested through the kernel. Usually, up to two pages are requested every time and those pages are split and feed the lists.

- Segregated Fits. This variant, uses an array of lists, but each list holds a class of sizes. When servicing a request for a particular size the corresponding list is searched for a block large enough to satisfy the request. Usually the search is sequential and a first or next fit algorithm is used. In case a free block is not found in the appropriate free list, the algorithm keeps searching the list of the next larger size class until a fit is found. Splitting and coalescing are supported by this algorithm. However, coalescing may increase search times. The use of segregated fits is also called good fit.

![Figure 3-6 Segregate List Example](image)

### 3.6.2.3 Buddy Systems

This algorithm is a variant of segregated free lists. It supports splitting and coalescing. In an abstract overview the heap area is divided into two large areas called buddies. These areas are further split into two smaller until the requested block size is found. A block can only be split to a pair of blocks, while a block can only be coalesced with its buddy (the corresponding block at the same hierarchical level). Some of the most important variations are:

- Binary Buddies. Is the simplest and best-known algorithm. All buddy sizes are a power of two and each size is divided into two equally sized buddies. This
makes the address computation fairly simple and fast. Any requested object should be rounded up to power of two, thus leading to excess wasted space. This variation is illustrated in Figure 3-7

- Fibonacci Buddies. This variation uses a closer spacing of size classes, based on Fibonacci numbers. A block can always split into two blocks whose sizes are also part of a Fibonacci series. One disadvantage is that the split sizes are of different size, and may not be useful if the program requests blocks of the same size.

- Weighted Buddies. A different kind of size classes is used for this variation. Some size classes can be split only one way, while other can be split in two ways. It provides blocks whose sizes are power of two and three times the power of two.

3.6.2.4 Indexed Fits
Indexed fits is a more sophisticated method to store free blocks. Instead of using linked lists, an indexing data structure is used, such as a tree or a hash table, to identify the free blocks according to the placement policy.

3.6.2.5 Bitmapped Fits
Bitmapped fits is a form of indexed fits, where a bitmap is used to keep track of which parts of the heap are in use and which are free. Under this mechanism the heap is
organized in words. Bitmap is a vector of one bit flags corresponding to each word of the heap. In case a word is used the corresponding bit is set. Allocation is done by searching for available clear bits in the bitmap.

### 3.7 Fragmentation

Arbitrary allocation and de-allocation of memory blocks can lead to memory holes, which may not be used to serve future allocation requests if they are too small. This problem is called fragmentation. In general fragmentation is the inability to reuse memory that is free [8]. Traditionally it distinguished between internal and external fragmentation. Figure 3.8 illustrates the two types.

- **Internal**. Happens, when a block is allocated to hold an object, but the block is larger than the actual requested size. The excess of the block is wasted. It is called internal because the waste is inside the allocated block.

- **External**. This is a phenomenon where the continuous memory becomes divided into small pieces over the execution time. The small holes may not be usable by future allocation. Even though the total amount of requested size is available to the heap, it is scattered through the heap.

![Fragmentation Diagram](image)

We can notice that the external fragmentation blocks are locked in between the allocated blocks. The sum of the locked blocks could be used to create a larger block to serve a request. However, this is not possible and even though the amount of free
memory is there, it cannot serve a larger continuous memory request. In that case the allocator will request for more memory through the kernel. To sum up, even though the free space is there, it cannot serve a bigger request because it is not contiguous.

3.7.1 Causes of Fragmentation
In general, fragmentation is the inability to reuse memory that is free. It may occur due to the allocator’s policy choice, which may reuse the leftover memory. There may be small holes generated by splitting but their neighbors are not free so they cannot coalesce. The inability of an allocator to reuse free memory depends not only on the number and sizes of the holes, but on the future allocation and de-allocation behavior of the program. According to [7] there are two main causes for fragmentation:

- **Isolated Deaths.** The creation of free holes whose adjacent blocks are not free is crucial. To address this issue the order of adjacent allocations and when those allocations die should be evaluated. If the blocks are allocated and the die at the same time, there is no fragmentation, since they are live (occupying memory space) at the same time using contiguous space and when they die at the same time, this space is still contiguous. If an allocator could predict which objects will die at the same time, it can place them in adjacent blocks.

- **Time Varying Behavior.** Fragmentation arises from changes in the way a program uses memory. It is important to consider patterns of changing behavior. The allocator’s job is to figure out those patterns and place the right blocks to the right place.

3.7.2 Quantifying Fragmentation
It is not easy to quantify fragmentation in terms of wasted memory, since it is based on the order of the allocations and de-allocations. This thesis follows the definition of [6] where fragmentation is defined as percentage over and above the amount of live data (amount of memory requested by the program and not de-allocated yet). In that research four different methods of measuring the fragmentation were presented and are shown in figure 3-9.
The diagram illustrates memory usage over execution time. The lower line indicates the amount of memory requested by the program (live memory), while the increasing line represents the actual heap memory size occupied by the operating system.

1. The amount of memory used by the allocator relative to the amount of memory requested by the program averaged across all points in time. In figure 3-9 it is equivalent to averaging the fragmentation for each corresponding point on the upper and lower lines for the entire run of the program.
2. The amount of memory used by the allocator relative to the maximum amount of memory requested by the program at the point of maximum live memory. This corresponds to the point 1 relative to the point 2.
3. The amount of memory used by the allocator relative to the maximum amount of memory requested by the program at the point of maximal memory usage. This corresponds to the point 3 relative to the point 4.
4. The maximum amount of memory used by the allocator relative to the maximum amount of live memory. These two points do not necessarily occur at the same time in the run of the program. This corresponds to the point 3 relative to the point 2.

The final method is the most commonly used in the literature of the past decade. The thesis will follow the last method.
3.8 Summary

During this chapter the concept of Virtual Memory was analyzed while different methods of organizing the structure of the memory allocator were investigated. Also, the fragmentation problem was described and we introduced the suggested quantification method. In the next chapter the most widely used allocators will be examined.
Chapter 4 Memory Allocators

In the previous chapter the concept of memory allocation was investigated. Through this chapter the most important and commonly used allocators in practice are presented.

The user level memory allocator must scale with different number of applications under a wide variety of system configurations.

The main task of a dynamic memory allocator is to handle the dynamically generated memory requests of a process by managing the available free memory while keeping track of memory in use and free memory.

The three allocators to be discussed are:

1. Ptmalloc2
2. Tcmalloc
3. TLSF

4.1 Memory Allocator Design
The design of a good memory allocator needs to consider these factors [4]:

- Maximize Compatibility. In order to achieve a smooth transition, the allocator’s interface should follow the existing POSIX standards.
- Maximize Portability. Reliance on operating system specific system calls should be kept to bare minimum. However, the allocator should provide the means to optionally support specific operating system features.
- Minimize Space Waste. The allocator’s own data structures should be kept to minimum. Additionally, it should obtain as little memory from the system as possible, to reduce the fragmentation.
- Be Adjustable. The memory allocator behavior should cover the spectrum of all possible usage variations.
- Be Tunable. Optional features should be controlled by the users.
• Maximize Locality. Allocations that are used together should be placed near each other. This will minimize the page and cache miss.
• Maximize Debugging Features. The memory allocator should provide memory errors detection capabilities.
• Execution Time. The execution time should be as minimized as possible.

4.2 Ptmalloc2
The Ptmalloc2 allocator is currently used as the default allocator to the Linux glibc distributions and is based on the DLmalloc allocator by Doug Lea [3]. The main difference between those two is the support of multithreading programming using locks, and the use of multiple arenas/heaps. There are two types of arenas, the main arena that is managed by the dynamically allocated heap segment and multiple non-main arenas one per thread which are allocated using the mmap call. The notion of heap is changed in non-main arenas, since each one can contain several sub heaps.

4.2.1 Chunks
A memory block managed by Ptmalloc2 is called chunk. There are four different chunk types:

I. **regular chunk**, organized in bins.
II. **top chunk**, controlled directly by the arena and is the one that can be arbitrarily extended via the sbrk call.
III. **last remainder chunk**, which is never directly stored in bins and is used as a technique to improve cache hit rate.
IV. **mmaped chunk**, which is controlled by the mmap function, when the size of the request is larger than 128Kb.

Chunks of memory are represented using the boundary tag method. The size of the chunk is stored both in the front and at the end of it. The boundary tag method makes coalescing very fast. Also in the size field a bit showing whether the chunk is free or not is placed. Figure 4-1 shows an allocated chunk while figure 4-2 shows a free chunk [11].
It is worth noting that due to size alignment (two times the `sizeof(size_t)`), the lower 3 bits of the size field is guaranteed to be always zero and therefore they are used as metadata flags. A indicates if the chunk is in a non-main arena, M indicates whether the chunk was allocated using `mmap` function and P indicates if the previous chunk is in use. In case the previous chunk is free, the prev_size field states the size of that chunk. Otherwise, this field is part of the user data field.

4.2.2 Top Chunk
This is the chunk that borders the end of the available heap segment. It is used as the last resort to handle requests and if required to extend the heap by calling the `sbrk` system call. In short, it is like any other chunk placed in a specific memory area.
4.2.3 Last Remainder Chunk

This is another special chunk and like top chunk it is not directly found in a bin. It is the result of small allocation requests where they do not have an exact fit in any of the bins.

4.2.4 Bins

Chunks once freed are stored in linked lists (segregated lists), sorted by size, except the top chunk which is directly managed by the arena. The linked lists are managed by an array of pointers called bins. To reduce frequent system calls, chunks that are freed are not immediately returned to the operating system but are kept into organized bins for future use [11].

