

Ivan Stanev Stanev

Creating an x86 kernel

MSci Hons Computer Science  
(with Industrial Experience)

20th March 2015

“I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Department’s use of plagiarism detection systems in order to check the integrity of assessed work.

I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.”

Date: 20th March 2015

Signed:

## **ABSTRACT**

This project is focused on developing an operating system kernel for the x86 processor architecture, implementing, testing and comparing a variety of algorithms, and providing users with a basic interface for communication with the system. The development process encompasses the usage of a variety of low-level programming languages, utilisation of virtualisation software, and adoption of well-studied theoretical principles in the field of Computer Science. Novel methods for kernel memory management are presented and discussed.

# Table of Contents

<b>1 Introduction</b>	<b>4</b>
<b>2 Background</b>	<b>6</b>
2.1 General Information	6
2.2 System Design	9
2.3 Multi-threading and Multi-core Programming	10
2.5 Existing Systems Overview	12
<b>3 Design</b>	<b>15</b>
3.1 Overview of the i86 Kernel Architecture	16
3.2 Memory Management in the i86 Kernel	19
<b>4 Implementation</b>	<b>25</b>
4.1 Boot Loading and Protected Mode	25
4.2 The Global Descriptor Table	27
4.3 The Interrupt Descriptor Table	29
4.4 Standard C Library Functions	31
4.5 Basic Drivers -- Video and Keyboard	32
4.6 Memory Allocation Schemes and Paging	33
<b>5 System Operation</b>	<b>38</b>
<b>6 Testing and Evaluation</b>	<b>39</b>
<b>7 Conclusion</b>	<b>44</b>
<b>References</b>	<b>46</b>
<b>Appendix</b>	<b>48</b>
A Common phrases	48
B Project Gantt Chart	49
C Project Proposal	50

**Working documents**

<http://www.lancaster.ac.uk/ug/stanev/>

# 1 Introduction

Developing a kernel is a task that creates many software engineering problems. The kernel is the core of any operating system and governs how it should behave. Since operating systems are multimodal, provide a plethora of functionality and are expected to provide variable options to the end-users, the job of any kernel becomes increasingly complex with every new generation. Nowadays technology improves rapidly, with an increasing demand on a faster and more interactive experience for users, but at the same time systems are required to spare resources and focus on strong optimisation. There is an increasing number of reports in regards to hardware components reaching their operational and structural limits (Timmer, 2014) (Kang, 2009). Developers can no longer hope that users will replace their old processor with a newer generation one so that their software runs faster. Instead, it is broadly recognised that software must be strictly optimised to the smallest detail and fine-tuned for efficiency and speed. This is because the advent of multi-core programming has shifted the notion of linear programming. Novel methods must be adopted, because system speed is no longer a matter of relying on ever-improving hardware -- speed is instead achieved through multiple cores (Sutter, 2005). These novel approaches put new responsibilities on kernel developers, which must ensure adherence to evolving designs in hardware. Any small delays and cuts in performance within the kernel due to incorrectly utilised hardware capabilities have a snowballing effect as the structure of an operating system is mostly layered and dependant on several components. Luckily, performance can be measured, therefore any implemented algorithm can be compared. This is particularly important in areas such as process scheduling, where the kernel determines which process gets to utilise the system's resources, or in memory management, where accesses to physical memory are relatively slow (Drepper, 2007). Arguably this is an area where no absolute answer can be given as there are numerous end-goals within an operating system, so it is not possible to determine the perfect allocation scheme. This is because the advent of multi-core programming raises a new and significant problem -- software unpredictability. Non-determinism breaks the existing programming notion of a linear and predictable execution path (Lee, 2006) and there is little to do in this area in terms of solving the issue. It is possible to apply research and scientific methods to solve similar concerns within an operating system kernel and surprisingly, as it is discussed further in the report, most well-known operating systems are not corrected. It is the goal of this kernel development project to agglomerate known, established and proven concepts, to compare and discuss a variety of solutions and to provide a working example of alternative approaches. Created purely for research purposes, this kernel, named the *i86* kernel, aims to maximise performance in terms of work being done in general. The *i86* kernel aims to service requests as quickly as possible. Focusing on optimising workload distinguishes this kernel from the desktop world and opens many possibilities for innovation.

Showing how all of this can be done is the goal of this development task. The author believes that by applying well-known practices and principles from the field of Computer Science, and by analysing and gathering the vast majority of knowledge on kernel and operating systems development, a stable, simplistic and well-built product is made that can aid in teaching the applied concepts. Also, by building the kernel for the widely available x86 processor architecture, distribution, testing and extensibility are made easy. Additionally, the platform is stable, mature, and extensively documented, making it the perfect candidate for writing an operating system, just as the Linux operating system began only with the support of x86 in 1991 (Bovet and Cesati, 2001).

## 2 Background

The operating system kernel is the core software package that governs a system's resources. As the name implies, it can be visualised as sitting in the middle of an operating system, with the whole system depending on it. The operating system itself sits between the user's application programs and the hardware. The hardware is then exploited to provide numerous services that the end-users need. Usually the internals of an operating system are not obvious to the end-user as there is an enforced abstraction on this. The idea behind it is that the users of the system are mostly concerned with the applications provided and not the underlying architecture with its complications. Application programmers generally do not require direct access to hardware resources and instead focus on the usage of other applications. This is due to the massive complexity of controlling the hardware resources, as well as ensuring compatibility when using the same software tools on different platforms. It becomes clear that these concerns must be handled by an additional 'layer' of software, not only for convenience but also for efficiency and security reasons. There is a set of frequently used services within an operating system to deal with requests for memory and file operations, to execute programs, to control devices, to manage users and to resolve errors (Liu, Yue and Guo, 2011). Not only that but modern computers also consist of several processors, multiple peripheral devices and are required to respond to more than just one user. This immensely complicates the job of the kernel and creates design problems. Since the kernel is required to support a plethora of functionality, it is unclear how it should be structured to meet the variable demand imposed on it (Tanenbaum, 1992).

In general, operating systems' evolution is a direct cause of the gradual improvement of hardware capabilities and functionality. These improvements have put a demand not only on enhancing and modifying existing architectures but also on organising the kernel in new ways. There is a vast amount of ideas that have been tried both commercially and for research purposes, but much of the work fits into these categories: microkernels, multi-threading, symmetric multi-processing, distributed operating systems and object-oriented design. The concepts are introduced in the following sections.

### 2.1 General information

It is essential to analyse what a typical modern computer constitutes of and how the kernel or operating system exploits its available features. This also presents some concepts and terminology that is used further in the report.

The core operational unit of a computer is the processor. A processor consists of several parts: storage for data, manipulation of data through logical instructions, data paths in the form of pipelines, and a clock and synchronisation methods (Tennis, 2007). There are several

levels of storage that a processor uses, each one with increasing access and storage times but also with increasing storage capacity. Figure 2.1.1 (Fiu.edu) is a depiction of these storage units, which are laid out in a hierarchy for the purpose of displaying their properties.

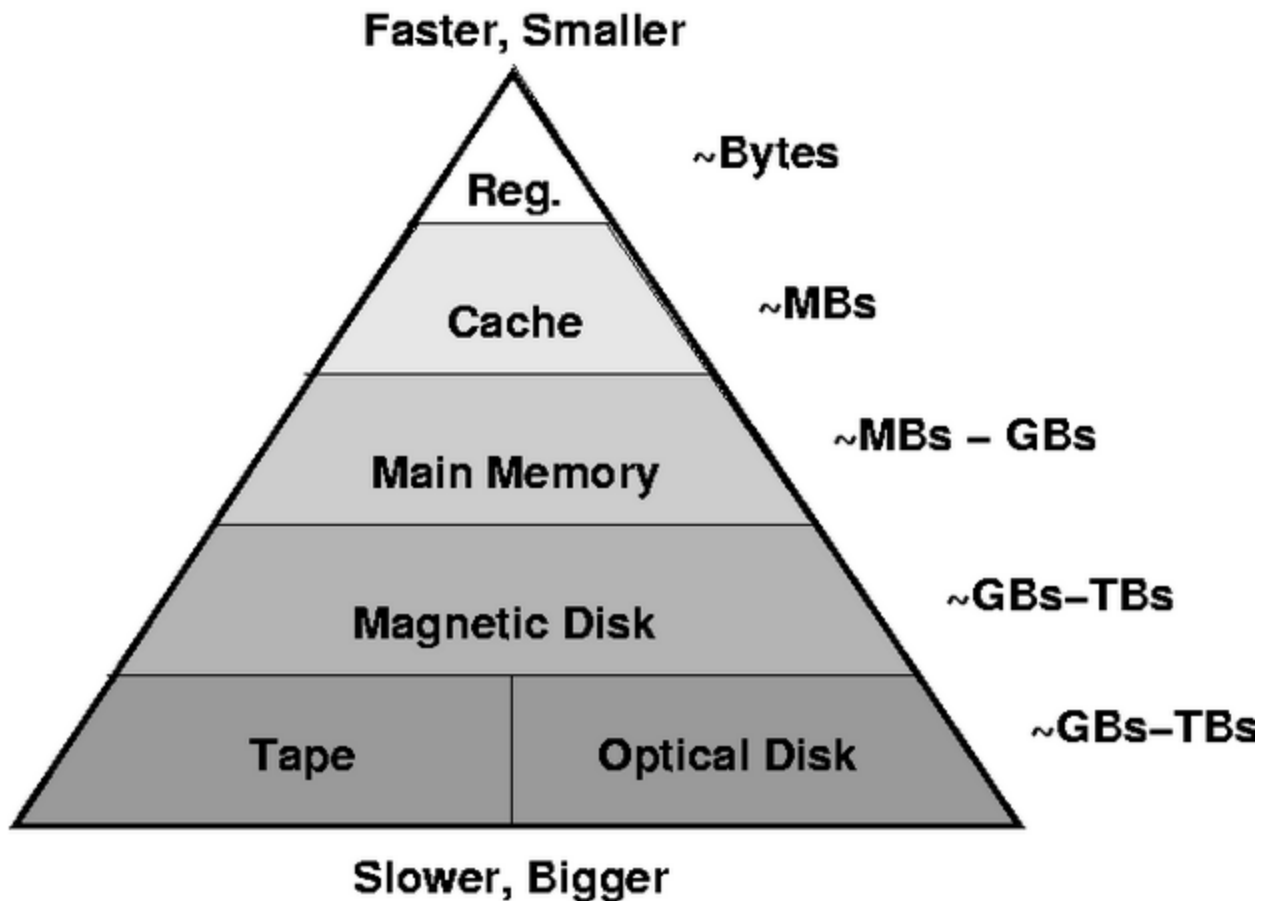


Figure 2.1.1. The memory types of a computing system

Registers, the fewest and fastest elements, are banks for data and are usually made of expensive technology because they need to have short read and write times. They are typically in the form of Static Random Access Memory (SRAM) (Drepper, 2007).

The next type of processor memory is the cache. As discussed in (Drepper, 2007), the cache has been segmented into levels with each level being larger in size but slower in access times. The role of the cache is to store all data written by the CPU. Thus, if the processor needs a data word, a lookup is performed in the cache first. If the data is not found on the cache, it is accessed in main memory. In order to fully utilise the cache of a processor, kernels must be carefully designed so as data accesses do not cause too many misses in the look-up of a cache, resulting in decreased performance.

Further down the memory hierarchy are main memory and secondary storage. Main memory is typically Dynamic Random Access Memory (DRAM) and the processor needs it in order to operate as this is where it fetches instructions from. Secondary storage nowadays moves



from hard disks to solid-state drives. However, main memory and secondary storage are always interlinked by the operating system, particularly in the cases where pages of memory are swapped in and out of RAM -- a technique for using virtual memory.

The world of the computer does not end here as there are additional devices that can be connected. These input-output (I/O) devices sometimes make heavy use of the processor by signalling it to handle their requests, which may happen thousands of times a second. To make things simple, the operating system dedicates specialised software for each device. This software is called a device driver and it is responsible for the correct operation of that particular device (Tanenbaum, 1992).

There are many different computer architectures that have been researched. Of particular significance is the Von Neumann computer architecture (as seen in Figure 2.1.2).