There are two types of bins based on the size of the chunk they maintain:

- **fast bins.** Those bins do not directly coalesce after a free request with surrounding chunks (P flag is always on) and are stored to single linked list. Additionally, they are removed from the list in LIFO order and are used to handle requests of up to 80 bytes. Since most of the time they do not merge they may be faster than the other types of bins. Their main usage is to act as a cache for fast small size allocations. Usually there are ten fast bins holding chunks ranging from 0 to 8 bytes size. Figure 4-3 shows the organization of them.

- **normal bins.** Those bins are held in a double linked list and are further categorized to:
  
  - unsorted bin. It contains recently freed chunks. Before freed chunks are placed to the corresponding bins, a single effort is made to use them to serve the next malloc request. If the request cannot be satisfied, it is placed in one of the other two types.

  - small bins. It handles chunks of up to 512 bytes. It seems redundant to have small bins with the same size as the fastbins. Hence, since under certain circumstances fastbins may consolidate (when a malloc or free request for large chunks is made in order to avoid reduce fragmentation), it is possible to let fast chunks bins to consolidate into a larger size that fit into a small bin. There are 64 small bins.
- **Large bins.** The chunks with size over 512 bytes and less than 128Kb are stored here. Those bins are sorted by size in descending order and a FIFO order is used. There are 64 of them. Figure 4-4 illustrates the normal bins.

![Fast Bins](image1)

![Normal Bins](image2)

Searching in bins is made using the best fit policy while the normal chunks use deferred coalescing.

Bins of sizes smaller than 512 bytes contain chunks of the same size, spaced 8 bytes while larger bins are approximately logarithmic spaced as shown in table 4-1.
### Table 4-1 Bins Size

<table>
<thead>
<tr>
<th>Number of Bins</th>
<th>Spacing Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
</tr>
<tr>
<td>4</td>
<td>32768</td>
</tr>
<tr>
<td>2</td>
<td>262144</td>
</tr>
<tr>
<td>1</td>
<td>what’s left</td>
</tr>
</tbody>
</table>

#### 4.2.5 Multithreading and Arenas

The use of processors supporting multiple threads has led to the need of efficient handling of concurrent memory requests. In a single thread program, only the main arena (the actual heap segment) is utilized. On the other hand, in multithreading, several arenas are utilized and all threads share all available arenas. Heap contention is avoided because if a thread requests a memory block and all arenas are occupied, a new arena is created and is linked to the others. The total number of existing arenas is less than double the number of physical CPU cores.

So when multiple threads are utilized multiple non-main arenas also exist. Each arena is locked using mutex locks (a lock that is set before using a shared resource and released afterwards) to prevent multiple threads from accessing that resource at the same time) [12], each one of them containing several heaps as illustrated in figure 4-5.

![Figure 4-5 Multiple Arenas](image-url)
Heap_info is a single contiguous mmaped memory region holding chunks that can coalesce. When a thread locks an arena it keeps using this one only.

4.2.6 Allocation Algorithm

In this section the steps followed to serve an allocation request from a thread using Ptmalloc2 are presented.

I. First of all, an arena must be obtained using a mutex lock. If the locking process is successful the allocation will be served by that arena, otherwise a new arena will be created. Arenas are held in a linked list, so traversing this list can quickly determine which arenas are not locked.

II. After successfully obtaining an arena the requested size is converted to the actual chunk size where the additional metadata area is added and alignment is made. At this point the size of the newly created chunk is checked to determine from which bins to satisfy the request. The available outcomes are:

   a. **Allocate from fast bins.** If the actual size is less than the default 64 Bytes (max_fast variable) fast bins are used. The array index is calculated for the particular arena which determines from which list to allocate the chunk. The first bin from that list is flagged as victim and is the one that is used to satisfy the request. Pointer rearrangements are made to keep the list consistent. Additionally, several security checks are made to ensure that a valid chunk is returned.

   b. **Allocate from small bins.** If the converted size is between 64 and 512 bytes it falls into this category. In the same way as the previous outcome, the correct array index is calculated to obtain the corresponding list pointer. In this case the victim is the last chunk in the bin. If this chunk is NULL, indicating no free small bin, an attempt to satisfy the request from recently freed unsorted chunks is made. Those bins are created by consolidating free fast bins. The list of fast bins is traversed and any free bins are merged and placed to the unsorted bins. If an appropriate chunk is found it is used to serve the allocation.
c. ** Allocate from large bins.** If after the consolidation process the victim was still NULL or the chunk size is larger than 512B but smaller than 128Kb, the allocation continues to the large bins. The suitable array index is calculated to find which list pointer to search for a suitable large bin. Every time a large bin search is made, a consolidation is also performed, to merge the free fast bins. When a large bins is found to satisfy the request, it is returned and the excess is split and placed to the suitable bins.

d. **Allocate using mmap call.** When the request size is larger than 128Kb the `mmap` function is called to handle the request, creating an anonymous mapping.

III. In case none of the above outcomes satisfies the request, the topchunk is used to carry out the allocation. Top chunk can also be split to better handle a request.

IV. It is possible that even the top chunk is not able to satisfy the request. In this case the heap is grown by calling either the `brk` or the `mmap` calls.

Figure 4-6 illustrates the above steps.
4.2.7 Free Algorithm

During this section the Ptmalloc2 free operation is presented:

I. If the argument in free call is zero then nothing happens and the call exits.

II. As in allocation, an arena mutex lock is required to ensure thread safety. So the arena of the chunk to be freed needs to be locked.

III. In case it is a mmaped chunk then the unmmap function is called to return the mapped memory space back to the operating system.

IV. According to the chunk’s size it is placed to the corresponding bins.
   a. If the chunk is in the fast bins size, it is placed to the corresponding fast bins arena lists and do not coalesce.
   b. If it is not in the fast bins range, coalesce with neighbor chunks and increase its size.

V. In case the chunk is close to the top chunk then merge with the top chunk.

VI. Two checks are made to determine the case that the top chunk is larger than:
   a. fastbin_consodilation_threshold, call consolidation function, as previously explained.
   b. trim_threshold, the heap trimming function is called, where part of the top chunk is given back to the operating system and the heap shrinks.

Figure 4-7 illustrates those steps.

![Figure 4-7 Ptmalloc2 free](image)
4.2.8 Ptmalloc2 Parameters

Ptmalloc2 provides the programmer with the mallopt function to fine tune several parameters and change the behavior of its operation. The signature of the function is:

\[
\text{int mallopt( int param, int value);}
\]

On modern Linux distributions the following parameters are available:

- **M_MAP_MAX.** This parameter specifies the maximum amount of \texttt{mmap}ed chunks allowed to be used at a given time. The default value is: 65536. It is quite useful when the underlying operating system does not scale well with \texttt{mmap} system call.

- **M_MXFAST.** The maximum size of a fast bin. All allocation requests up to that size are satisfied by fast bins. Default value is 64, while a value of 80 is commonly used, 0 disables the use of fast bins.

- **M_MAP_THRESHOLD.** The size threshold over which an allocation request is served via an anonymous mapping instead of the heap segment. A value of 0 indicates that all requests (up to M_MAP_MAX), are satisfied by the \texttt{mmap} system call. Default value is 128KB. However, in the latest Ptmalloc2 version this threshold is dynamically adjusted. Every time a request is served using \texttt{mmap}, the threshold increases to the size of that request.

- **M_TOP_PAD.** Whenever glibc uses \texttt{brk} to increase the size of the heap segment, additional M_TOP_PAD bytes are requested in the hope that those bytes will minimize the need of future system calls. Those extra bytes are called padding. Default value is 0.

- **M_TRIM_THRESHOLD.** Indicates the minimum amount of memory allowed at the top of the heap segment [figure 4-5]. In case there is excess memory stored at the top of the heap segment, the \texttt{malloc_trim} function is called and this amount returns to the operating system. Default value is 128KB.

Using the above parameters a programmer can fine-tune the operation of the \texttt{malloc} call, to make it more memory efficient or faster. It is worth mentioning that, in case there is no access to the source code; those parameters can also be set as Environment Variables.
4.2.9 Discussion on Ptmalloc2
According to [7], this implementation is among the fastest while also being among the most space-conserving allocators. A summary of its operation:

- For small allocation requests of up to M_MXFAST it is a caching allocator, which maintains pools of quickly recycled chunks.
- For large request between 128KB and 512KB, it is a best fit allocator, with ties solved via FIFO order.
- For very large requests, larger than M_MAP_THRESHOLD it relies on system mapping functions.

The main problem using Ptmalloc2 is its execution time performance in multi-threaded environments. Memory does not move between arenas, which may lead to amounts of wasted memory, since a thread may request additional space instead of using an underutilized arena. Furthermore, since each thread locks an arena, other threads may have to wait to use the locked one or create a new one. This contention reduces performance and may increase fragmentation, under specific allocation request sequences.

Finally, each header and footer contains 4 bytes of information about the size of the chunk, as well as information about other chunks, leading to memory waste, especially if the allocation requests are for small sizes.

4.3 Tcmalloc
Tcmalloc is Google’s attempt on developing a dynamic memory allocator and is part of the company’s google-perftools package. It provides support for almost every operating system and provides the entire suite of memory management calls. Its usage is fairly simple; the programmer just needs to link the Tcmalloc library with their program and all malloc calls are automatically substituted by Tcmalloc calls, meaning no source code changes needed [3,11,9].
4.3.1 Architecture

As with Ptmalloc2, Tcmalloc’s goal is to minimize the heap contention while it is optimized for high concurrency programming situations. Multithreading environment speed was a development goal since its designing phase. The allocator uses three levels of abstractions to handle the memory provided by the system. Each level can be considered as a memory cache to avoid excess system calls.