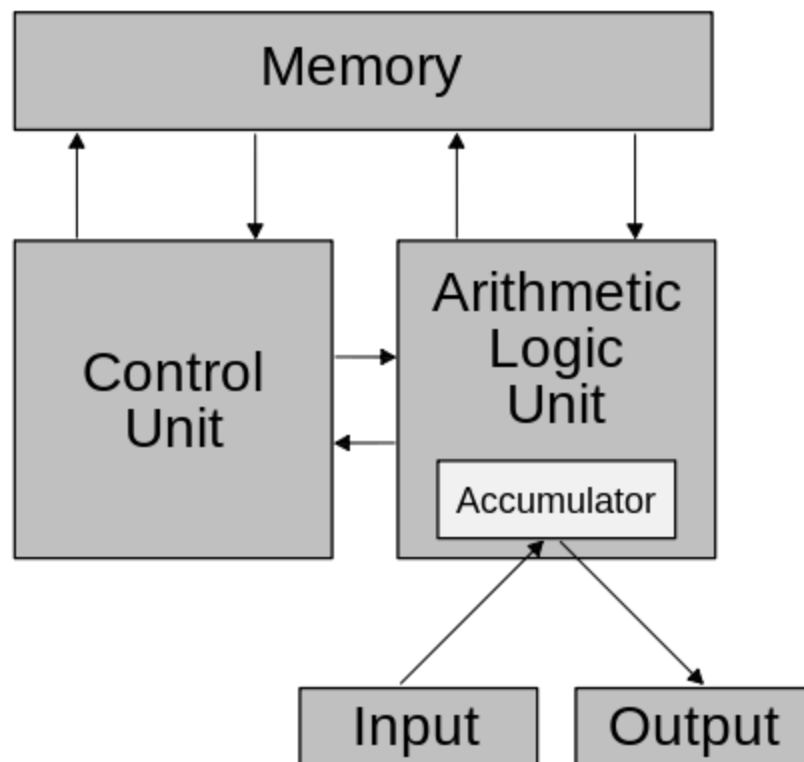


Figure 2.1.2. The Von Neumann computer architecture

With this many different and intricate components, the operating system has great responsibilities and must manage a large amount of resources. How this should be approached is mostly a matter of supporting features required by the end-users and the stakeholders, though approved design choices have been known for a long time.

## 2.2 System design

Over the years there have been avid supporters of two conceptually different kernel architectures; one could say they are the two ends of the spectrum of kernel design -- the monolithic kernel and the microkernel. The monolithic kernel consists of a large bundle of software components that is pre-compiled and cannot change. It includes all the known functionality, such as scheduling, memory management, networking and more. On the other hand, microkernels have the minimalistic role of providing scheduling, process communication and address spaces. The rest is supplied in the form of services or separate modules that run with normal privileges -- they are treated like any other regular application. This approach simplifies implementation and decouples the kernel from its services in terms of development. The model closely resembles that of distributed systems, so this architecture is well-suited for such cases (Stallings, 2001).

Of course in the practical world there are always compromises and a combination of existing methodologies can sometimes be the best solution. Keeping in mind both previously discussed architectures' strengths in certain areas, several systems have evolved to call themselves 'hybrid' kernels. Most commercially available operating systems adopt a variety of architectural solutions and it is rare to see just one model in place. The well-known operating systems, such as Linux, Microsoft and OS X, are largely monolithic for performance reasons, but they have underlying microkernel support or a microkernel structure in some components (Silberschatz, Galvin and Gagne, 2011). Even this is not the complete picture for these systems, as they use additional mechanisms to run their services.

It is argued that the most optimal design approach to building a kernel is having it load separate blocks of code called modules. In this scenario the kernel has only minimal functionality and some core components; the rest is just linking a variety of needed services, distributed into modules. All of this is done in a dynamic fashion, which means that the kernel can decide which modules it needs at run time, instead of having them pre-compiled directly into it, resulting in a large binary. A statically structured kernel is not as versatile as a modular one because any small changes in the kernel code require a new compilation of the whole kernel unit. However, the dynamic approach focuses in finely tuning the minimum core of the kernel (say, memory management and process scheduling) while providing everything else as separate modules that are loaded depending on the needs of the system. Again, this concept is implemented in commercial operating systems. The result is, as mentioned previously, a hybrid system, resembling both ends of the spectrum. It is layered similarly to a monolithic system, but it is more flexible because modules can communicate directly. It also resembles a microkernel architecture, where the kernel is responsible only for maintaining the critical parts of the system, but the modular approach does not have the overhead of inter-process communication as the modules do not require message passing to interact (Silberschatz, Galvin and Gagne, 2011). Figure 2.2.1 shows the structure of the

Solaris operating system, a modularised system, with all of the modules being loaded by a core kernel. The same loading principle is adopted by Linux but for loading device drivers (Rubini and Corbet, 2001).

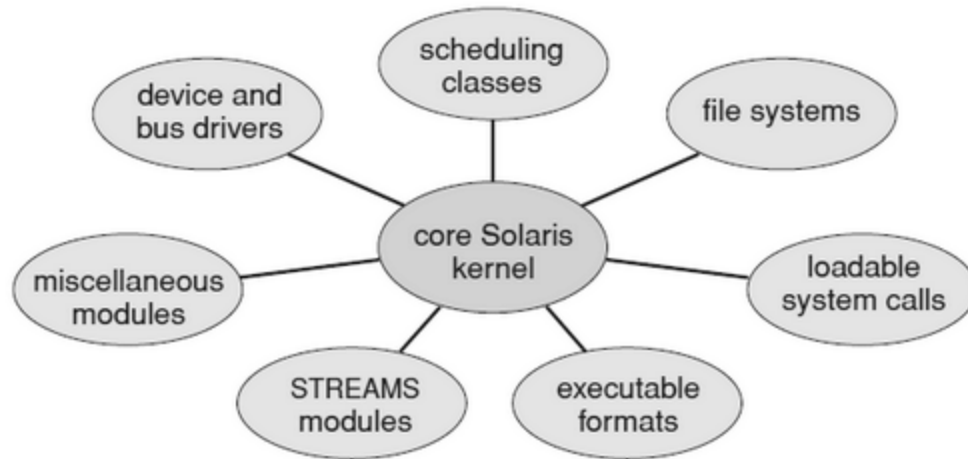


Figure 2.2.1. The Solaris kernel and the modules it loads

### 2.3 Multi-threading and multi-core programming

The concept of multi-threading is explained by first defining what a process is as it is the building block of a runnable system. A process represents a piece of work in execution. Irrespective of what the system type is -- a batch system or a time-sharing one -- the unit of work that is executed in it can be called a process. The structure of a process is complicated once all the details are examined. It contains several sections, such as code and data, it keeps track of the next instruction to be executed, it also operates on a set of processor registers and requires dynamic memory and possibly secondary storage for large sets of data. In contrast, a program is a passive entity -- just code in the form of instructions stored in secondary storage. Once a program is loaded into memory it is considered a process. As such, several processes can refer to the same program, or run the same program, but they are individual entities. Their code sections may be identical but they are loaded in different regions of memory (an example of this is running three different terminals on a desktop computer). Additionally, processes can spawn other ones, or they can duplicate themselves.

However, a process can do more than just one piece of work. A process may consist of several units of execution with distinct goals, which are referred to as threads. Threads are a recent extension to the process model that is present in operating systems and are extremely beneficial in multi-core systems as the cores can run multiple threads simultaneously. However, threads are not separable from processes because they share the address space of the process itself (at least for the most part). What this means is that a process can own

multiple threads that execute different tasks but they share the process' memory and are bound to it. Being the smaller part of the whole, threads can be considered as a sort of light-weight entity that does not have the creation overhead on the system like a process does. This benefits the system as a whole in many ways. As mentioned, threads share a common memory space, which removes the communication overhead that message passing would have if threads were to exchange data. They also require less resources to start working. What is more, they support extensibility and scale very well with multi-core systems that support multi-threading.

Computers have evolved from having a single processor to supporting multiple ones. Nowadays it is possible to see multiple cores on a single hardware chip, each one being treated as a single processor capable of doing work in parallel with the others. The benefits become obvious: in single-processor systems only one thread can run at any point of time. The kernel provides each of the system's threads some time to do work before switching to another one. This happens so fast that to the end-user who is using her computer everything seems to run concurrently -- she can listen to the radio, browse the internet, have text files being opened and have a chat client in the background. All of this may seem concurrent but it is not parallel. The radio process, the browser, the text process and the chat client both take turns to complete a bit of their work. In contrast, multi-core systems allow the multi-programming model to be adopted -- a thread can run on any of the processors available. On a quad-core system, this means that each of the processes mentioned above, considering they have only one thread of execution, utilise one of the four processors, which is both concurrent and parallel. Although an ingenious idea to improve performance, it comes with its own set of design considerations, which become an issue for the programmer. She must decide how to divide tasks within an application to make them runnable on multiple processors, in order to utilise the multi-core implementation. In the best-case scenario all the tasks will run on every core. However, she must also consider if it is beneficial to dedicate a task to a separate core. If the task has little value to the system overall, it is probably better not to spend a significant amount of resources on it. Often a process that is split into multiple tasks requires the tasks to work on the same set of data. This creates additional problems and the programmer must decide how to deal with concurrent accesses, thus preventing, for example, data to be written by two entities at the same time, resulting in one thread overwriting the other's information. The notion of parallelism in multi-core systems implies some non-determinism being present in the whole system. It is not practically possible to predict the flow of execution in a system with multiple cores as there can be several data paths. This results in harder testing and verification of the end program and concurrency bugs can be difficult to identify (Silberschatz, Galvin and Gagne, 2011). What is more, the programmer often does not know the number of system cores she is writing code for, which runs the risk of splitting a process into more threads than the number of present processors. In the case where a kernel provides a fair share of execution time to every thread, this can

create performance bottlenecks and degradation (Christmann, Hebisch and Weisbecker, 2012).

The biggest problem with multi-core programming is that it deviates from the common, deterministic approach of general programming. Instead of providing any real benefits, non-determinism actually forces programmers to create artificial determinism, which is a huge step backwards. Resources that are utilised by multiple executing entities have to be protected and strictly governed, even potentially locked because the order in which they might access the resource is practically unknown. The notion multi-programming does provide significant improvements to performance and work being done but cannot be fully exploited yet.

## **2.4 Overview of the UNIX Operating System**

UNIX is an open-source operating system, which made its history in a two-decade period from 1969 to 1989. Because of its wide availability, as well as people's huge interest in developing it and porting it to different systems, it received special attention in the academic world and was able to grow rapidly. (Liu, Yue and Guo, 2011) argue that the widespread usage of the UNIX system and its well-known design and architecture is what distinguished it from commercial counterparts as a better choice -- competitors provided perfect solutions but rarely exposed their systems' internal workings and such solutions were often constrained, leaving application developers with limited choices. The UNIX architecture is displayed in Figure 2.4.1 (Stevens, 1992), which shows that it is a layered, monolithic system. As such, it comes with all the benefits and drawbacks of monolithic systems, discussed previously.

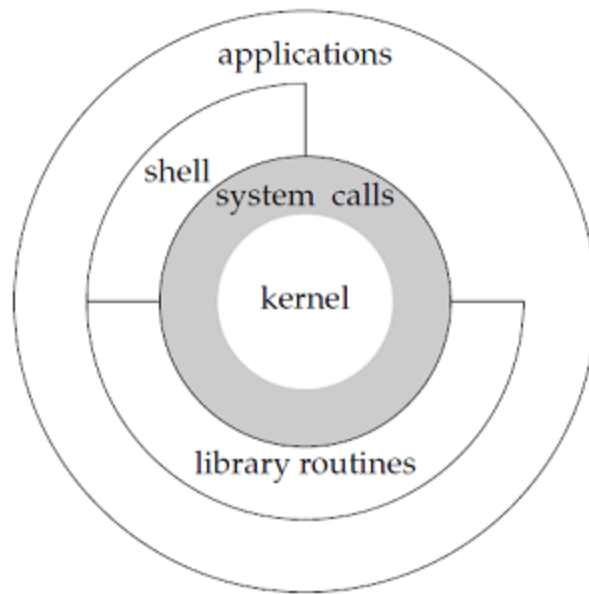


Figure 2.4.1. The UNIX architecture -- a monolithic system

UNIX supports multi-programming with the only active entity in the kernel being the process. Processes can be created, multiplied, synchronised, terminated, and even much more. A feature of UNIX is also the usage of pipes, which are channels for communicating data from one process to another.