To achieve good multithreading performance, minimum heap contention is considered really important, so the allocator’s internal structures are almost lock free, while each thread has its own thread local heap used for small allocations. The memory blocks in the local heap vary from 8B to 32KB and no thread synchronization is needed to serve allocation requests for those sizes. The use of thread local heap enables most of allocation and deallocation requests to be satisfied by them without requiring any locking mechanism, increasing the allocator’s threading scalability.

In case the request cannot be served by thread local heaps the allocator implements a central cache, which is shared by all threads. Spin locks (thread waits in a loop until the resource is available)[12] are used to minimize the contention of sharing the central cache. The central heap is in charge of handling the pages handed over by the page heap (spans), a number of contiguous pages, and make them more manageable for the requests by feeding the local thread heaps. The pages are divided and the memory blocks are placed on the central cache’s Free Lists.

Finally, there is the page heap allocator, which is the abstraction level responsible for dealing with the system calls (either sbrk or mmap) in order to acquire additional memory to feed the central cache and the local thread heaps. Furthermore, large requests (over 32KB) are served by the page heap. In this level, memory is handled in integral page multiples called Spans.

The heaps are illustrated in figure 4-8.
We can observe that page heap is used to manage the heap segment of the operating system. Central cache is in charge of feeding the local thread caches with available memory obtained from page heap. Central heap and page heap require obtaining a lock to enforce synchronization. Most of the allocations are served by the local thread cache, making Tcmalloc one of the fastest allocators since a lockless mechanism is used for small sizes. Additionally, Tcmalloc utilizes automated garbage collection to return unused memory from thread local caches to the central cache heap.

4.3.2 Page Heap
At this level of abstraction the allocator deals with the communication between the internal structures and the system calls. When additional memory is needed this is the place where a call to mmap or sbrk is performed. The requested memory from the system is an integral multiple of pages. Furthermore, when the program requests
memory larger than 32KB, this size is rounded up to the nearest full page, where the default page size is 4KB and is satisfied by the page heap.

Page heap organizes the available pages into span objects. Spans represent a contiguous run of pages and are presented in the figure 4-9.

![Figure 4-9 Tcmalloc Spans](image)

We can see that all spans occupy 3 contiguous pages. Each span can be either allocated or free. Allocated spans were either directly used to serve a large allocation request or they were carved to feed the lower abstraction levels with available memory, since allocation in page sizes is not convenient.

The free spans are managed using a global array of free lists. Each one of the 256 positions in that array indicates the number of pages the corresponding span contains. The 256 position is a free list of spans that have length \( \geq 256 \) pages. The array is illustrated in the figure 4-10. An allocation of \( k \) pages is satisfied by looking at the corresponding array position and the available free lists. In case the particular list is empty the allocator looks at the next array position for available free spans. In case there is no free span, additional memory is requested by the system. If the allocation is satisfied by a higher number of pages, the remainder is returned to the suitable free list.

Searching the free lists is done using the best fit algorithm. However, since the free lists are not address ordered this is not the most beneficial with regard to the fragmentation.

4.3.3 Size Classes and Page Map

Before we continue, the notion of size class and page map need to be explained. Size class is the main data structure used to store available memory chunks and it is an array of single linked lists. Each span is carved to fill them, Tcmalloc uses 170 size classes.

Each size class contains free chunks of a particular size and is spaced with the next one based on that size:

- Small sizes are separated by 8 bytes
- Larger sizes by 16 bytes
- Even larger by 32 bytes and so on..

The maximum spacing is 256 bytes.
Figure 4-11 illustrates the size classes

![Figure 4-11 Size Classes](image)

Page map serves two functionalities:

- Maps addresses to their corresponding span objects. For 32bit architecture it is maintained using an array of size 4MB (2^20 entries each 4 bytes), while for 64bit a radix tree is utilized.
- Maps a chunk’s id to its respective size class.

Page map is the mechanism that the allocator uses to store its metadata. Even though it is not as space efficient as other allocators it is cache beneficial.

4.3.4 Central Cache

Managing the memory directly in pages provided by the page heap is not convenient; this is why another abstraction level is introduced. The central cache is in charge of making the spans usable by the application by carving and transferring free chunks from the page heap to the thread specific cache.

Figure 4-12 shows that central cache is shared between the threads. This cache is organized as a two level data structure: a set of spans and a linked list of free objects per span called transfer list cache. Its main functionality is to manage the spans into free lists per size class, called central free list, and appropriately feed them to the thread local cache.
In this abstraction level, spans are managed by an array of free lists indexed by size class. Each free list contains a set of spans which are split into chunks of the corresponding size class. Figure 4-12 illustrates this abstraction level.

Central Cache manages the available spans and allocates them to free chunks which will be used to feed the local thread cache.

4.3.4 Thread Local Cache
Access to central cache requires acquiring a spin lock; this is why the thread local cache is created. Tcmalloc assigns each thread a local cache to satisfy allocation requests of up to 32KB.

In this level of abstraction the available chunks previously transferred from the central cache are locally managed so there is no need for additional locks. The thread local cache consists of an array of single linked free lists indexed by size class as shown in figure 4-12. The lists are fed with free chunks created by the central cache.
The data structures in this abstraction also contain metadata about the size of the lists and their contents. In best case scenario the allocation and de allocation from the thread cache are made in constant time.

4.3.5 Garbage Collection
Under certain circumstances the allocator utilizes thread cache garbage collection. In case the thread cache exceeds the threshold of 2MB the garbage collector moves the unused chunks from the thread cache lists back to the corresponding central free lists. The number of objects to move back to the central lists is dynamically determined.

4.3.6 Allocation Algorithm
Through this section the steps followed to serve a small allocation request (<32KB) are presented.

I. Map requested size to the corresponding size class.

II. Search the thread local cache free lists for that particular size.

III. If the corresponding list is not empty, the first chunk is used to satisfy the request. So far no locks are needed.

IV. In case the thread local cache is empty:
   a. Acquire the spin lock to access the central free list
   b. Fetch a bunch of chunks from the central free list from the particular size class.
   c. Fill the thread local cache with the newly fetched chunks.
   d. Return the chunk from the local thread cache.

V. If the central free list is also empty:
   a. Acquire the spin lock to access the page heap
   b. Allocate a run of spans from the page heap
   c. Split the spans to the corresponding size class central free lists.
   d. As before move a bunch of chunks from central cache to the thread local cache and satisfy the allocation.
   e. If the page heap is empty request additional memory from the system.
For allocations over 32KB:

I. Look into the corresponding free list of spans of the page heap.
II. If the exact list is empty look to the next list and so on.
III. If none of the lists contains the allocation size, request additional memory from the system.
IV. If the request is satisfied using more memory than needed (i.e., rounding), the excess is placed to the corresponding list.

Figure 4-13 illustrates the above steps.

**Figure 4-13 Tcmalloc Allocation Algorithm**

### 4.3.7 Free Algorithm

The `free` algorithm executes the following steps:

I. Compute the free’s request page number.
II. Map the page number to the corresponding span object.
III. Determine the type of object; three possible outputs can occur:
a. Small object: insert it into the appropriate free list of the current local thread cache. An object is considered small if it is less than 32KB.

b. In case that list exceeds the 2MB threshold, trigger the garbage collector and move unused chunks to the central free lists.

c. Large Object: look at the neighboring spans and if those are free, coalesce them and insert the newly created span to the corresponding page heap list.

Figure 4-14 shows the de allocation algorithm.

![Figure 4-14 Tcmalloc Free Algorithm](image)

4.3.8 Tcmalloc Parameters
As in Ptmalloc2, the behavior of the allocator can be manipulated using several parameters which are set as Environment Variables. The most important are:

- **TCMALLOC_SKIP_MMAP.** If set to true the `mmap` system call will not be used to request additional memory from the system.
- **TCMALLOC_SKIP_SBRK.** It is the same as the previous but the `sbrk` call is disabled.

- **TCMalloc_RELEASE_RATE.** Indicates the rate at which the physical memory is freed by the system using the `madvise(MADV_DONTNEED)` system call. Note that it concerns the release of physical and not virtual memory.

- **TCMalloc_MAX_TOTAL_THREAD_CACHE_BYTES.** The parameter controls the maximum amount of memory that may be allocated to the thread local caches. However, this is not a strict order. The increase of this may increase the performance in multi-threading programs with the cost of additional memory wastage.

4.3.9 Discussion on Tcmalloc

In general, this allocator is considered one of the fastest around, especially in multithreading allocations. The use of per thread local cache for allocations up to 32KB is really efficient and increases its scalability, since it does not require any locking mechanism.

However, up to its current version it does not return any memory back to the operating system and is considered to have higher fragmentation than the other allocators mostly due to its metadata usage.

4.4 TLSF

We can notice that the behavior of the above algorithms changes according to the allocation or de-allocation block size. Different cache levels and mechanisms are used based on that size, which dramatically affects the response time of the allocation process. Additionally, the performance is influenced by the sequence of the requests making the performance of the allocator quite unpredictable.

However, there are applications called Real Time Applications that need to be completely predictable and meet the following dynamic memory management requirements [5,13]:

• Bounded response time: The worst case execution time of a de-allocation or allocation request should be constant and bounded no matter the block size.
• Fast response time: The response time of the allocator must be fast, not just bounded and predictable.
• Memory requests always must be served: Although non real time applications can receive a NULL pointer in case the request is not satisfied or when the system runs out of memory, in real time applications this chance must be kept to minimum.

Applications such as VoIP or telemedicine fall into the real time category, where certain criteria (low latency, fast and predictable response time) must be met to keep the user experience at high levels.

LTE was developed having in mind those needs and applications rely on LTE’s infrastructure to fulfill the mentioned requirements. [1]

The need to meet those requirements led the team of M.Masmano, I. Ripoll, A. Crespo and J.Real to develop the Two-Level Segregated Fit (TLSF) allocator.[2]

4.4.1 Architecture
The main structure of TLSF uses a set of segregate lists that can be directly reached through the size of the blocks implementing a good-fit policy (return the smallest chunk of memory big enough to satisfy the request).

The First Level Index (FLI) contains size ranges, similar to size classes, in power-of-two, 16..31, 32…63 and so on up 4GB.

The second level(SLI) splits each first index range linearly in a number of ranges of an equal width, for example the range of 32..63 can be linearly divided to four sub-ranges: 32..39, 40…47, 48…55 and 56…63.

Each segregated list has an associated bitmap used to mark which lists are empty and which contain free blocks while information for each block is stored inside of it. Figure 4-15 [2] illustrates this architecture.

Additionally, the boundary tag technique is used for fast immediate coalesce.
In this figure we can see that lists 32Bytes up to 4GB are all allocated and there are available free chunks in the 32 and 64 Bytes lists. The free elements in the second level indexing indicate the free chunks. Each free chunk in a particular size range has a pointer to the next available chunk using the boundary technique (as in Ptmalloc2), so the immediate coalescing can be performed fast.

A word-sized bitmap is used on both levels to indicate which lists are free and which are not. The usage of bitmaps increases the performance of the allocator, because processor bit instructions are used, while keeping the constant allocation time. In our figure the first level bitmaps show that the first and second indexes in the array are available while the same occurs for the second level bitmap.

We can note that there is only one available memory pool shared by all threads. TLSF does not address memory contention, but since FLI and SLI are small and searching is performed in constant time, that is not a problem. While initializing for the first request, the allocator gets a user defined POOL_SIZE memory area from the Operating System and this area is used for creating the FLI and the SLI. When this
pool is exhausted the allocator will automatically grow it by requesting additional memory from the Operating System by calling either sbr or mmap system call.

Finally, the searching for an available free chunk always starts at the list that contains sizes larger or equal to that requested size, in order to eliminate searching for smaller sizes. Consider the following example: for request allocation size 74, the list ranging [72…79] would be used by the best fit policy. However, using that list implies that the 72 and 73 sizes must be first compared and then discarded, making the allocation time non-predictable. Instead of discarding the smaller sizes the allocator always searches the list whose minimum size is at least as large as the requested one. In our example the request is satisfied by the [80…87] list.

Even though this policy may be faster and provide constant response time, it can increase the fragmentation. To overcome this problem the allocator uses a sophisticated rounding and splitting policy.

4.4.2 Allocation Algorithm
In this section the allocations steps for TLSF are presented. It is worth noting that the requested size does not alter the behavior of the allocator

I. Compute the indexes of the list that holds chunks of size larger or equal than the requested size.
II. Starting from that obtained list, search until a non-empty free list is found.
III. Extract the chunk at the head of that list and if it is larger than the requested size split it.
IV. In case the leftover is larger than a predefined threshold:
   a. Round up, split the chunk and create a chunk header on the remaining memory while updating the header of the original chunk.
   b. Compute the indexes on where to place the remaining chunk.
   c. Insert that chunk to the corresponding list.
V. Return the memory chunk
VI. Update the bitmaps.
VII. If no memory is available, request additional memory from the Operating System.

Figure 4-16 illustrates the above steps

4.4.3 Free Algorithm

The steps being followed for the deallocation of a requested size are the following:

I. Check whether the neighbor chunk is free.
II. If it is free remove it from the segregated list and coalesce with the newly freed chunk.
III. If the neighbor is not free, place the chunk being currently freed to the corresponding list.
IV. Update the bitmaps to indicate that there is more available space.
Figure 4-17 demonstrates those steps.

### 4.4.4 TLSF Parameters

There is a parameter and a compilation flag that may alter the behavior of the allocator.

- **POOL_SIZE**: This parameter is set by the programmer and indicates the size of the default pool to be used by the allocator. A small value will trigger more system calls to request additional memory from the operating system, while a larger value will increase the fragmentation.
- **SBRK** or **MMAP**: A compile flag is used to guide the allocator on which system call to use each time additional memory is requested from the operating system.

### 4.4.5 Discussion on TLSF

This allocator is the one considered to be more suitable for real time applications.

The allocation and the `free` algorithm are performed in constant O(1) time, no matter the requested size. This time is achieved using the bitmap search functions and the boundary tag technique.
The fragmentation, according to the writers, has a worst case scenario of 15%, meaning that at most 15% of additional memory may be wasted to satisfy the requests.

A drawback of that allocator is that the programmer needs to have access to source code in order to drive all the allocation and de allocation calls to be served by the TLSF. In particular the malloc should be replaced by tlsf_malloc and free by tlsf_free, given that the default memory pool is used.

Additionally, the programmer must determine the size of that Pool based on the needs of the application.
Chapter 5 Evaluation Method and TestBed

Through this chapter the method used to evaluate the performance of the allocators is analyzed and different test case scenarios are introduced.

5.1 Random Synthetic Traces

Using random synthetically generated *traces* is the traditional technique used to evaluate the performance of the dynamic memory management.

A trace is a recorded sequence of allocation and deallocation requests. Synthetic traces are automatically and randomly generated by a subprogram; size and allocation/de-allocation time (allocation to de-allocation time specifies the life of an allocation) distribution may be inputs for that program.

Given those inputs the program generates a sequence of requests. The most commonly used method to generate this sequence is through mathematical functions to determine the size and lifetime distribution such as an exponential distribution.

The output of that program is then processed by the driver program, which executes the sequence of allocations and de-allocations. The results are recorded and presented. Figure 5-1 illustrates the synthetic traces.

![Figure 5-1 Synthetic Trace]
Even though the generated distributions are widely used, they are not so suitable to capture the behavior of the allocator. The allocator uses a strategy to exploit regularities in a program’s behavior. Using randomly generated order of allocation and de-allocation requests eliminates some real program regularities. Fragmentation is affected by the order of these events, meaning that those traces will not capture the real fragmentation performance of the allocator. Randomly generated traces assume that there is no ordering information in the requests that could be exploited by the allocator. This lack of information leads to the conclusion that randomly generated trace files are not the most suitable to determine the fragmentation behavior of an allocator.

5.2 Real Program Tracing

Real programs do not usually behave randomly. There are regularities found in real programs and typically the size usage follows the Pareto distribution (the majority of the requested objects have small sizes, while there do exist requests for larger objects but not so often).

In terms of memory usage over time for real programs three typical patterns have been observed and classified [7]. They are presented in the following figures where the X axis expresses the execution time in seconds, while the Y axis refers to the memory usage:

- **Ramps**. In this pattern data accumulate monotonically over time, it may keep a log of events. Figure 5-2 shows this pattern.
• **Peaks.** In this pattern the program allocates many objects which are briefly used and then it frees them all. It is considered being like a shorter time ramp. Figure 5-3 illustrates it.

![Figure 5-3 Peaks Pattern](image)

• **Plateaus.** In this pattern the program allocates its memory quickly and then uses this amount of memory over the period of its execution. Figure 5-4 shows this pattern.

![Figure 5-4 Plateau Pattern](image)

A program can exhibit more than one pattern. It can follow several at the same time, depending on its execution state. Especially for the ramp pattern, a synthetic trace cannot capture its behavior and cannot be used to measure the fragmentation effect.
5.3 Methodology
The most important periods of a program’s execution are those that require the most memory. In these periods it is most suitable to measure the fragmentation. During this thesis the 4\textsuperscript{th} method presented in chapter 2 is used to measure the fragmentation (maximum live memory relative to the maximum amount of heap size).

In order to capture the memory allocations and de-allocations of a real program, this program should be instrumented to log its dynamic memory activity. This log file is later fed to the driver program which line by line executes the free or malloc function.

In this thesis, the memwatch program was used to capture the behavior of a running process. Memwatch is a wrapper for malloc and free calls, capturing the requested size, the pointer which returned to the malloc function and the time when the event took place. Those statistics are stored in a log file which later will be the input for our driver program. Figure 5-5 demonstrates the real program tracing.

![Diagram](image)

Figure 5-5 Real Program Tracing

The output of the driver program is the total execution time, the maximum amount of live memory (the memory that is allocated and used by the program and not yet de-allocated), the maximum amount of the heap and finally the fragmentation calculated from the previous two values.
5.4 Driver Program Heap Size

To capture the heap’s size, the `/proc/[pid]/smaps` file was used. The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures. For the heap size the sub file `smaps` is utilized. This file shows memory consumption for each of the process mappings (the different segments shown in chapter 2). For each mapping several values are generated. In this thesis the focus is on the segment that is responsible for the dynamic memory management. This segment includes the heap and the anonymous mappings (mmap allocations).

For those segments the *Resident Set Size* (RSS) value is measured. RSS shows the amount of mapping that is currently resident to the main memory. To make sure that the kernel actually maps the segment to the main memory, once a size is allocated it is also used by the driver program, to overcome lazy swapping as explained in chapter 2. A script written in `awk` was used to extract the RSS values from the smaps file.

It is worth mentioning that each allocator provides its own statistics regarding the heap size, but each one uses a different approach. This is the reason that the universal smaps file was used.

5.5 Driver Program Live Memory

To capture the live memory used by the program, the `free` and `malloc` functions were overloaded by our own functions. Those functions interleave with each call to `free` and `malloc`, and measure the amount of memory requested and freed. Each time a `free` call is made, the size is also passed as a parameter to the overloaded function. Using a global static size variable we can measure the amount of live memory by increasing this variable each time the `malloc` function is called and decreasing it each time a `free` is called. To verify the results of live memory the Valgrind program was also used. Valgrind is a suite of tools for debugging dynamic memory related problems.