The problem of memory management is approached with virtual memory demand paging. What this means is that for a given process that is running in the kernel, only a set of pages are dynamically loaded into memory when needed. For example, if a video player is about to play a movie it need not load the whole file into memory (especially if it is several gigabytes large) but rather it would load parts of it on the fly as needed. In general, when memory is exhausted pages are swapped out to disk and vice versa. The algorithm for swapping is entirely dependent on the underlying implementation but the goal is always to minimise the number of page faults created by constantly swapping memory pages. Memory management in UNIX, as with the UNIX-like operating systems such as Linux (Gorman, 2004), is done with the help of a buddy allocation algorithm. This algorithm works by continuously splitting memory into two blocks of equal size and checking if the block of memory satisfies the memory request. If not, the process is repeated recursively for each newly split block. This is a type of binary algorithm that works quite fast but it is not the best solution, especially for allocating physical memory, because there are other ways of achieving the same goal but faster.

An interesting approach to file organisation in UNIX is that everything is treated as a file -- pipes, sockets (general interfaces), devices, even directories. This unique abstraction makes it

possible for common kernel functions to be used on a variety of files (i.e. `open()` on different I/O devices). In a nutshell, the UNIX OS is displayed in Figure 2.4.2.

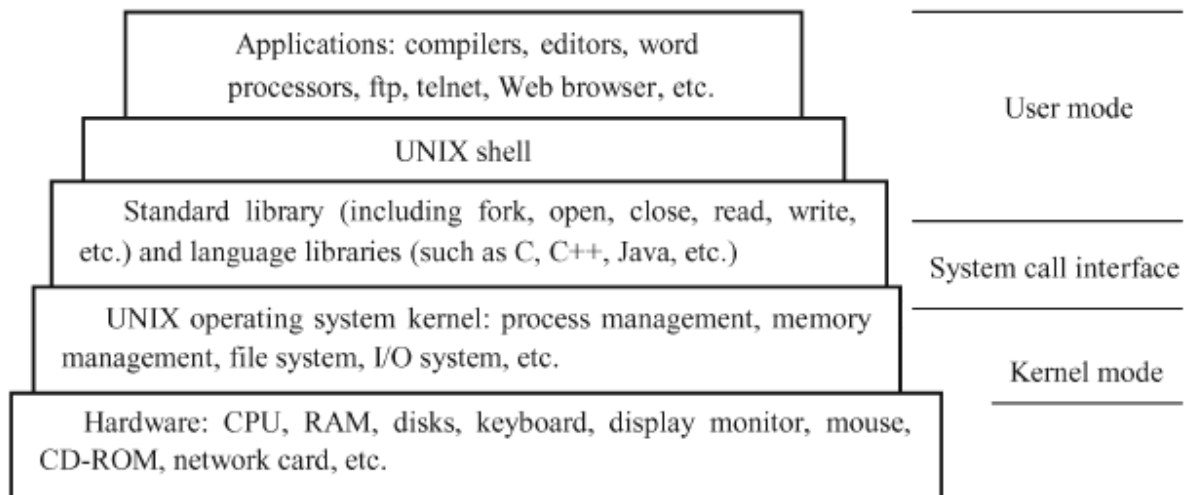


Figure 2.4.2. The UNIX architecture

### 3. Design

The i86 kernel tries to adopt the best design practices and implement them simply and elegantly. One strict design choice is not always plausible because the drawbacks become more prominent the larger the scale of the project becomes. For instance, maintainability and extensibility in monolithic systems is a nightmare on projects with millions of lines of code. On the other hand communication speed becomes a burden in microkernels with several modules that try to exchange messages. Combining both monolithic kernel and microkernel architectures results in a quick, modular, hybrid system. The core of the system is still hard-coded but it is split into blocks for efficiency and extensibility reasons.

It is important to understand that the kernel is designed to run on the Intel processor architecture -- a commonly featured processor in commodity hardware. The x86 architecture is a long line of processor generations with backwards compatibility, which is an important feature. Starting with the i586, Intel introduce the Pentium processor (Intel, 2015). From there on, instruction set and core architecture upwards compatibility with future models is guaranteed, thus the i86 kernel is primarily targeted at i586. Because of their wide availability, thorough documentation and support, and many experimentations by different developers in the field of operating systems, the x86 processors are a suitable candidate for building a simple kernel. An overview of the architecture is shown in Figure 3.1 (Intel, 2015). Important additions to the Pentium (i586 and onwards) over the i486 are branch prediction, separate instruction and data caches, dual integer pipelines and more. Since Intel focus on supporting their products for as long as possible, a large amount of the transistors on the chip are dedicated to providing backwards compatibility with the previous generations of processors.



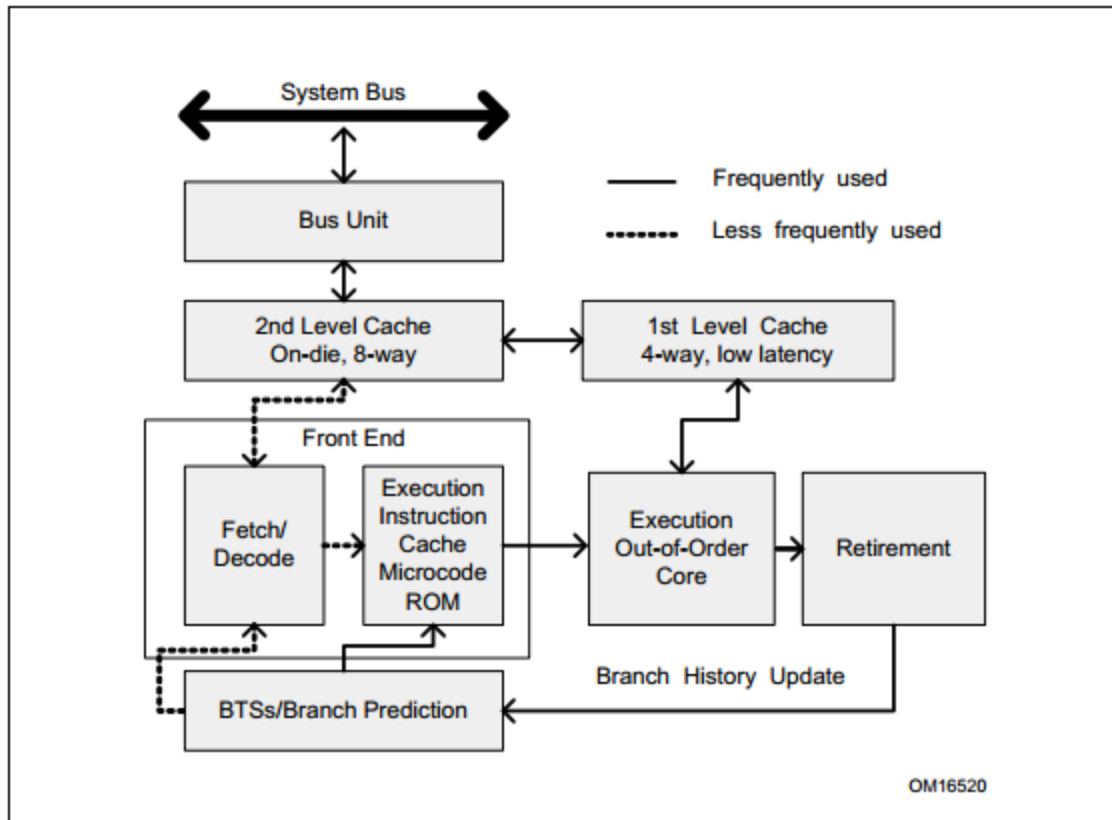


Figure 3.1. Pentium architecture

Even though the i86 kernel is currently designed only for one architecture, care is taken to make it support future extensions and other architectures. There are many tiny details that need to be considered in order to provide a concrete end-product designed for performing any job as efficiently as possible. This is why all design considerations are split into sections; each component has its own world of related problems.

### 3.1 Overall system structure

As noted previously, it is the author's belief that having a mixed architecture works best for most problems. A hybrid design provides the right mixture of encapsulation of components, security, control and development extensibility. The i86 kernel is both modular and layered, with different layers being responsible for a limited set of functionality. Layering provides the necessary security and encapsulation of code and data whilst logically separating the components in the system. Modularisation ensures that no external software interferes with or modifies any of the packages that it is not responsible for. Additionally, this provides fine control of individual packages and changes can be easily made. This isolation also ensures that problems do not propagate through the system, which makes them arguably easier to

trace. A simple overview of the i86 kernel is shown in Figure 3.1.1. It should be noted that layers are not uni-directional -- it is possible for higher-level modules to communicate with lower-level ones; the communication is full duplex.

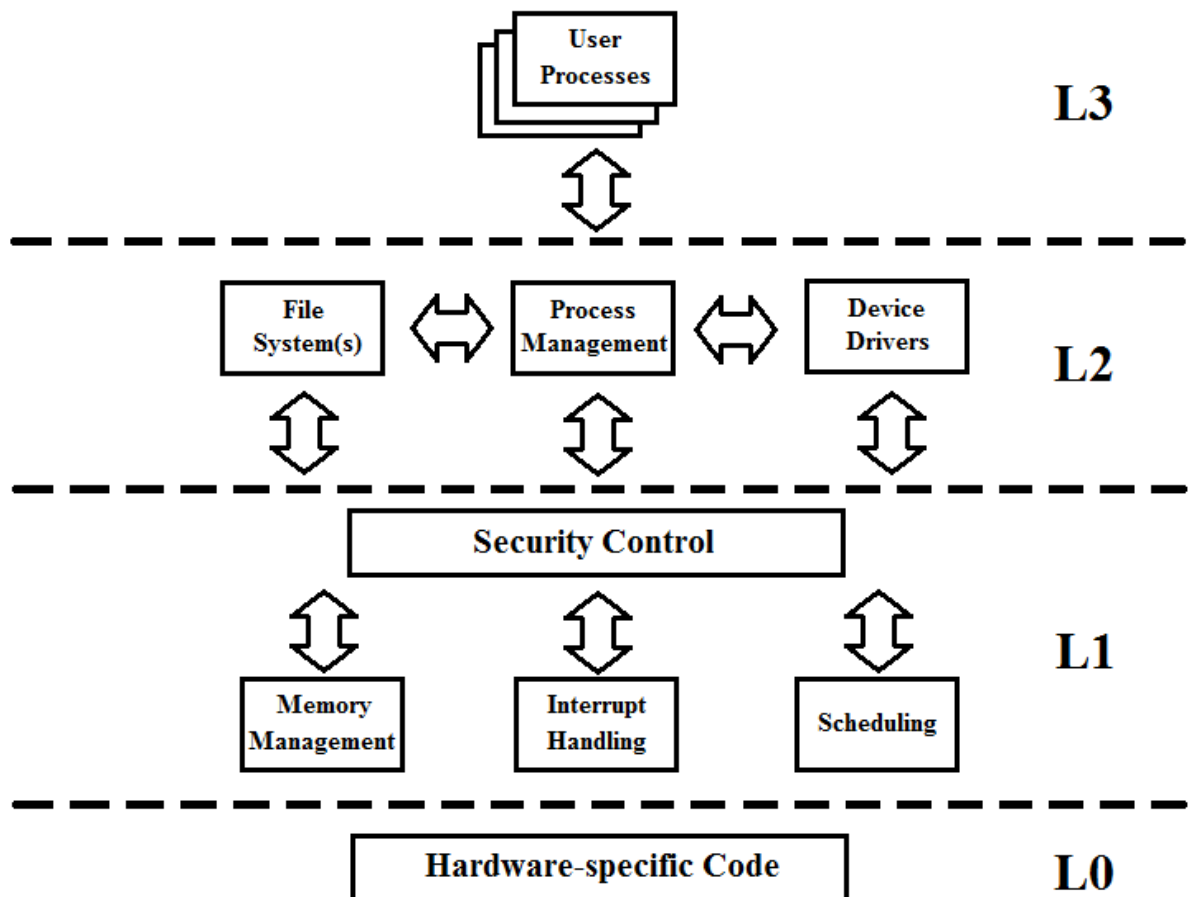


Figure 3.1.1. The conceptual view of the i86 kernel implementation

The lowest level (L0) in the hierarchy displayed is the hardware layer -- this is where the devices reside, such as the processor and interconnected I/O devices. Because of the variety of connected elements from one system to another, software has to accommodate the peculiar workings of these devices. The main concern here is the underlying processor architecture because the system starts with raw code -- programmers cannot enjoy the high-level abstraction provided by most programming languages. Because of this, assembly programming languages have to be used that address the specific architecture that runs the code. Not only that but different architectures also have significantly different setup requirements. This is why a simple abstraction layer is needed to handle the individual setups -- in order to provide a common interface, a solid ground for building up the system.