Additionally, an array is used to keep track of how many allocations are still live, over the period of time in the trace file.
5.6 Driver Program Execution Time
To measure the execution time the following function is used:

```c
struct timespec start, stop;
clock_gettime(CLOCK_MONOTONIC_RAW,&start);
```

The `clock_gettime` provides a high resolution clock suitable for measuring the execution time. Before the actual execution of the tracing file the function is called using the start structure and after finishing the reading of the trace file it is again called using the stop structure.

In general, there are several methods of measuring the elapsed time of a program’s execution:

- **CLOCK_MONOTONIC.** Represents monotonic time since some unspecified starting point. It adapts to NTP
- **CLOCK_MONOTONIC_RAW.** Similar to CLOCK_MONOTONIC, but provides access to a raw hard ware-based time that is not subject to NTP adjustments
- **CLOCK_REALTIME.** Reports the actual wall clock time. Can jump forwards and backwards as the system time-of-day clock is changed

In this thesis the CLOCK_MONOTONIC_RAW was used.

5.7 Driver Program Overview
In order to get the fragmentation and execution time results a real program’s tracing file was captured. This trace file contains the sequence of malloc and free calls.

Without getting into details, the file is processed line by line and calls the stored call. Figure 5-6 illustrates an overview of the driver program while Appendix A has a deeper explanation of the structure and the execution details.
Figure 5-6 Driver Program Overview
5.8 Test Bed and Scenarios

In this section the different scenarios and the hardware used are presented.

First of all, the simulations were run on a virtual machine environment under Windows, running the Lubuntu 12.10 Version. The kernel version used is 3.5.0-17 generic, the gcc is 4.7.2 and glibc version is eglibc 2.15. For the virtual machine, 5.8 GB of RAM were assigned while 2 cores were utilized. The CPU used is the i5 3427U, which can use up to 4 threads and has 3 MB of Cache. The High Performance profile is selected from the windows power options, otherwise the CPU may throttle while the virtual machine is loaded which will lead to false results. For each scenario the –O3 compiler optimization flag is set. For TLSF the pool size was chosen to be 130MB and add this amount in each scenario. This was determined after a complete execution of the real program, where the amount of memory used was recorded.

To test the performance of each allocator 5 different scenarios were created, each scenario using different amount of threads. The execution time and the fragmentation is measured for each allocator, Table 5-1 summarizes the different scenarios.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Number Of Threads</th>
<th>Number Of Execution Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only Main Thread</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10000</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>10000</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>10000</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>10000</td>
</tr>
</tbody>
</table>

After executing each scenario the maximum live memory is reported per thread, the total execution time and the total maximum heap size.

Additionally, in this thesis as a separate case, the impact of different settings for each algorithm is evaluated. The details regarding the values of the settings are presented in the results chapter.
5.9 Trace File Analysis

To get more accurate results we used a real program’s trace file.

This trace file originates from a program which has important aspect on the control plane execution on the eNodeB. A run showing the live memory over time is shown in figure 5-7.

We can clearly see that our program falls into the ramp allocation pattern.

The

LD_PRELOAD=/lib/i386-linux-gnu/libmemusage.so

shared library was used to produce several statistics about the size distribution and the total amount of free and malloc calls. Figure 5-8 shows the distribution of the allocated sizes and the relative frequency for each size classes.
We can notice that the trace file we will analyze follows the pareto distribution, where the majority of the allocations are small sizes, where there is the tail containing larger allocations but less frequent.

Finally, Table 5-2 shows the total amount of `malloc` and `free` calls for that trace file under 10000 iterations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Malloc</code></td>
<td>5810003</td>
</tr>
<tr>
<td><code>Free</code></td>
<td>4740014</td>
</tr>
</tbody>
</table>

### 5.10 Makefile and Trace

To automate the procedure of compiling all the necessary parts a Makefile was created. Each time the make command is executed the following executable programs are produced:

- `malloc`
- `Tcmalloc`
- `tlsf`
- `trace_malloc`
- `trace_Tcmalloc`
- `trace_tlsf`

The first three executables are used to run the corresponding algorithm to evaluate its behavior.

The rest are used to trace the live memory and the heap utilization, so we can produce additional plots and figures. Two files are created, one containing the live memory over time and the other the heap size over the total used memory. The filenames consist of the executable name, whether it is the live or the heap memory and the scenario we are tracing.
Each scenario was executed 5 times and the average execution time was recorded. To make things easier, a bash script was written to instrument the execution of each scenario for 5 times. For example, the `malloc_auto_script.sh` executes the given trace file, for 5 times for each scenario. The results are appended to different files based on the executing scenario.

Finally, the total amount of `mmap` and `sbrk` system calls was also recorded. However, this amount represents system calls that may not be directly related to the memory calls, since `mmap` and `sbrk` are also used to serve other operations, i.e., opening a file results in several `mmap` calls.

This will not impact the test results since there is always a constant system call overhead to the memory allocator itself.
Chapter 6 Results

In this chapter the performance of the three different allocators, based on the trace file and methodology introduced in chapter 5, is reported. Additionally, tuning settings for each allocator are presented and evaluated.

6.1 First Scenario

Recall that in this scenario only the main thread is used and all allocators use their default behavior while for TLSF the pool size is 130MB.

Figure 6-1 presents the fragmentation results of the different allocators in terms of wasted memory over the actual live memory requested by the program.

![Figure 6-1 First Scenario Memory Usage Over Live Memory](image)

Table 6-1 shows the maximum values for each allocator as well as the difference between the ideal live memory usage and the fragmentation percentage.
Table 6-1 First Scenario Maximum Size Values and Fragmentation

<table>
<thead>
<tr>
<th>Live Memory</th>
<th>116903</th>
<th>7548</th>
<th>6.46%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2 Heap Size</td>
<td>124452</td>
<td>548</td>
<td>4.53%</td>
</tr>
<tr>
<td>Tcmalloc Heap Size</td>
<td>129776</td>
<td>12872</td>
<td>11.01%</td>
</tr>
<tr>
<td>TLSF Heap Size</td>
<td>127764</td>
<td>10804</td>
<td>9.29%</td>
</tr>
</tbody>
</table>

We can note that Tcmalloc requires more memory to satisfy the same amount of live memory compared to the other allocators. Ptmalloc2 requires the minimum amount of extra memory while the TLSF is between the two allocators. Tcmalloc requests memory in multiples of integer page numbers from the operating system while its metadata need more memory compared to the other allocators.

Figure 6-2 presents the execution time for each allocator while Table 6-2 shows the difference between the fastest allocator and the rest.

Table 6-2 First Scenario Execution Time

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Execution Time in seconds</th>
<th>Difference between Tcmalloc</th>
<th>Percentage Difference Between Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>2.23256285</td>
<td>0.406421187</td>
<td>22.26%</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>1.826141663</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TLSF</td>
<td>2.439340691</td>
<td>0.613199028</td>
<td>47.55%</td>
</tr>
</tbody>
</table>
The tcmalloc allocator is the fastest, for the single thread scenario. All the requests are satisfied by its thread local cache and this is the reason behind its performance advantage. TLSF is the slowest because the coalescing algorithm is often triggered as well the splitting and rounding occurring fairly often. Ptmalloc2 is in between of Tcmalloc and TLSF, because most of the requests are served by the fast bins; however, this will eventually trigger the coalescing algorithm.

For the first scenario the system calls are also presented in table 6-3. It is worth mentioning that only the mmap and brk calls are shown since those two calls are used to handle the dynamic memory allocations. For reference, the system calls of the following simple HelloWorld program are presented.

```c
#include <stdio.h>

main() {
    printf("Hello, world");
}
```

<table>
<thead>
<tr>
<th></th>
<th>MMAP calls</th>
<th>SBRK calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>18</td>
<td>1073</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>27</td>
<td>130</td>
</tr>
<tr>
<td>TLSF</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

By default, TLSF uses the mmap system call to expand the default memory pool. In this scenario the default memory pool is set to 130MB, based on the recorded live memory. This setting was based on real program’s memory consumption. In general, this setting is connected with the running program and should be tuned accordingly.

It is clearly shown that TLSF requires less system calls than the other allocators. The default memory pool size is adequate enough to satisfy the requests without making additional system calls. Ptmalloc2 uses sbrk to grow the heap segment and the number of system calls shows that it does not request large amounts of memory each time a sbrk call is made. Hence it keeps requesting more and more compared to Tcmalloc, which requests big chunks of memory(integer multiple of pages).
The latter, also again indicates the fragmentation results, where Tcmalloc uses almost 70% more memory compared to Ptmalloc2 (7548 to 12872) for satisfying the same amount of live memory requests.

Some of the HelloWorld system calls are:

- `execve("./helloworld", ["./helloworld"], /* 22 vars */) = 0`, which executes the program pointed by the filename.
- `uname({sys="Linux", ..), which gives information about the system the program is executing
- `brk(0) = 0x85a5000`, indicates the starting brk address
- `fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0`
  - `mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb774f000`, loading of the standard library
- `write(1, "Hello World!\n", 13) = 13`, the actual execution

### 6.2 Second Scenario

In this scenario an additional thread is utilized, while TLSF uses 260MB pool size. Figure 6-3 presents the memory utilization.
Table 6-4 shows the maximum values

<table>
<thead>
<tr>
<th></th>
<th>Maximum Size in KBytes</th>
<th>Difference compared to Live Memory</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Memory</td>
<td>233788</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ptmalloc2 Heap Size</td>
<td>248912</td>
<td>7548</td>
<td>6.47%</td>
</tr>
<tr>
<td>Tcmalloc Heap Size</td>
<td>2600648</td>
<td>12872</td>
<td>11.49%</td>
</tr>
<tr>
<td>TLSF Heap Size</td>
<td>255508</td>
<td>10804</td>
<td>9.29%</td>
</tr>
</tbody>
</table>

The fragmentation results are basically the same as in the first scenario. Only Tcmalloc shows an increase over the previous scenario.

Figure 6-4 presents the execution time for this scenario. While table 6-5 shows the differences between the fastest and the slower allocators as well the exact values.
Table 6-5 Second Scenario Execution Time

<table>
<thead>
<tr>
<th></th>
<th>Execution Time in Seconds</th>
<th>Difference compared to Tcmalloc</th>
<th>Percentage Difference compared to Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>6,166512170</td>
<td>1,502276693</td>
<td>27,24%</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>4,664235477</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TLSF</td>
<td>5,230129087</td>
<td>0,936383083</td>
<td>16,43%</td>
</tr>
</tbody>
</table>

The heap contention of Ptmalloc2 becomes evident in this scenario. The performance difference between the fast Tcmalloc and the Ptmalloc2 has grown compared to the first scenario. This is clearly an indication that the usage of locks and utilization of only one heap area for all threads, introduces a big performance penalty. On the other hand the TLSF performance has drastically increased due to the negligible performance hit of accessing the shared pool memory. Of course the non-locking access mechanism of Tcmalloc makes it the fastest when an additional thread is used.

Table 6-6 shows the performance increase for the same allocator over the two different scenarios, where we can see how well the allocator scales when an additional thread is added.

Table 6-6 First to Second Scenario Difference

<table>
<thead>
<tr>
<th></th>
<th>First Scenario Execution Time in Seconds</th>
<th>Second Scenario Execution Time in Seconds</th>
<th>Increase between the Two Scenarios in Seconds</th>
<th>Percentage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>6,166512170</td>
<td>2,23256285</td>
<td>3,93394932</td>
<td>176,21%</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>4,664235477</td>
<td>1,826141663</td>
<td>2,838093814</td>
<td>155,41%</td>
</tr>
<tr>
<td>TLSF</td>
<td>5,230129087</td>
<td>2,439340691</td>
<td>2,790788396</td>
<td>114,41%</td>
</tr>
</tbody>
</table>

We can observe that TLSF and Tcmalloc scales better than the Ptmalloc2, which boils down to the heap contention and the way Ptmalloc2 handles additional threads.
The last table 6-7 demonstrates the system calls for the second scenario.

<table>
<thead>
<tr>
<th></th>
<th>MMAP calls</th>
<th>SBRK calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>19</td>
<td>1067</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>TLSF</td>
<td>21</td>
<td>3</td>
</tr>
</tbody>
</table>

A small increase is noticed for the TLSF and Ptmalloc2 MMAP system calls. On the other hand, Tcmalloc reduces its system calls, since it dynamically requests more memory when needed, to avoid excess system calls. Tcmalloc allocates more than needed memory using the sbrk and mmap calls, which is the reason it has excess memory usage and less system calls.

### 6.3 Third Scenario

In this scenario 2 additional threads are used. The TLSF’s memory pool is increased to 520MB.

Figure 6-5 presents the live memory of that scenario and the extra memory used by the allocators to satisfy this amount of live memory.
Table 6-8 shows the maximum memory usage of the allocators, the maximum live memory as well the fragmentation results.

<table>
<thead>
<tr>
<th></th>
<th>Maximum Size in Kbytes</th>
<th>Difference Compared to Live Memory</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Memory</td>
<td>467558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ptmalloc2 Heap Size</td>
<td>497884</td>
<td>30325</td>
<td>6.49%</td>
</tr>
<tr>
<td>Tcmalloc Heap Size</td>
<td>518356</td>
<td>50797</td>
<td>10.86%</td>
</tr>
<tr>
<td>TLSF Heap Size</td>
<td>511028</td>
<td>43469</td>
<td>9.30%</td>
</tr>
</tbody>
</table>

We can notice that the allocators tend to keep the same amount of wasted memory over live memory, meaning that the fragmentation is kept roughly at the same levels. The Tcmalloc fragmentation has reduced by 1%, while the other two introduce a 1% and 2% increase respectively over the previous scenario.

It seems that the allocators have bounded fragmentation result. So far Ptmalloc2 has around 6.50%, Tcmalloc around 11% and TLSF about 9.30% need of excess memory to satisfy the same amount of live memory.

Figure 6-6 illustrates the execution time for the third scenario and table 6-9 shows the exact values and the difference between the fastest allocator and the rest of them.
This result clearly shows the benefit of using Tcmalloc for multithreading applications over the other allocators. The use of local per thread cache which does not require any locking mechanism, is 39% faster than the mutex locking mechanism used in Ptmalloc2. We can also note that TLSF has decreased the execution time gap compared to Tcmalloc and the first scenario, which makes it more time efficient compared to Ptmalloc2.

The table 6-10 illustrates the system calls for that scenario.

<table>
<thead>
<tr>
<th></th>
<th>Execution Time in Seconds</th>
<th>Difference compared to Tcmalloc</th>
<th>Percentage Difference compared to Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>11,260893692</td>
<td>3,160632672</td>
<td>39.02%</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>8,10026102</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TLSF</td>
<td>10,16481476</td>
<td>2,06455374</td>
<td>35.38%</td>
</tr>
</tbody>
</table>

There is not a big increase in the system calls compared to the previous scenario. This result indicates that the allocators keep changing their behavior according to their strategy in order to minimize the excess system calls.

We can clearly conclude for the third scenario that Tcmalloc is the best fit in terms of execution time and multithreaded operation. However, this performance comes with a price tag and in this case it is the excess wasted memory required compared to the rest of the allocators.

Ptmalloc2 is the slowest of the allocators, but displays the smallest fragmentation while TLSF is between the two of them, in both execution time and fragmentation.
6.4 Fourth Scenario

The number of additional threads is increased to 7 for this scenario. The thread contention is really important for this scenario. The TLSF pool size increases to 1090MB, which is based on the memory used. As in previous scenarios, firstly the memory usage is presented in figure 6-7 accompanied by the fragmentation table, then the execution time and finally the system calls.

![Figure 6-7 Fourth Scenario Memory Usage Over Live Memory](image)

Table 6-11 illustrates the maximum memory usage and the fragmentation results

<table>
<thead>
<tr>
<th></th>
<th>Maximum Size in KBytes</th>
<th>Difference compared to Live Memory</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Memory</td>
<td>935097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ptmalloc2 Heap Size</td>
<td>996976</td>
<td>61878,76953</td>
<td>6,62%</td>
</tr>
<tr>
<td>Tcmalloc Heap Size</td>
<td>1033504</td>
<td>98406,76953</td>
<td>10,52%</td>
</tr>
<tr>
<td>TLSF Heap Size</td>
<td>1021924</td>
<td>86826,76953</td>
<td>9,29%</td>
</tr>
</tbody>
</table>
We can note that Tcmalloc’s fragmentation keeps decreasing over the different scenarios (from 11% at the first to the 10.52% at this one), Ptmalloc2’s shows a small increase and TLSF’s seems almost constant.

The execution time is presented in figure 6-8 followed by the table 6-12 which presents the exact values.

The difference between Tcmalloc and Ptmalloc2 is remarkable. Ptmalloc2 requires 87% more time to execute the trace profile. When multiple threads are utilized Ptmalloc2 lacks the performance to keep up with the rest of the allocators. The reasons are:
• The locking mechanism used
• The use of shared arenas
• The contention each time thread wants to lock an arena.

Tcmalloc does not face those problems, since each thread has its own memory area to serve the requests.

TLSF on the other hand, manages to handle the increasing number of threads better than Ptmalloc2, while the performance difference between this allocator and the Tcmalloc has decreased from the previous scenario.

So far Tcmalloc provides better scalability for the increased multithreading environment. Even though, it has higher fragmentation, it is not blown up through the different scenarios but rather stays at a constant amount.

Table 6-13 presents the system calls

<table>
<thead>
<tr>
<th></th>
<th>MMAP calls</th>
<th>SBRK calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>30022</td>
<td>1077</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>30031</td>
<td>4</td>
</tr>
<tr>
<td>TLSF</td>
<td>30002</td>
<td>3</td>
</tr>
</tbody>
</table>

This scenario stresses the allocators and this leads to them making excess usage of system calls to satisfy all the requests. All allocators use more than 30000 system calls, both mmap and sbrk, to handle the increased amount of dynamic memory activity. TLSF does not allow the programmer to manually set a value larger than ~1GB for the default pool size and this is why it calls the mmap function several times. Again note that these calls do not only correspond to the allocator; creation of a thread requires mmap calls, for instance.
6.5 Fifth Scenario

In this scenario 11 additional threads were utilized while TLSF pools size was kept at the same value as before. Figure 6-9 shows the live memory and the memory used by the allocators.

![Figure 6-9 Fifth Scenario Memory Usage Over Live Memory](image)

We already can see that the percentage of excess memory usage will be almost the same as the previous scenarios, table 6-14 shows the maximum values and the fragmentation percentage.

**Table 6-14 Fifth Scenario Maximum Memory Size and Fragmentation**

<table>
<thead>
<tr>
<th></th>
<th>Maximum Size in KBytes</th>
<th>Difference compared to Live Memory</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Memory</td>
<td>1402636</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ptmalloc2 Heap Size</td>
<td>1495328</td>
<td>92691</td>
<td>6.61%</td>
</tr>
<tr>
<td>Tcmalloc Heap Size</td>
<td>1548280</td>
<td>145643</td>
<td>10.38%</td>
</tr>
<tr>
<td>TLSF Heap Size</td>
<td>1533148</td>
<td>130511</td>
<td>9.30%</td>
</tr>
</tbody>
</table>
As in the previous scenario the fragmentation effect is almost unaffected by the number of threads. We have a small decrease in Tcmalloc’s memory waste, while the other two allocators have the same performance.

Figure 6-10 presents the execution time.