Responsibilities here, for the x86 architecture, include managing interrupts, data flow control, setting up table structures, switching modes of operation of the processor, and more. The next level sits right above the abstraction layer and makes use of the provided environment. All the core kernel operations are handled here, so speed, efficiency and perfect resource utilisation are of utmost importance. This is where critical work is performed and it is what the system depends on. The L1 layer is unrestricted in terms of resource usage and is trusted with the full availability of the hardware and its features. Additionally, it provides interfaces and building blocks for further system development and extensibility. This is where core operations reside, such as memory management, interrupt handling and process scheduling. The components within this layer also interact with each other with well-defined interfaces (for example, scheduling relies on timer interrupts to switch to other processes). System definitions are presented here -- entities such as processes or pages are supplied by this layer for usage within the system. This is the most critical area in the kernel, so the most focus is on polishing the features here.

Level two (L2) has a more loose structure for the components as users can contribute to the extension to this layer. However, this does not mean that this layer is with user-mode/restricted privileges but rather that the modularity supported here allows for the injection of code, thus extending existing functionality. For example, device driver control is positioned in this layer, which means that drivers are in the form of loadable modules. This area is governed by security policies that ensure data is protected and unaffected by external, unrelated or otherwise malicious software. It should be noted that communication upwards (from lower levels to higher) bypasses security. What this means is that modules that depend on lower-level functionality do not need to strict security rules to govern the data flow -- they are practically none as the low layer software is considered secure and effective. The opposite direction of communication is, in contrast, highly sensitive and modules must be extremely careful with the data they are supplied as it must often be trusted blindly. Security policies come into play to minimise abuse and recognise threats to the integrity of the system. This layer is not concerned only with the security of the system -- this is where general functionality resides, one that is not dependant heavily on the hardware and can be abstracted for common usage.

The final, user-mode layer, is layer three, where the user programs reside. This is the intended and normal operation of the system, with processes running with restricted privileges. Access to system information, sensitive data or privileged functionality is done by using system calls, or an exhaustive application programming interface (API). Execution of code is strictly governed but the users of the system are generally free to do whatever they are willing in order to utilise the most out of it in doing work.

What is important to note is that the number or naming of layers is just a simple convention -- over time their number might change or the structure may be altered. However, the general idea should be the same -- having a layered design combined with extensible and loadable

modules. The rest is just a matter of slicing the whole structure and rearranging it for ease of understanding.

## 3.2 Memory management

One of the most important features of any operating system or kernel is quick, simple to use and efficient memory management. With the introduction of on-chip memory management units, systems are able to provide additional protection mechanisms and performance improvements by employing virtual memory. Virtual memory allows the division of physical memory into equal-sized blocks, called pages, which are then allocated to a variety of processes. The number of pages for a running process need not be large, as processes can receive pages whenever they are in need of additional memory. Additionally, if a program does not fit into physical memory, virtual memory can help transfer pages to secondary storage and swap them in and out whenever needed, thus artificially increasing said physical memory. Finally, every process within the system can seemingly make use of all of the available physical memory (4 GiB for 32-bit systems) by using virtual addresses, while the underlying memory management controls the mappings and access control (Hennessy, Patterson and Arpaci-Dusseau, 2007).

The i86 kernel makes use of the x86 paging facilities and harnesses the power of the available memory management unit. However, due to the design of the x86 architecture, the kernel must additionally use segmentation and this is a feature that cannot be disabled. With segmentation the processor's address space is divided into smaller, isolated segments that can be assigned to individual processes. Because segmentation is of no concern to the i86 kernel other than to provide the environment for 32-bit protected mode, it is initialised with the most minimal settings possible (Intel, 2015). Figure 3.2.1 shows the relationship between segmentation and paging in the x86 architecture. Note that segmentation is the basis of memory protection. Without delving into the details, the i86 kernel initialises entries in the global descriptor table just to have segments for the privileged and unprivileged modes of operation (kernel and user, or ring 0 and ring 3). This also provides the kernel with the ability to switch from 16-bit mode to 32-bit mode, which is done at around this point in the kernel implementation phase.

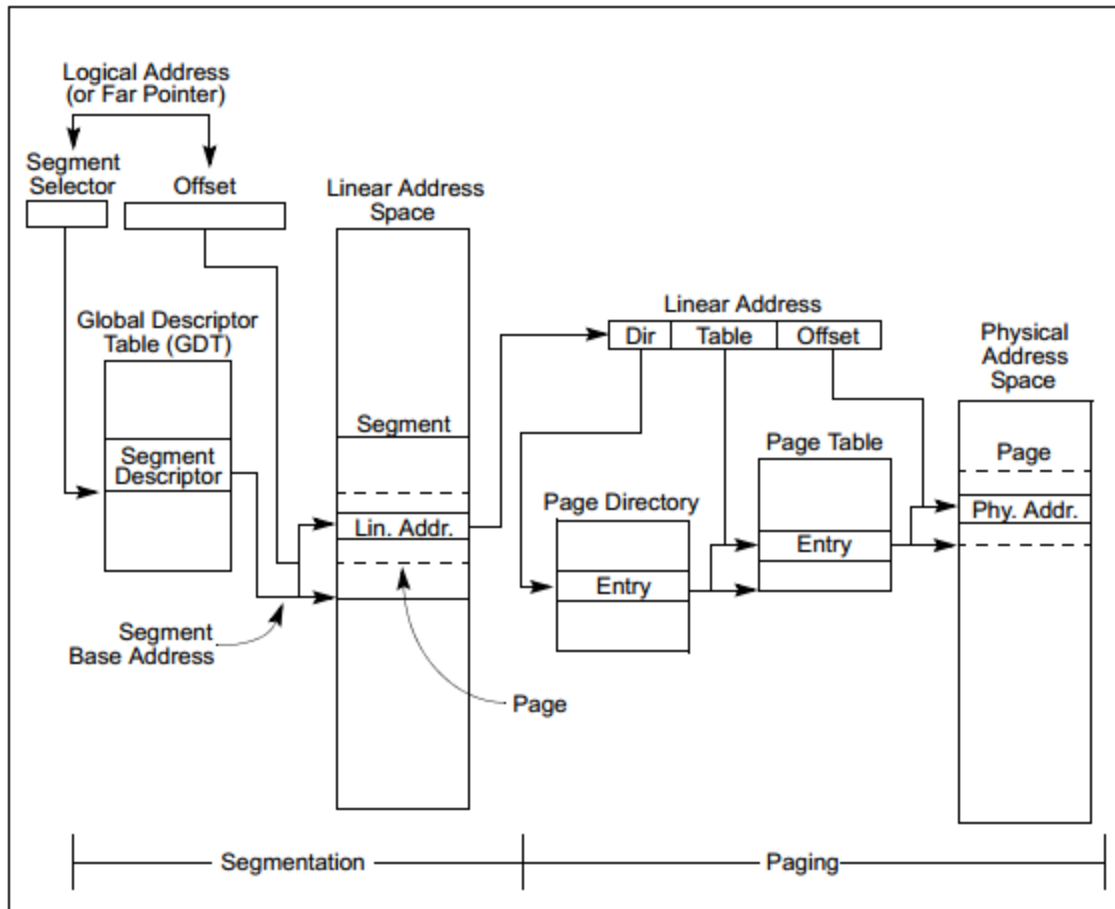


Figure 3.2.1. x86 segmentation and paging (as seen in the Intel software manuals)

In short, any address in the system is first treated as a logical address. This logical address consists of two parts -- a segment selector that is used as an index within the entries in the global descriptor table, and an offset, which is added to an address located in a previously matched segment descriptor in order to locate a byte within the segment. This address, called the base address, plus the offset form a linear address within the system. If paging is not used at this point, this linear address is directly mapped to a physical one. However, since most of the time systems define linear address spaces much larger than the available physical memory, virtualisation mechanisms have to be employed to accommodate the translation of large addresses into smaller, mapped physical addresses.

The other mechanism for managing memory supported by the architecture is paging, which, mentioned earlier, provides a way to simulate large linear spaces with a small quantity of RAM and some secondary storage. The i86 kernel employs paging because of its apparent simplicity -- the i86 architecture provides hardware-based paging, which requires setting up a few entries in software. It is worth mentioning that paging can be either software-based or hardware-based. As the naming implies, the former is performed by the operating system

kernel itself -- whenever the processor encounters a virtual address, it shifts the flow of execution to a specialised piece of software that has to find a corresponding mapping to a physical address in its page table structures. In hardware-based paging this step, called page walking, is purely done by a hardware unit, which may employ different strategies to improve the search as the page tables can be relatively large. The x86 architecture has its own hardware-based paging mechanism, which is utilised by the i86 kernel. What is more, according to the architecture's designers, there is a hardware unit that helps with speeding up address translations -- the translation look-aside buffers (TLB) -- as the page tables are stored in physical memory, where accesses are costly in terms of wasted processor cycles. Recently accessed page table entries are stored in the TLB and if no entry is found there during translation, the aforementioned page walk is performed, which accesses multiple memory locations and may be costly for the processor.

Because of its maturity, effectiveness, improvements and refinements, as well as its long-lasting history, the x86 memory management unit is the preferred choice for the i86 kernel and no software solutions in place of it. The kernel is only concerned with setting up paging for itself (and for its users, of course), which is as simple as providing physical addresses within the page tables that map to virtual ones and marking the entries as valid for translation, the details of which will be thoroughly examined in the Implementation section of the report. Thus, paging provides a strong foundation for building the kernel securely and effectively and provides ways to uniquely manage physical memory. The i86 kernel has a very specific and simplistic approach in this area, which the author believes is one of the most-effective modern implementations.

Physical memory management is of great concern to any kernel as it is the foundation of software creation. As outlined in the Background section of this report, the most popular and widely spread operating systems provide complicated solutions to memory management issues, with algorithms that tend to squeeze even the last bits of performance at the cost of increasing complexity. It is the author's belief that these solutions are present only because of backwards compatibility issues. In other words, the designers of these systems are dealing with commercial products that have years of history behind them. Completely altering a component is unacceptable for them as this may completely erase the product's value. The only solution is modifying the existing underlying mechanism by which the task is achieved but preserving the interfaces that are provided to the end user.

Different memory allocators have been previously discussed. It can be argued that one of the most performance-focused ones, the buddy allocator, is a real possibility as a low-level basis for the i86 kernel. The buddy allocator, however, is archaic and is in no way a good option for a modern operating system. With the presence of specialised and advanced hardware mechanisms, it is a complete waste of system resources to provide solutions that do not achieve the best performance possible. Often the simplest solutions are the best, so the i86 kernel provides a physical memory allocator that achieves constant allocation and

deallocation speed. In scientific terms (mainly computer science), the classification of any algorithm in terms of processing time based on variable input is described by Big O notation. What Big O notation tells about a particular algorithm is its time complexity, or how many steps it would take for it to complete its calculation. Common Big O classes for algorithms are (in regressing order):  $O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n^2)$ , where  $n$  increases unboundedly. For example, if an algorithm is described as  $O(n)$ , this means that the time it takes for it to complete is strictly proportional to its input; it completes in linear time. The best possible scenario is to have an algorithm that performs constantly,  $O(1)$ , independent of input or workset size.

With the help of the paging facilities, the i86 kernel adopts an allocator with constant performance. The way it achieves this is with a stack-based approach that stores all available free addresses. The stack is intended to work with paging and because of this it contains only page-aligned addresses (that is, physical addresses that are the beginning of a page). Because the stack is a “first in -- first out” structure, the obtaining of an address (the `pop()` operation) returns the first free address at the top. Consequently, returning an address back to the stack (the `push()` operation) is also as simple as putting the address at its top. These operations are achieved at a constant rate, because the input data is just a single element and the operations themselves require no special calculations. Figure 3.2.2 is a diagram with the representation of the i86 memory allocator, called `palloc` (which stands for prame allocator), and the operations performed on its stack. Note that the stack top increases or decreases with each deallocation or allocation respectively.