![Figure 6-10 Fifth Scenario Execution Time](image)

Table 6-15 presents the exact values and the percentage difference.

<table>
<thead>
<tr>
<th>Allocators</th>
<th>Execution Time In Seconds</th>
<th>Difference compared to Tcmalloc</th>
<th>Percentage Difference compared to Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptmalloc2</td>
<td>45,875697788</td>
<td>22,921199658</td>
<td>99,85%</td>
</tr>
<tr>
<td>Tcmalloc</td>
<td>22,95449813</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TLSF</td>
<td>30,35319539</td>
<td>7,39869726</td>
<td>44,06%</td>
</tr>
</tbody>
</table>

The results are clear. Tcmalloc executes the allocation pattern in half the time compared to Ptmalloc2. We can notice a massive performance difference between the two allocators. The lockless access to local thread cache of Tcmalloc seems to be the best solution in terms of performance for multithreading environments, when
compared to Ptmalloc2 implementation. TLSF has worse performance than the previous scenario; however it manages to be faster than Ptmalloc2.

It is more than clear that using Tcmalloc for multithreading dynamic memory allocation is preferred if execution time is the top priority. Table 6-16 presents the system calls.

<table>
<thead>
<tr>
<th>Table 6-16 Fifth Scenario System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Table 6-16 Fifth Scenario System Calls</strong></td>
</tr>
<tr>
<td><strong>MMAP calls</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Ptmalloc2</td>
</tr>
<tr>
<td>Tcmalloc</td>
</tr>
<tr>
<td>TLSF</td>
</tr>
</tbody>
</table>

We again notice that the system calls have increased significantly over the previous scenario. For Ptmalloc2, additional threads mean more arenas and more arenas means more mmap calls when a sub heap inside that arena is full.

### 6.6 Five Scenarios Conclusion

The conclusion for the first five scenarios is that Tcmalloc is more suited for multithreading programming since it has better threading scalability over the other two algorithms in terms of execution time. However, it produces the highest amount of fragmentation.

If the application’s priority is the minimum memory usage then the Ptmalloc2 allocator is the most suitable. In all the scenarios it produced the minimum fragmentation, requesting less than 6.70% additional memory to satisfy the live memory. Tcmalloc wastes more memory and may need up to 80% more memory compared to Ptmalloc2.

In case we need a real time allocator, TLSF is the one to use. It has scalable execution time while the fragmentation is less than the one observed at Tcmalloc. Additionally, it can handle the multithreading application faster than Ptmalloc2. One can argue that it is the best all-around allocator.
6.7 Ptmalloc2 Parameters

Through this section the performance of Ptmalloc2 behavior is analyzed, given the tunable parameters it provides to the user. The analysis will focus on the behavior between the default allocator’s implementation and the behavior after the optimizations.

Two distinct parameter settings are created with different purpose in mind, based on which goal we want to achieve, faster execution time or lower fragmentation. According to the goal the corresponding parameters were set to the suitable values. Table 6-17 presents the parameters and the values for each variation.

Table 6-17 Ptmalloc2 Profile Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>CPUProfile</th>
<th>MemoryProfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>MX_FAST</td>
<td>80</td>
<td>Default</td>
</tr>
<tr>
<td>TRIM_THRESHOLD</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>MMAP_MAX</td>
<td>0</td>
<td>Default</td>
</tr>
<tr>
<td>M_TOP_PAD</td>
<td>1024<em>1024</em>1024(1GB)</td>
<td>0</td>
</tr>
<tr>
<td>malloc_trim</td>
<td>Disable</td>
<td>Enable</td>
</tr>
</tbody>
</table>

Table 6-18 summarizes the fragmentation results for each different scenario.

Table 6-18 Ptmalloc2 Profiles Fragmentation Results

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CPUProfile</th>
<th>MemoryProfile</th>
<th>Default</th>
<th>Tcmalloc</th>
<th>TLSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>6.68%</td>
<td>6.20%</td>
<td>6.46%</td>
<td>11.01%</td>
<td>9.29%</td>
</tr>
<tr>
<td>Second</td>
<td>6.70%</td>
<td>6.22%</td>
<td>6.47%</td>
<td>11.49%</td>
<td>9.29%</td>
</tr>
<tr>
<td>Third</td>
<td>6.70%</td>
<td>6.20%</td>
<td>6.49%</td>
<td>10.86%</td>
<td>9.30%</td>
</tr>
<tr>
<td>Fourth</td>
<td>6.75%</td>
<td>6.25%</td>
<td>6.62%</td>
<td>10.52%</td>
<td>9.29%</td>
</tr>
<tr>
<td>Fifth</td>
<td>6.76%</td>
<td>6.26%</td>
<td>6.61%</td>
<td>10.38%</td>
<td>9.30%</td>
</tr>
</tbody>
</table>

We can note that the fragmentation results did change using different parameter values. An increase is noticed for the CPUProfile (3.45% on average) while the MemoryProfile shows a decrease of wasted memory (4.20% on average). This
happens mainly due to the frequent calls to the trimming mechanism while no additional memory is requested each time the `sbrk` system call is used. It is worth to mention that many of these parameters only apply for the main arena which is controlled by `sbrk`. Table 6-19 shows the execution time results in seconds.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CPUProfile</th>
<th>MemoryProfile</th>
<th>Default Ptmalloc2</th>
<th>Tcmalloc</th>
<th>TLSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>2,040673119</td>
<td>2,284160818</td>
<td>2,23256285</td>
<td>1,826141663</td>
<td>2,439340691</td>
</tr>
<tr>
<td>Second</td>
<td>5,255585262</td>
<td>5,825501628</td>
<td>6,166512170</td>
<td>4,664235477</td>
<td>5,661942637</td>
</tr>
<tr>
<td>Third</td>
<td>8,451330038</td>
<td>12,000505537</td>
<td>11,260893692</td>
<td>8,10026102</td>
<td>10,16481476</td>
</tr>
<tr>
<td>Fourth</td>
<td>27,364226350</td>
<td>31,244709392</td>
<td>30,704147983</td>
<td>16,40313736</td>
<td>19,72573974</td>
</tr>
<tr>
<td>Fifth</td>
<td>41,050878266</td>
<td>45,958673065</td>
<td>45,875697788</td>
<td>22,95449813</td>
<td>30,35319539</td>
</tr>
</tbody>
</table>

Some interesting results are shown here. The CPUProfile can almost match the Tcmalloc performance, when up to 3 additional threads are utilized. This performance benefit does not come with the penalty of fragmentation, since at that particular scenario CPUProfile has 38% less fragmentation compared to Tcmalloc. When compared to the default Ptmalloc2 implementation there is definitely an increase in the execution time (17% on average).

The MemoryProfile can be considered in execution time identical to the default Ptmalloc2, but using less memory.

Table 6-20 presents the `MMAP` system calls for the different profiles.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CPUProfile</th>
<th>MemoryProfile</th>
<th>Default Ptmalloc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Second</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Third</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Fourth</td>
<td>30022</td>
<td>30022</td>
<td>30022</td>
</tr>
<tr>
<td>Fifth</td>
<td>70022</td>
<td>70022</td>
<td>70022</td>
</tr>
</tbody>
</table>

When a large amount of threads is used, the `mmap` calls drastically increase to create all heaps in the thread arenas.
Table 6-21 shows the `sbrk` system calls.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CPUProfile</th>
<th>MemoryProfile</th>
<th>Default Ptmalloc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>3</td>
<td>34487</td>
<td>1073</td>
</tr>
<tr>
<td>Second</td>
<td>3</td>
<td>34407</td>
<td>1067</td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>34433</td>
<td>1090</td>
</tr>
<tr>
<td>Fourth</td>
<td>3</td>
<td>34460</td>
<td>572</td>
</tr>
<tr>
<td>Fifth</td>
<td>3</td>
<td>18573</td>
<td>1058</td>
</tr>
</tbody>
</table>

The results show that CPUProfile has minimized the number of `sbrk` system calls. The reason is that the tuned Ptmalloc2 does not call trimming at all, leading to no `sbrk` calls with a negative value to return the top pad memory chunk back to the operating system. In each `sbrk` call, the CPUProfile requests extra `M_TOP_PAD` memory to minimize the excess `sbrk` call, and this is captured in the results.

On the other hand, MemoryProfile, calls excessively the trimming algorithm and the `malloc_trim` function. This is the reason we have so many `sbrk` calls.

To conclude, optimizing Ptmalloc2 behavior can gain benefits in execution time terms. However, when a larger amount of threads is utilized, the performance decreases radically compared to Tcmalloc.

### 6.8 Tcmalloc Parameters

In the previous chapter we introduced several Tcmalloc environment variables that may affect its performance. In this section the performance dependency on those parameters is presented. The main competitor is the default Tcmalloc implementation which uses a mix of the system calls. It should be noted that Tcmalloc does not completely comply to the environment variables values.

The first parameter `TCMALLOC_SKIP_MMAP=1` disables the usage of `mmap` calls to request additional memory from the system, while the `TCMALLOC_SKIP_SBRK=1` disables the usage of `sbrk` calls.
Table 6-22 shows the fragmentation results compared to the default Tcmalloc implementation.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>SKU_MMAP</th>
<th>SKU_SBRK</th>
<th>Default Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>11,01%</td>
<td>11,07%</td>
<td>11,01%</td>
</tr>
<tr>
<td>Second</td>
<td>11,48%</td>
<td>11,48%</td>
<td>11,49%</td>
</tr>
<tr>
<td>Third</td>
<td>10,86%</td>
<td>10,85%</td>
<td>10,86%</td>
</tr>
<tr>
<td>Fourth</td>
<td>10,54%</td>
<td>10,53%</td>
<td>10,52%</td>
</tr>
<tr>
<td>Fifth</td>
<td>10,41%</td>
<td>10,41%</td>
<td>10,38%</td>
</tr>
</tbody>
</table>

There is no real difference to the fragmentation results. Tcmalloc will request the same amount of memory no matter the system call to feed the underlying caches.