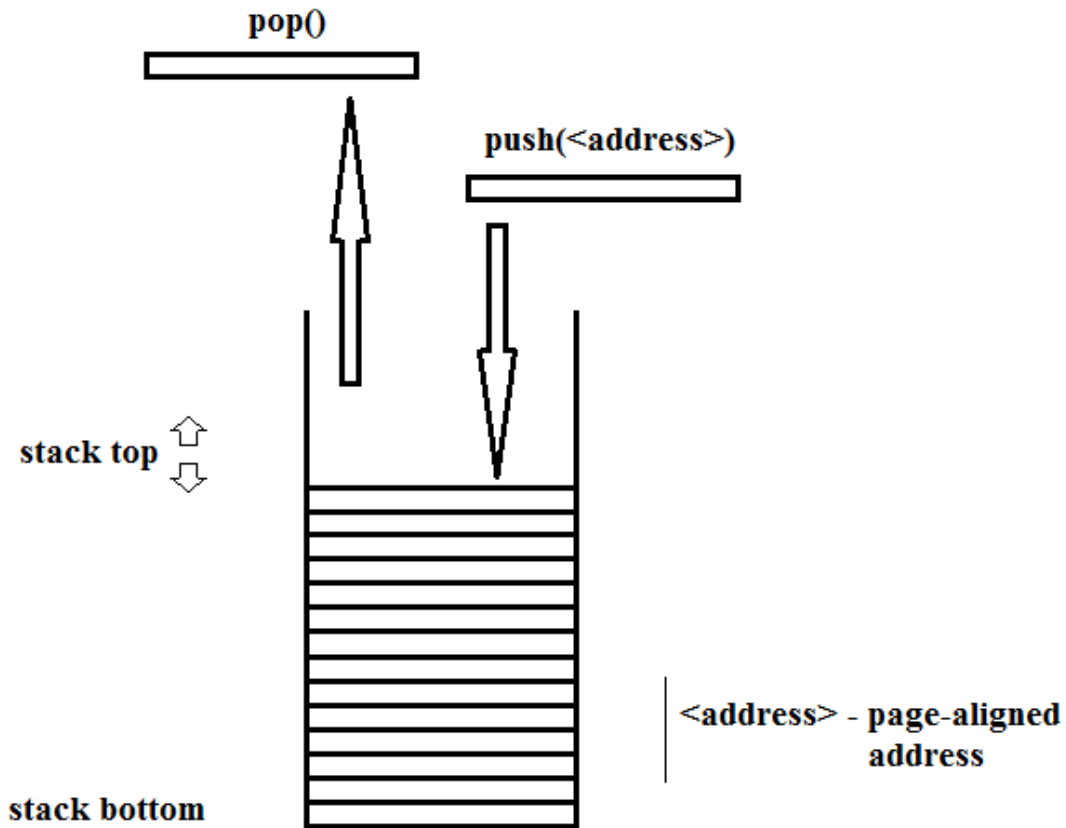


Figure 3.2.2. Page frame stack for the *palloc* allocator

It must be noted that there is an inherent relationship between storage and performance requirements when designing a physical memory allocator. It is not a rule, just an observation, and it means that the more performance-oriented an allocator is (that is, achieving faster allocation/deallocation times), the more storage it requires. For example, a buddy algorithm allocator may store its page-to-address mappings in a single contiguous structure -- a bitmap, with each bit corresponding to a page. This means that an integer, typically but not necessarily represented as a 32-bit value, can represent up to 32 different pages. As far as the value of the bit is concerned, a clear bit would mean the page is empty and a set bit would represent an occupied page. The details of such an implementation are discussed later on in the report. The buddy allocator, being implemented as a tree structure (which will also be mentioned later), takes  $O(\log n)$  work time to complete its calculations. For modern systems, this is still not a good approach.

Logically, from the above-mentioned rule, the i86 kernel's page frame allocator performs fast but requires relatively large amounts of storage compared to other allocation schemes. This is because the stack is filled with as much usable page-aligned addresses as possible, thus the



size of the stack can be calculated by dividing addressable memory size with the page size. Therefore for small page sizes, the stack becomes larger and requires more memory. However, modern systems with x86-based processors (the “desktop” or “laptop” environment) rarely come with small amounts of memory (small being defined as less than a gigabyte), thus the tradeoff of having a bulky stack is insignificant. The allocation speed provides great benefits as it shoves off lots of calculations and frees a significant portion of kernel time that can be utilised for other tasks.

Of course, a single stack space does not solve all the memory requirements of a system. The allocator must provide usable and accessible physical addresses and deal with RAM faults. Other cosmetic but mostly necessary features with this design can include: optimisation for systems that require only particular ranges of addresses, like in systems with non-uniform memory design; supporting larger page sizes; supporting large contiguous, but limited in number, pages; page colouring (mapping physical pages to a variety of cache lines to optimise cache accesses) (Zhang, Dwarkadas and Shen, 2009). The different memory requirements can be split into different stacks, with allocation and deallocation code performing additional checks when required. Luckily, the distribution of pages within a variety of stacks does not impose additional memory requirements for the stack itself, because the number of pages is always constant. The part that consumes the most time in the system is the initialisation part, which is only performed once, before actual system usage. What is more, the checks do not affect the implementation’s time complexity -- it is still  $O(1)$  in Big O notation.

One disadvantage of this approach is that it does not guarantee allocating physically contiguous blocks of memory, or memory below a certain threshold. These requirements are a part of some hardware devices’ operation and they cannot be met with this general-purpose allocator. The kernel’s solution is to provide a zone allocator, `zalloc`, that covers memory requirements up to a predefined range, in multiples of the page size. In the usual case, the allocator covers memory below the 4 MiB mark. The allocator is also very simple, utilises address stacks, and performs at constant time just like the page allocator but has a limited amount of block sizes that it can provide. `Zalloc` is discussed additionally in the Implementation section. With both of these allocators, the kernel tends to satisfy a variety of memory requests.

## 4 Implementation

The implementation of the i86 kernel is divided into logical components that are more or less independent modules. The way they supply data is strictly through interfaces in order to protect their own operation. In this section the flow of development is followed and discussed, with extracts of code supplied to support vague concepts.

The implementation of the kernel is achieved thanks to a basic bootloading setup, as created by (Scott, 2014). The setup includes a shell script for the quick execution of numerous shell commands in terms of compilation and process execution, and a virtual machine emulator, QEMU, for executing the compiled kernel. The development process is completely done in VirtualBox, with a virtual image of Ubuntu Linux, also provided by Dr. Scott. The `kprintf()` function provided as an exercise is also completed and it proves to be an invaluable tool. Thus, the requirements for compiling and running the kernel are: a 32-bit distribution of the Ubuntu operating system, the GNU C Compiler version 4.7.2 or above, targeted at the i586 architecture or higher, and the virtual machine emulator QEMU. These requirements are important for accessing the project files.

This section may also be heavy on language-specific details. Please refer to Appendix A in the report for the definition of commonly used terms and phrases.

### 4.1 Boot loading and Protected Mode

The start of any computer begins in an architecture-specific way. Since the focus of this project is loading the kernel on an x86 machine, the setup can be easily described as it is pre-programmed. On powering up an x86 machine, the microprocessor is instructed to read directly from its erasable programmable read-only memory (EPROM), which contains the basic input/output system (BIOS). The BIOS is a piece of software that the manufacturer of the machine's motherboard has pre-programmed on the read-only memory chip and it contains basic data flow operations in order to communicate with the connected hardware devices. The contents of the BIOS are written at the start of physical memory, so they can be easily accessed, with basic operations being provided to the programmer. The content is mostly functionality mapped into entries in an interrupt table, provided by the BIOS, and at this stage the programmer can communicate with the BIOS through software interrupts. Once the BIOS is in memory, the first 512 bytes of a secondary storage medium are loaded and executed. This is a minute amount of memory and is definitely not enough to get an operating system initialised. Additionally, Intel is a company that is known for supporting backwards-compatibility in its products and this creates even more complications for kernel development. This means that the system starts executing in a mode named Real Mode, where only 1 MiB of physical memory can be accessed and only about 640 KiB of it are

freely available. During the development of the i86 kernel, a specialised bootloader had been created to initialise the running environment for the kernel, which includes reading sectors from secondary storage and loading them into memory, setting up entries in the Global Descriptor Table, activating Protected Mode, and running the actual kernel code. However, the archaic methods of reading data from a secondary storage medium, such as a floppy disk, posed implementation issues and took more time than the project timetable allocated for it, thus the idea of creating a custom loader was abandoned. Additionally, the mechanism for switching on the processor's capabilities of addressing memory above the 1 MiB mark could not be implemented because the underlying hardware did not respond to well-known methods for enabling it. Thus another approach was taken.

The booting process in the i86 kernel is handled by a well-known bootloader -- the Grand Unified Bootloader (GRUB), which takes away the complexities of basic system initialisation. The advantage of this is that any software (in this case -- the kernel) that runs after GRUB is in an initialised and known state -- it can address the full of 4 GiB memory (the most that can be represented by a 32-bit address). Additionally, GRUB provides useful information, such as the amount of memory installed on the system, as well as its reserved/unaccessible ranges. The minimum required by the developer at this point is to set up a stack for kernel usage and jump to the entry point of the kernel.

With the advent of virtualisation software that mimics the behaviour of existing system, it makes sense to adopt it while testing the behaviour of the kernel. Because the process of compiling the kernel, moving it to a medium that can then be loaded by the system, and testing if the code works (which in most cases makes the machine restart and provides no diagnostics) is a tedious, lengthy and unproductive job, the quick solution is to test the run with QEMU. QEMU is a virtual machine emulator that behaves like real hardware and supports the x86 processor architecture, particularly the i386 and above family of processors by Intel. After doing its job of booting the system, control is handed to the BIOS, which passes it on to GRUB, which in turn runs the first line of kernel code. The sequence of execution is shown in Figure 4.1.1, with the blocks representing a dedicated piece of software (Renberg and Helin, 2015).



Figure X. The bootloading chain

As mentioned above, the first few lines of code are setting up an appropriate stack for the kernel and calling `kmain()` -- the kernel main function. This is done in assembly language as the higher-level language used in the project, C, cannot provide direct access to the processor's registers. At this point the kernel is in the processor's so-called Protected Mode, meaning it can access addresses up to the 4 GiB mark. It is now possible for the kernel to start initialising the structures required by the underlying hardware, mainly the processor and its memory management unit. The following figure, Figure 4.1.2, represents the function call sequence within the kernel's `main()` function, which consists only of initialisation functions. Each of the designated modules in the kernel does the job of initialising itself.

```
void __noreturn
kmain(void *mbd, uint32_t magic)
{
    gdtinit(); /* global descriptor table */
    idtinit(); /* interrupt descriptor table */
    vgainit(); /* video graphics array */
    kbinst(); /* keyboard */
    getbootinfo(mbd, magic); /* info about the system */
    painit(); /* page allocator */
    bainit(); /* buddy allocator */
    sti(); /* enabling interrupts */
    pginit(); /* paging */
    done();

    /* omitted code */
}
```

Figure 4.1.2. The i86 kernel's setup process

## 4.2 The Global Descriptor Table

Even though the GDT is initialised by GRUB, it contains only entries to keep the code running. The task of the kernel is to immediately update it with new entries that it is going to use. As discussed in the Design section, the x86 enforces segmentation upon the system while in Protected Mode. Segmentation and 32-bit addressing are inseparable. The way Intel does this is by having a global, as well as a local, table of segments called the Global Descriptor Table. In the i86 kernel, there are entries in this table that correspond to privileged-mode (kernel) and restricted-mode (user) segments. In the GDT the first entry must always map to zero, or null, as required by the design of the hardware. The second and third entry are marked as kernel code and kernel data segments respectively, capable of

addressing the full 4 GiB. The next two segments are user code and data segments, with the same addressing capabilities but they are marked to be with restricted privileges. No other entries are present in the GDT because the finished state of the kernel does not require additional ones. Figure 4.2.1 (Intel, 2015) displays the structure of a GDT entry.

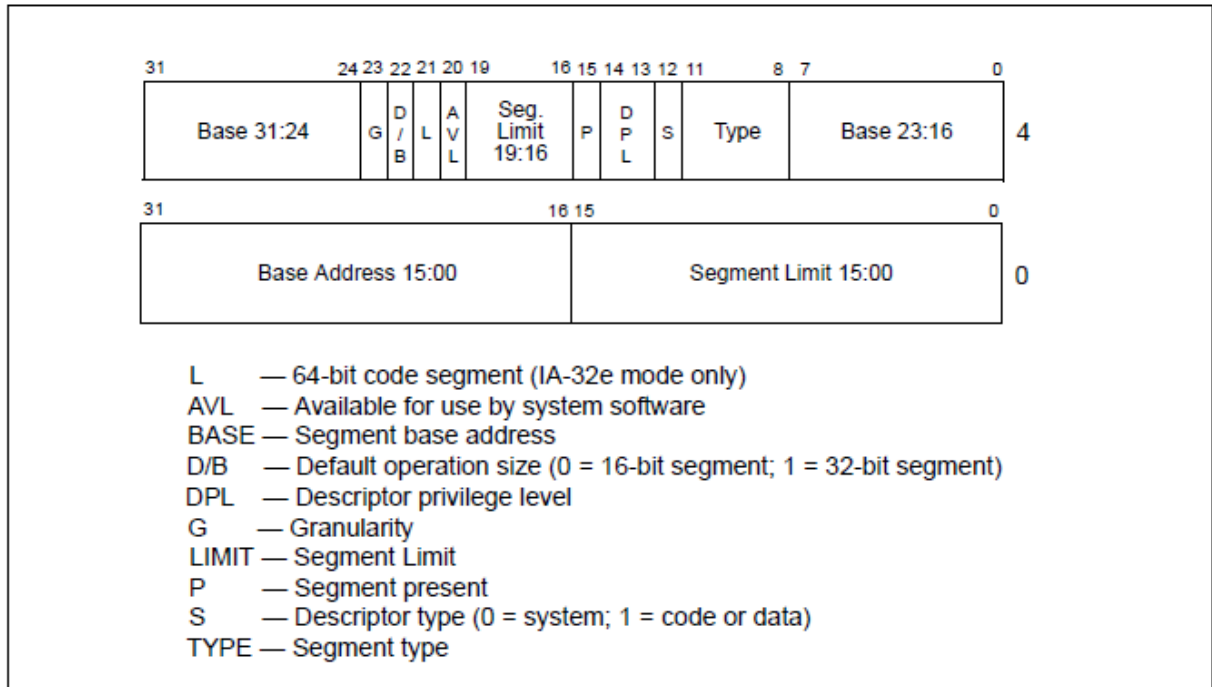


Figure 4.2.1. A Descriptor Table's segment selector structure

Within the kernel code, the GDT entries are represented as C structures and are set up through helper functions. The software representation of a GDT entry can be seen in Figure 4.2.2:

```

struct gdt_entry
{
    uint16_t limit_low;
    uint16_t base_low;
    uint8_t base_middle;
    uint8_t access;
    uint8_t granularity;
    uint8_t base_high;
} __attribute__((packed));
  
```

Figure 4.2.2: Representation of a GDT entry within the i86 kernel in C code

Individual bits are not defined; instead they are grouped logically and are represented by a larger data type (either a 8-bit value or a 16-bit one). After setting the descriptor table entries, the address at which they begin, as well as the table's size are passed to the processor's specialised register -- the Global Descriptor Table Register. Whenever the segmentation unit needs to translate an address, it accesses the address stored within the GDT register to search the entries of the GDT for an appropriate segment selector. The GDT register is loaded with the appropriate structure through a special instruction, which can only be accessed through assembly code. The kernel jumps to an assembly function that does this with a simple line of code: `lgdt gdt.`

However, the function does not end here. It also sets the first bit in the processor's control register, CR0, which is the Protection Enable bit. This ensures the processor operates in Protected Mode. After this a specialised jump instruction is executed that reloads the processor's code segment to the value of 8, which translates to the second entry of the global descriptor table once the segmentation unit starts operating. Finally, all the segment registers are loaded to contain the value of 16 -- the third entry into the Global Descriptor Table. As it becomes apparent, the code register points to the kernel's code segment descriptor in the GDT, and the remaining registers point to the kernel's data segment descriptor. With this the kernel finishes the process of updating the Global Descriptor Table and software continues in the 32-bit address space.

### **4.3 The Interrupt Descriptor Table**

The x86 provides hardware interrupts that basically do what the name implies -- they interrupt the current execution flow of the processor in order to raise awareness of an exceptional event that has occurred. Most of the interrupts are raised randomly and unpredictably by hardware devices, though they are not the only entities that can cause them. The broad term of interrupts encompasses three categories: exceptions, which are caused when the processor detects an error condition while executing an instruction; hardware interrupts, which signal the processor through hardware interrupt pins; and software interrupts, or traps, which are caused by executing the `int#` instruction. All of these interrupts correspond to an interrupt vector that ranges from 0 to 255. The i86 kernel does not implement software interrupts handling and focuses on the reserved exceptions from 0 to 31, as well as those from 32 to 47, which are re-mapped in this particular range. In x86 the Interrupt Descriptor Table (IDT) is similarly structured as the GDT in the sense that when an interrupt occurs a special register is accessed, the IDT register, which contains the address of the IDT. This IDT is then referenced to find the corresponding interrupt service routine stored within the table, which is mapped to a particular vector. Therefore, when an interrupt is issued, its assigned routine is called, which generally disables interrupts from further signalling the system, stores the registers used by the processor at the point of execution

before the interrupt, and stores the interrupt vector, along with an error code, if it is present for the vector. All of this implies that the routines are written in assembly language because they access the values of registers. The next step is to call a general-purpose handler, which is now written in C, to deal with advanced resolutions of the problem at hand.

Once the IDT has been set up and the IDT register has been loaded with the appropriate structure, the kernel must perform a finalising step before enabling interrupts -- it must configure the Priority Interrupt Controller (PIC). The PIC is a unit that manages hardware interrupts by accepting requests from peripheral devices, determining their priority and signalling the processor that a device wants to be acknowledged (Intersil, 1997). The initial hardware architecture was designed to work with a single PIC with eight interrupts but over time another PIC was chained onto the first one to accommodate more requests (now up to 15). Thus, the main PIC has the additional role of receiving and managing interrupts raised by the second PIC. The interrupt vectors and their corresponding devices are shown in Figure 4.3.1 (Renberg and Helin, 2015).

PIC 1	Hardware	PIC 2	Hardware
0	Timer	8	Real Time Clock
1	Keyboard	9	General I/O
2	PIC 2	10	General I/O
3	COM 2	11	General I/O
4	COM 1	12	General I/O
5	LPT 2	13	Coprocessor
6	Floppy disk	14	IDE Bus
7	LPT 1	15	IDE Bus

Figure 4.3.1. PIC1 and PIC2 interrupt vectors and their corresponding mapping

There problem with the PIC vectors is that they map directly over the first 16 processor exceptions. Luckily, the entries can be remapped so they can be serviced by a different interrupt vector. This is the final piece of interrupt setup that the kernel handles and at this point the C function *irqinit()* is called. This routine is illustrated in Figure 4.3.2, which makes use of an assembly function that outputs data on a specified port. Note that the defined values reside in a header file but are included in the figure to discern their values.

```

#define PIC1_PORT_A (0x20)
#define PIC1_PORT_B (0x21)
#define PIC2_PORT_A (0xA0)
#define PIC2_PORT_B (0xA1)

static void
irqremap(void)
{
    outb(PIC1_PORT_A, 0x11);
    outb(PIC2_PORT_A, 0x11);
    outb(PIC1_PORT_B, 0x20);
    /* the rest is omitted
       for the report */
}

```

Figure 4.3.2. The `irqremap()` routine that remaps the PICs' interrupt vectors through a series of commands

One function that starts appearing in the code is a language-specific library function -- *memset()*. Because the kernel is a stand-alone software bundle, this function is normally not provided to it, thus it must be coded from scratch, alongside other functionality, the details of which are mentioned in the next sub-section.

Once the PIC vectors have been remapped, the interrupt setup has been completed and the kernel issues an assembly-code instruction to enable interrupt reception on the processor. The PIC interrupt requests also have their own generalised handler provided by the kernel, which responds to the PIC vectors in the range 32-47. Similarly to the exception handler, it calls specialised handlers which are installed based on their corresponding vectors.

## 4.4 Standard C library functions

The C standard library is specified by the ANSI C standard and is specific to the C programming language. Its task is to provide functionality in terms of definitions, macros, routines and operating system services (ISO/IEC 9899:2011, 2011). Because the size of the library is huge and only a fraction of its services are needed by the i86 kernel, it does not make sense to have a full implementation of it. This is why the kernel implements only the functions and definitions it needs, in a designated section called *libc*. The functionality present by the kernel includes operations on memory (*memset*) and strings (*strcpy*, *strlen*) and defining some language-specific routines for identifying characters.

The modules are intended to be stand-alone so that they can be used both in kernel and user code.



## 4.5 Basic drivers -- video and keyboard

The logical step after initialising interrupt handling is to add support for communicating with peripheral devices. A necessary step in the creation of the kernel is detection of user input; the kernel must be made interactive. Additionally, the kernel must also provide some feedback as the communication with the user has to be two-way. The most basic form of output is displaying data on the screen through a hardware device called the framebuffer. The framebuffer is a memory-mapped input/output (I/O) device, meaning that it is directly mapped onto a portion of physical memory. Writing to an address or a segment of memory that corresponds to a memory-mapped device results in data directly being transferred to it. What this means for screen output is that any characters that are written in the region occupied by the framebuffer will appear directly onto the screen -- no need to communicate with the device through special commands and communication ports. The framebuffer is structured conceptually like a table of 80 rows and 25 columns and begins at physical address  $000B8000_{16}$ . Physically represented, it consists of 2000 consecutive entries but, as mentioned, every 80th entry counts as the end of a line on the screen, thus the next entry wraps around and moves onto the next line. Each framebuffer entry is 16-bits long, a short integer, and represents a character on the screen. The entry contains an ASCII character (ASCII is a character-encoding scheme), its foreground colour (the colour of the character itself) and its background colour. Writing on the screen is then simply a matter of deciding where to put the character, which is logically right after the previously written one, and storing its colour properties. Since the kernel strives to support a more friendly environment, it also ensures that the written screen *scrolls down*, meaning that any character that can be potentially written after the framebuffer's last column and last line causes the currently displayed text to be shifted upwards, freeing a new line for additional output. Writing text on the screen is an invaluable way to alert the user of anything that requires attention and allows the kernel to display diagnostic messages. This immensely helps with debugging the kernel itself because the contents of any entity, be it registers or memory locations, can simply be displayed to the user.

Of course, the task of writing to the screen, as simple as it may seem, is very tedious as it requires the programmer to store at most a character at a time. The logical thing to do is to create a wrapper function that does all of the printing in a familiar way and removes all the internal complexities. After all, it is fairly obvious that no user would like to supply a character of a sentence one at a time when the whole sentence string could be passed to a worker routine in a single pass. In the kernel, the function `kprintf()` (standing for kernel `printf()`) is supplied that works just like the familiar "print formatted string" function in the C programming language. Though not as detailed as the regular library function, it provides functionality for printing characters/letters, unsigned or signed numbers in base 10 and 16, and strings. Figure 4.5.1 shows an example piece of code that utilises `kprintf()`. The text

“RESOLVED WITH FRAME: 0x” is printed on the screen, along with the numerical representation of the address of *mapaddr* in base 16. The additional options here are that the value is treated as unsigned

```
kprintf("RESOLVED WITH FRAME: 0x%08ux\n", mapaddr);
```

Figure 4.5.1. Example usage of `kprintf()` within the i86 kernel

With the support of easily displaying text on the screen, it is necessary to acquire input from the user so that the kernel has a way of knowing what to do. Because the interrupt vectors for the PIC have been remapped and are fully functioning, the kernel installs a keyboard interrupt handler on vector 33. Whenever a key is pressed on the keyboard, a signal is generated and the PIC interrupts the processor. From there on the general interrupt handler transfers control to the specialised and newly installed keyboard interrupt handler. The handler then examines the interrupt scan code sent by the keyboard to determine what character this code corresponds to. It is essential to read the scan code because not acknowledging it effectively prevents the keyboard from raising additional interrupts. Once the character is extracted by mapping the scan code to an entry in a predefined character table, the handler decides what to do with it -- whether to print it out to the screen or to do something else in case it is a control character.