Execution time in seconds is presented in the table 6-23.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>SKU_MMAP</th>
<th>SKU_SBRK</th>
<th>Default Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1,772889502</td>
<td>1,729553238</td>
<td>1,826141663</td>
</tr>
<tr>
<td>Second</td>
<td>4,550674092</td>
<td>4,512883612</td>
<td>4,664235477</td>
</tr>
<tr>
<td>Third</td>
<td>7,896820720</td>
<td>8,023947113</td>
<td>8,10026102</td>
</tr>
<tr>
<td>Fourth</td>
<td>16,830561543</td>
<td>16,890453551</td>
<td>16,40313736</td>
</tr>
<tr>
<td>Fifth</td>
<td>24,356924013</td>
<td>24,780461723</td>
<td>22,95449813</td>
</tr>
</tbody>
</table>

When sbrk calls are used, we can note a minor improvement (5% on average) over the default implementation and for up to 3 additional threads. The reason may be that mmap has to write the page before it returns it with zeroes. However for 7 and 11 additional threads the optimizations are slower compared to the default implementation.

Finally table 6-24 shows the number of mmap system calls while table 6-25 presents the sbrk calls.
A minor increase can be noticed when the `SKIP_SBRK` is set for the first scenario where only the main thread is used, since `mmap` call is used. However, Tcmalloc does not fully comply with the environment variables as shown.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>SKIP_MMAP</th>
<th>SKIP_SBRK</th>
<th>Default Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>27</td>
<td>152</td>
<td>27</td>
</tr>
<tr>
<td>Second</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Third</td>
<td>30</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>Fourth</td>
<td>30013</td>
<td>30022</td>
<td>30031</td>
</tr>
<tr>
<td>Fifth</td>
<td>70031</td>
<td>70022</td>
<td>70031</td>
</tr>
</tbody>
</table>

For the Tcmalloc parameterization it seems that the allocator does not follow the set values very strictly and when it does there are not really any benefits. Additionally, there are some experimental variables that are not mentioned further here since they introduce instability to the allocator’s behavior.

### 6.9 TLSF Parameters

The main tunability TLSF offers is the option whether to use `mmap` or `sbrk` system call to grow the default memory pool. Also, the programmer has the control over the size of the default pool size. In our case this option was set according to the live memory and the number of threads. By default TLSF is compiled to use the `mmap` system call.
In this section we will evaluate the performance of using the `sbrk` call and compare it to the default behavior. The pool size is again set based on the live memory and the number of threads. It must be mentioned that, as Tcmalloc, TLSF also takes its parameters as an indication of how to behave and does not fully comply with them.

Table 6-26 shows the fragmentation values of using `sbrk` instead of `mmap`.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Use <code>sbrk</code></th>
<th>Default TLSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>9.30%</td>
<td>9.24%</td>
</tr>
<tr>
<td>Second</td>
<td>9.30%</td>
<td>9.29%</td>
</tr>
<tr>
<td>Third</td>
<td>9.29%</td>
<td>9.30%</td>
</tr>
<tr>
<td>Fourth</td>
<td>9.29%</td>
<td>9.29%</td>
</tr>
<tr>
<td>Fifth</td>
<td>9.29%</td>
<td>9.30%</td>
</tr>
</tbody>
</table>

There is no real difference in the fragmentation between the different calls.

Table 6-27 presents the execution time of all scenarios compared to the default behavior and the behavior of Tcmalloc.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Use <code>sbrk</code></th>
<th>Default TLSF</th>
<th>Tcmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>2,634534856</td>
<td>2,439340691</td>
<td>1,826141663</td>
</tr>
<tr>
<td>Second</td>
<td>5,38624546</td>
<td>5,661942637</td>
<td>4,664235477</td>
</tr>
<tr>
<td>Third</td>
<td>11,16686062</td>
<td>10,16481476</td>
<td>8,10026102</td>
</tr>
<tr>
<td>Fourth</td>
<td>21,95772435</td>
<td>19,72573974</td>
<td>16,40313736</td>
</tr>
<tr>
<td>Fifth</td>
<td>33,06929</td>
<td>30,35319539</td>
<td>22,9549813</td>
</tr>
</tbody>
</table>

The performance difference is due to how the allocator requests additional memory. When `mmap` is used the requested size is rounded up to avoid extra system calls. On the other hand, using `sbrk` only the allocation size is requested by the Operating System leading to extra system calls.
Table 6-28 illustrates the usage of `mmap` calls while Table 6-29 the usage of `sbrk` calls.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Use <code>sbrk</code></th>
<th>Default TLSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Second</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Third</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>Fourth</td>
<td>30024</td>
<td>30022</td>
</tr>
<tr>
<td>Fifth</td>
<td>70024</td>
<td>73725</td>
</tr>
</tbody>
</table>

There is no distinct difference except the last scenario where 5% less `mmap` calls are used.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Use <code>sbrk</code></th>
<th>Default TLSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Second</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Fourth</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Fifth</td>
<td>4033</td>
<td>3</td>
</tr>
</tbody>
</table>

In the last scenario, the default memory pool is exhausted and this is why the number of `sbrk` calls is so high.

To conclude on TLSF optimizations, there is a small performance boost when `mmap` system call is used to grow the default memory pool size.

We have to mention that the size of the default pool, would affect the overall execution time and fragmentation performance as well as the amount of system calls. Since there is no default pool we chose to adjust this size based on the number of threads and the live memory.
Chapter 7 Conclusions and Discussion

Through the previous chapter the performance of the three different allocators was evaluated in terms of fragmentation and execution time.

We can clearly draw the conclusion that Tcmalloc is the fastest allocator and also has the best scalability when multiple threads are utilized. Its local thread caches were frequently used to satisfy the requests. If we aim for a multithreading application that only focuses on the performance, this allocator should be used.

However, its performance comes with the excessive memory usage penalty, since it can consume up to 70% more memory compared to the Ptmalloc2 implementation.

Ptmalloc2 cannot scale well enough when multiple threads are utilized. The arena sharing mechanism makes the allocator the slowest one in terms of execution time. However, it excels in minimizing memory fragmentation.

The TLSF allocator is placed between the two of them in terms of both execution time and fragmentation. If we need real time predictability and bounded response time, this allocator comes in handy.

The parameters that can alter the behavior of the allocators were also analyzed. Under certain circumstances and for the particular trace file we used, the Ptmalloc2 allocator can be pretty close to the Tcmalloc’s execution performance while consuming less memory. These optimizations should be considered when we have to deal with an application which uses few threads. TLSF again, is between the two of them, balancing fragmentation and execution time performance.

The major drawback of TLSF is the need for source code access in order to replace all the malloc and free functions with the ones that the allocator uses. Furthermore, it is up to the programmer to figure out the size of the pool to use in order to optimize the performance and minimize the need for excess system calls. However, it is a very balanced real time application allocator with constant O(1) allocation time and bounded memory wastage.

The excess system calls for all the allocators when more than 4 threads were used can boil down to the used CPU’s architecture. The particular processor has 2 cores and
has the ability to create 4 threads. Every time more threads are utilized, the Operating System has to synchronize their execution which leads to the excess number of system calls, especially `mmap` calls. In case we had a more advanced CPU, which could support more threads, the excess system calls would decrease but the performance difference, both fragmentation and execution time, between the allocators would be the same.

Additionally, the behavior of the allocators applies only to the particular trace file and the allocations and de-allocations event sequence we used. This means that there may be a sequence of events that under different scenarios may favor one allocator over the other.

The application developer should consider the actual application’s memory profile and behavior before choosing the suitable allocator.

Finally, we should state that the goals of the thesis were successfully achieved, since the different allocators were tested under different scenarios where the execution time and the fragmentation performance were evaluated.
APPENDIX A
Driver Program Analysis

Once the trace file is created by the memwatch program it can be an input to our driver program. The first thing to consider is to sort the trace file based on the allocation time (it was observed that the memwatch may not sort its results), while discarding unnecessary fields of that trace log file.

A sorted trace file is created using the same name including a Sorted_ prefix. This Sorted file is used to simulate the behavior of the real program. Each line in that file contains the size, the allocation and the de-allocation time, in case there is no de-allocation time (live memory over the whole life of the program) this field is left blank.

The next step is to start reading each line of the sorted file and either allocate or deallocate the listed size. To achieve this an array was created where each index contains each line of the file but in terms of variables now. Using the number of variables we can conclude if it is a malloc or free request, since malloc will also include size and allocation time.

A loop is created to traverse this array and based on the number of variables the malloc or free function is called. For each array’s position (which corresponds to a file’s line), a check is made to determine whether the current position’s de-allocation time is earlier or later than the next one’s. In case it is later it is stored to the live memory array, introduced previously, which means that current position allocation is to be freed after the next position allocation and until it is deallocated it is considered live memory, if it is not, then that memory is immediately freed (for a reminder the file was sorted based on the allocation time), which means that current’s position allocation is to be freed before next allocation time.

Furthermore, for each line the live memory array is searched to check whether the current line’s allocation time is later than any of the deallocation time in the live memory array’s indexes. If this happens it is time for that allocation to be deallocated.
Multithreading is supported by the driver program. The user can choose whether to use more than one thread. For each extra thread, a worker is created to execute the trace file, as in the main thread.

Finally, the user can define how many times to execute the main loop event, of traversing the array. Each complete loop indicates a complete execution of the tracing file.

The name of the trace file to use, the number of iterations over the trace file and the number of threads are user defined in the command line.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.
According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Författarens för- och efternamn]