With the end of the initialisation of the two drivers mentioned above, the kernel operates with a fully functional video and keyboard handler that are the basis of communicating with the user.

## 4.6 Memory allocation schemes and paging

It is necessary for the kernel to provide basic necessities to the running software, one of which is memory. However, memory management is a rather difficult task because the requirements for memory vary between applications and it is impossible to determine them. Thus, the kernel is required to provide any amount of memory in the addressable range, otherwise it can be considered as unsupportive to the applications it accommodates. In other words, if the kernel cannot satisfy the requirements of its applications, it is basically worthless and fails to do its job properly. The big question is how the kernel manages to seemingly provide the full addressable 4 GiB of memory when the system might not have this much installed and the kernel itself occupies a part of that memory. Again, virtualisation is the answer and the concept is discussed in the next few paragraphs.

Because the kernel has to support the allocation of variable-sized memory, the underlying management has to cope with a trade-off between performance and space. As mentioned previously in the report, there is a tendency for memory algorithms to become faster at the

expense of requiring more memory to work with. The basic physical memory allocator of the i86 kernel is no different -- it operates as fast as possible but requires a rather hefty amount of physical memory to perform its job. However, this allocator provides significant benefits that outperform existing memory allocation implementations. With this better and more elegant approach, the i86 kernel distinguishes itself as a high-performance kernel.

The physical memory allocator, *palloc()*, which stands for page allocator, is a high-speed module that provides addresses to page-sized blocks of memory. As discussed later on, the size of a page on the x86 architecture is 4 KiB, meaning that *palloc()* can potentially allocate a little more than 1 million entries. This, however is practically impossible and the number of entries available is less than that for several reasons. One of the reasons is that there are regions of memory that are memory-mapped, or that are reserved for special functionality, thus they should not be supplied for general usage. The other reason is the kernel itself -- most of the kernel image is not expected to change as it is a static chunk of data. Therefore memory that the kernel occupies is never expected to be freed. The initialisation function of the allocator, *painit()*, deals with allocating the appropriate amount of memory that can be used for memory management. It does so by getting all the free regions from the end of the kernel image (presumably several megabytes into the address space) up to the last usable byte of physical memory. The region is then divided into allocatable blocks of 4 KiB each, which the allocator can then supply to memory requests. It should be noted that the speed of the setup for the allocator does not matter. Even if the kernel spends seconds in this portion of code just to initialise itself, it is insignificant just because the kernel might be running for days and the allocator will still perform at constant speed, regardless of the load on the system. The valid allocatable 4 KiB blocks are then put onto a stack, which is a data structure that provides great performance capabilities. Requests for a free address are met by just taking an entry from the top of the stack. Addresses that are returned are simply pushed on top of the stack. The processing requirements of this algorithm are miniscule and shall be compared with a buddy allocator in the Testing and Evaluation section. Figure 4.6.1 is an excerpt of the *palloc.c* source file, showing the simplicity of the approach.

```

void *
palloc(void)
{
    return (void *)page_stack[--stack_ptr];
}

void
pfree(void *address)
{
    page_stack[stack_ptr++] = (uint32_t)address;
}

```

Figure 4.6.1. The allocation and deallocation routines of the physical memory allocator

The servicing of 4 KiB blocks of memory is of immense help to the paging mechanisms employed in the kernel. This is noted in the following few paragraphs related to paging.

One obvious concern with this approach is the fixed amount of memory that can be allocated. Palloc() works exclusively with 4 KiB blocks, no more no less. Requests for smaller or larger amounts of memory cannot be met. However, this is resolved by allowing an additional allocator to service the variably-sized requests. This is not implemented in the i86 kernel but conceptually it is possible to create a *virtual* memory allocator that operates with virtual addresses only. Since it is *virtual* memory that is being allocated, any amount can be serviced as the memory is required only when it is being *accessed*. This is important to be understood, because the virtual allocator may lend a large amount of memory to a process. Once the addresses are accessed or used in some way, physical pages are *dynamically* brought into memory to map to the virtual addresses. Thus, an allocation may indicate that a large block of memory is serviced but the underlying mechanism does not actually allocate the memory until it's needed and when it does allocate it, it provides the memory in small blocks (because of palloc(), this would turn out to be 4 KiB).

Palloc does not provide support for contiguous physical address allocation (for memory larger than the page size, which is 4 KiB) or for addresses below a certain range. For historical reasons, these requests are for devices below a certain address range, typically below the 4 MiB mark. For such requests, another memory manager is provided -- *zalloc()*, or the zone allocator. The address range 0 to 4 MiB is reserved of usage by zalloc() once the kernel is in memory. Zalloc() also employs a mechanism of providing blocks that are stored in a stack-like structure, similarly to palloc(). However, zalloc() has a limited number of blocks per block size. It can service requests from 8 KiB to 64 KiB, in powers of two, and once the bank/stack of the particular size is exhausted, no more requests can be satisfied. Zalloc() is provided only for extremely rare cases where aligned memory is needed and the allocator is not expected to be used in more than a few cases. Thus, it does not need to

service large amounts of memory and is not concerned with running out of blocks to allocate. Figure 4.6.2 depicts the conceptual representation of `zalloc()`.

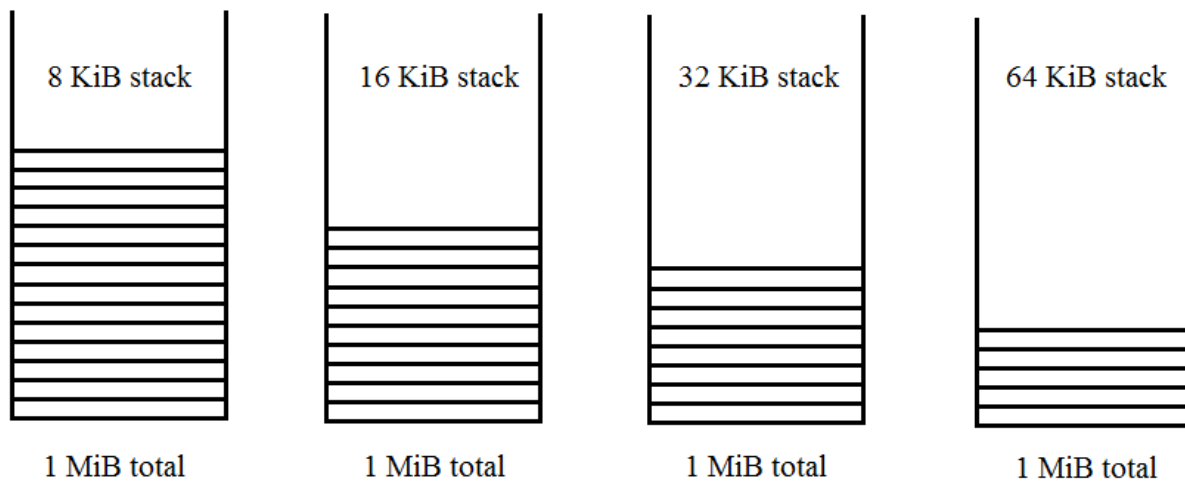


Figure 4.6.2. Conceptual view of the i86 zone allocator

All of the previously discussed allocators provide very efficient means of allocating *physical* memory. Because of the simplistic means of allocating this type of memory, there is no apparent protection mechanisms. Additionally, programs cannot address large amounts of memory and once all of it is allocated it is a dead end both for the kernel and for the applications themselves. In such a case the software chain is supposed to break or lock itself in a state where programs wait for memory requests that cannot be met as other parts of code keep it locked. The solution is to provide virtualisation mechanisms to seemingly allow every program to access the full addressable 4 GiB address space. This is achieved through paging, as discussed before.

The kernel's job is to simply perform the following steps: fill in page table and page directory entries in its own page directory, store the address of this directory in the CR3 processor register and set the Page Enable bit (bit 31) in the CR0 register. The first of the steps is done by first calculating the number of entries that need to be present in the page directory and the page tables. This is done so that the physical memory that the kernel occupies is mapped to the same virtual addresses -- a process called identity mapping. This ensures that the kernel is at a well-known location at any point of execution and allows user-mode applications to have restricted access to this area of memory for security reasons. Once the entries are populated within the tables, the start of the page directory is loaded into the CR3 register, the 31st bit of CR0 register is set and the kernel immediately sees the changes of operating with paging. At this point if any faults occur within the address mappings or translations, a page fault is caused by the processor.

Page faults can occur for a variety of reasons: it could be because there is a temporary inconsistency between the mappings that can be easily resolved by the kernel's fault handler, or the reason might be that the entry is just invalid for the current execution context. As mentioned above, the kernel's page allocator provides great performance improvements, one of which directly affects the page fault handling code. Interrupts are usually costly because they lock the system and as such they must be serviced as quickly as possible. Whenever the page fault handler requires a new page to be mapped into the page table directory, it requests an address from `palloc()`. The interaction between the two entities is as quick as it can get -- the handler receives an address in just a few machine instructions. The mapping is also performed quickly and the address that caused the page fault is executed again by the processor. With the speed of the `palloc()` allocator, this type of interrupt is serviced quickly and the allocator does not spend significant amounts of time running a "smart" algorithm for such basic memory needs.

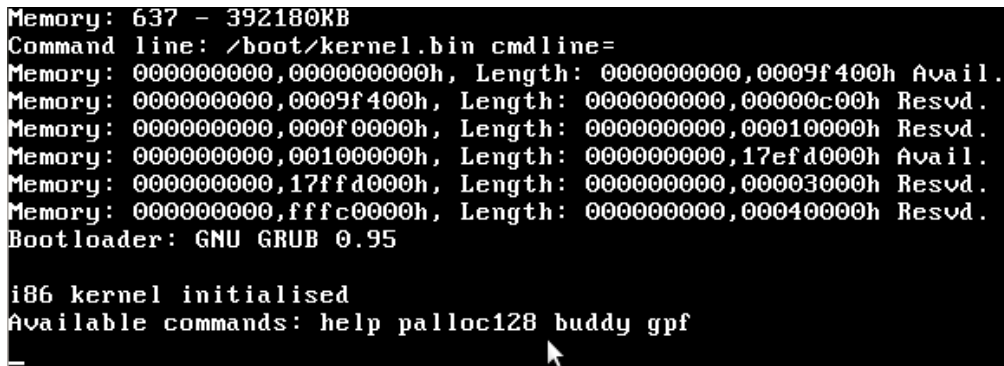
This lengthy, yet simple to implement memory management process is the basis of the i86 kernel. In fact, the core memory allocator is the strength and pride of the project, as it provides high-performance memory management routines to software.

This concludes the detailed development cycle of the i86 kernel. Just like any software, it is important to be tested, debugged, evaluated and possibly extended. These are issues that are discussed in the next few chapters.

## 5 System operation

This section aims to present the i86 kernel as seen by the user. The kernel itself is software that can hardly be visualised -- it is simply rough, low-level code that performs algorithms. It does just what it is told to do and it restricts user's options to what functionality it offers to them. The minimum implementation in the i86 kernel for interactivity consists of a keyboard handler and basic video output, as previously discussed in the Implementation section. This means that the simplest form of communication is to process commands entered through the keyboard and to display their result on the screen.

Upon booting up, the i86 kernel prints out relevant information to the programmer, such as the available and reserved regions of physical memory. After that it announces whether or not it has initialised all of the kernel's functionality and prints out the available commands to the user. Figure 5.1 shows these three seemingly simple operations.



```
Memory: 637 - 392180KB
Command line: /boot/kernel.bin cmdline=
Memory: 000000000,000000000h, Length: 000000000,0009f400h Avail.
Memory: 000000000,0009f400h, Length: 000000000,00000c00h Resvd.
Memory: 000000000,000f0000h, Length: 000000000,00010000h Resvd.
Memory: 000000000,00100000h, Length: 000000000,17efd000h Avail.
Memory: 000000000,17ffd000h, Length: 000000000,00003000h Resvd.
Memory: 000000000,fffc0000h, Length: 000000000,00040000h Resvd.
Bootloader: GNU GRUB 0.95

i86 kernel initialised
Available commands: help palloc128 buddy gpf
_
```

Figure 5.1. The i86 kernel screen once it boots and initialises itself

This sums up what the user sees from the kernel. The rest of the interaction consists basically of the user inputting commands and the kernel providing output in response. However, these programs are just tests, so they are discussed in the next section of the report.

## 6 Testing and evaluation

Throughout the system development process, the kernel has undergone very basic forms of debugging as there is no need to implement sophisticated and exhaustive tests at its current stage of the kernel. The usage of emulation software simplifies the testing process significantly and errors are easily recovered from. The approach taken for building the kernel is that of a linear process -- features are added to the kernel in small increments and are tested multiple times. Because of this any faults that occur and break the kernel are most likely due to freshly implemented features and are not caused by the current kernel code. This statement applies in all of the cases when the kernel has failed, thus no special cases are reported. The `kprintf()` function proved to be an invaluable tool when examining data that cannot directly be observed. This applies to a variety of situations, from spewing register contents and addresses onto the screen to validating data flow paths and verifying correct initialisation of kernel structures.

The debugging of the kernel therefore did not produce any significant findings and it just the matter of the author's unfamiliarity with coding in a low-level environment. The i86 kernel is in no way a complete and fault-proof because that is not its purpose -- it exists solely to follow the lifecycle of a development process; that is, it is created just to show how an efficient kernel can be built with modern techniques. A very important observation about the behaviour of the kernel must be made -- it trusts any software that runs its interfaces. This means that most of the edge cases that should be handled in regular software are omitted from kernel code. For instance, supplying a value in an invalid range to a function is generally not detected and it should not happen in any intentional way. What is more, any additional preventative measures that monitor data which is passed between routines creates run-time overhead -- it wastes precious processor cycles that could potentially service user code. As the kernel is expected to never work with random values and to be as ideal as possible, test cases are often not needed for any of the functions. Testing is performed strictly while developing new functionality and after that this functionality relies on correct input to be provided to it.

As discussed in the previous sections of the report, the i86 kernel is very efficient in managing physical memory. In fact, a quick analysis reveals that it is superior to well-known allocation algorithms, such as the buddy algorithm. To prove this, the kernel provides an implementation of a buddy allocator within itself, which, however, is unused and only serves as a demonstration. The `palloc()` allocator and the buddy allocator are compared side to side to show both their requirements and their performance.

The implementation of the page frame allocator functions has been shown in the Implementation section. However, this is only part of the code, as the allocator requires some work memory where it stores its entries. `Palloc()` maintains a static array of uninitialised data



to store all of its available addresses. Within the kernel, this is represented as follows (Figure 6.1):

```
static volatile uint32_t page_stack[PALLOC_STACK_SIZE];
```

Figure 6.1. Declaration of `pallocc`'s stack of free addresses

Because the kernel can theoretically access the whole of 4 GiB memory, the array is initialised to store a little more than 1 million entries (1024 page directory entries multiplied by 1024 page table entries). Each of the entries in the array contain the address of the start of a free block of physical memory. Of course, not all of this physical memory is addressable in practice, so the array may not be fully populated with entries, resulting in storage being wasted. The kernel allows the size of this array to be adjusted at compile time, but at run-time the requirements may be difficult to predict. Thus, it safely keeps the maximum possible number of addresses -- 1 million of them. Additionally, the `pallocc()` allocator requires 4 bytes of memory to store its stack counter, as represented by the *stack\_ptr* variable.

The run-time requirements for `pallocc()` are minimal. At most, it requires one function call, be it for allocation or deallocation, and each of the functions executes in just a few machine instructions. There are no complex calculations. The stack pointer is simply increased or decreased, depending on the request, and an entry, whose index is the stack pointer value, is referenced. Because the allocator is a very critical component of the system, it must check requests that are impossible to satisfy (like allocating a free page block when there are no more left). This condition, however, is not expected to happen at all, as there shall be components in the system that ensure the pool of addresses is never empty. Thus, this part of the code is optimised through branch prediction (improving the flow of execution in the instruction pipeline), because the condition must never happen. This is done in order to extract the last bits of performance from the routine.

On the other hand, the kernel implements a buddy allocator for comparison. One strong benefit of this allocator over `pallocc()` is its ability to allocate large blocks of contiguous physical memory but this is done at the cost of decreased performance. The implemented buddy algorithm requires a bitmap with each bit in it representing a 4 KiB block of memory, which totals to 128 KiB of static data required (much less than `pallocc`).

	<b>PALLOC</b>	<b>BUDDY</b>
<b>static data used</b>	1 MiB + 4 Bytes	128 KiB
<b>performance*</b>	O(1)	O(log n)
<b>processor cycles used</b>	~10	100+
<b>worst case load**</b>	1	up to 100
<b>block size</b>	2 <sup>12</sup> Bytes	2 <sup>12</sup> - 2 <sup>32</sup> Bytes

\* performance is expressed in Big O notation

\*\* worst case load is expressed in terms of the number of function calls performed

Figure 6.2. Palloc and Buddy performance characteristics

In the kernel, the supplied user programs for both of the allocators are *palloc128* and *buddy*. These programs are a sort of test on the system. *Palloc128* allocates 128 physical pages (512 KiB of memory), which creates a very interesting interaction with the paging system (effectively testing interrupt handling and page faults as well). Since at this point the kernel executes with virtual addresses, it receives a physical address from *palloc()*, which it treats as a virtual. The kernel does not have a mapping for this address, so the page fault handler invokes *palloc()* again to acquire a free page to map the newly allocated physical page. This is an intended feature and proves why a virtual memory allocator must be employed within the kernel. Because of this interaction, the program forces a waste of 1 MiB of memory. Figure 6.3 shows the series of page faults generated by *palloc128*.

The *buddy* program is not as exciting as *palloc128* -- it just shows that the buddy allocator correctly allocates 4 KiB blocks of memory -- the smallest block size it can allocate. This tests the buddy system at the edge cases, because the algorithm has to perform the most operations, searches and marks on the bits in order to determine if it can allocate a block of the requested size. Figure 6.4 shows successful output from the *buddy* program.

```

FAULTING ADDRESS: 0x014f9000
RESOLVED WITH FRAME: 0x014fa000
-----
EXCEPTION 14: Page Fault Exception, CAUSED BY: 0x00102225
MOVING TO DEDICATED HANDLER
FAULTING ADDRESS: 0x014fb000
RESOLVED WITH FRAME: 0x014fc000
-----
EXCEPTION 14: Page Fault Exception, CAUSED BY: 0x00102225
MOVING TO DEDICATED HANDLER
FAULTING ADDRESS: 0x014fd000
RESOLVED WITH FRAME: 0x014fe000
-----
EXCEPTION 14: Page Fault Exception, CAUSED BY: 0x00102225
MOVING TO DEDICATED HANDLER
FAULTING ADDRESS: 0x014ff000
RESOLVED WITH FRAME: 0x01500000
-----
Allocated and initialised 128 physical addresses
-

```

Figure 6.3. Program palloc128's operation

```

Memory: 637 - 392180KB
Command line: /boot/kernel.bin cmdline=
Memory: 000000000,000000000h, Length: 000000000,0009f400h Avail.
Memory: 000000000,0009f400h, Length: 000000000,00000c00h Resvd.
Memory: 000000000,000f0000h, Length: 000000000,00010000h Resvd.
Memory: 000000000,00100000h, Length: 000000000,17efd000h Avail.
Memory: 000000000,17ffd000h, Length: 000000000,00003000h Resvd.
Memory: 000000000,fffc0000h, Length: 000000000,00040000h Resvd.
Bootloader: GNU GRUB 0.95

i86 kernel initialised
Available commands: help palloc128 buddy gpf
buddy
Allocated address 0x02000000 with balloc()
-

```

Figure 6.4. Program buddy's operation

The last two programs perform very simple functions. The *help* program prints out the available system commands (currently the 4 programs), whereas *gpf* forces a General Protection Fault exception. Figure 6.5 shows *gpf* and its forced exception.

```
Memory: 637 - 392180KB
Command line: /boot/kernel.bin cmdline=
Memory: 000000000,000000000h, Length: 000000000,0009f400h Avail.
Memory: 000000000,0009f400h, Length: 000000000,00000c00h Resvd.
Memory: 000000000,000f0000h, Length: 000000000,00010000h Resvd.
Memory: 000000000,00100000h, Length: 000000000,17efd000h Avail.
Memory: 000000000,17ffd000h, Length: 000000000,00003000h Resvd.
Memory: 000000000,fffc0000h, Length: 000000000,00040000h Resvd.
Bootloader: GNU GRUB 0.95

i86 kernel initialised
Available commands: help pallocl28 buddy gpf
gfp
c
gpf
-----
EXCEPTION 13: General Protection Fault Exception, CAUSED BY: 0x00102033
NO HANDLER FOUND
SYSTEM HALTED
-
```

Figure 6.5. The *gpf* program in action

These basic programs test that the kernel is functioning as expected and can successfully satisfy requests by the user.

## 7 Conclusion

This concludes the analysis of the development of the i86 kernel. Computer architectures are discussed to provide an overview of what today's commodity hardware looks like, as well as how it works. These architectures define what can and cannot be done with software; they control the productivity of the software developer. It is discussed that such systems evolve constantly and sometimes the solutions that they provide raise new issues that need to be addressed. Such is the case with modern multi-core and multi-threaded programming. An additional aspect of kernel design is the employment of efficient memory management techniques. It is acknowledged that commonly-used algorithms that are persistent for years in operating system kernels are not the best approach and in most cases they are left unchanged due to software compatibility issues. In the area of memory management, the i86 kernel provides a novel and simplistic approach of quick allocation and deallocation of physical blocks. The performance improvements over other existing solutions are significantly large, and the kernel's allocator is compared with a general buddy allocator. While the project provides a concrete and working end-product, it does have areas for improvement. The components that deserve attention are as follows.

Firstly, the kernel's memory manager currently supports only allocations for physical memory. Even though this is a design feature so that the allocator is directly integrated with paging, processes do not have virtual memory management support. In the context of the kernel, paging accesses only virtual addresses and there should exist a virtual memory manager but this is not completed due to time constraints. Right now processes can write to any unmapped virtual address to trigger a page fault and load a physical frame that can be linked to the address.

An area which is not completed, although partially implemented, is process scheduling. There are numerous problems associated with scheduling and it is impossible to discuss them without separating them into their own section. It is sufficient to say that the plans for the i86 kernel are to provide a simplistic, priority-based scheduler with several queues of priority levels.

There are many ways in which the project helped in developing as a better programmer. One of the most important benefits gained is the ability to read large amounts of source code and discerning what it does, as it is the case of examining the Linux kernel. The process of kernel development is appreciated and understood from a very specific and detailed context, which applies directly to problem-solving skills. The world of low-level programming is much clearer now, with new styles of coding adopted and new bits of information acquired. Also, a great deal of algorithms have been appreciated, especially binary searching algorithms and sorting algorithms.

Overall, the project sparked a very strong interest not only in designing operating systems but also in creating low-level booting code, drivers and stand-alone software. It even strengthened the liking of the C programming language as a significant part of the code is implemented with new knowledge and understanding of it. Although it did not fully meet all of its requirements, it is definitely a project that is to be extended and worked on for enjoyment and self-improvement. All that can be said about the project is that it is very challenging but much more so enjoyable.

## References

- Bovet, Daniel P, and Marco Cesati. *Understanding The Linux Kernel*. Beijing: O'Reilly, 2001. Print.
- Christmann, Constantin, Erik Hebis, and Anette Weisbecker. 'Oversubscription Of Computational Resources On Multicore Desktop Systems'. *Lecture Notes in Computer Science* 7302.2012 (2012): 18-29. Web. 20 Mar. 2015.
- Drepper, Ulrich. *What Every Programmer Should Know About Memory*. 1st ed. 2007. Web. 2 Feb. 2015.
- Fiu.edu, 'CDA-4101 Lecture 5 Notes'. Web. 20 Mar. 2015.
- Gorman, Mel. *Understanding The Linux Virtual Memory Manager*. Upper Saddle River, NJ: Prentice Hall, 2004. Print.
- Hennessy, John L, David A Patterson, and Andrea C Arpaci-Dusseau. *Computer Architecture*. Amsterdam: Elsevier/Morgan Kaufmann Publishers, 2007. Print.
- Intel 64 And IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. 1st ed. 2015. Web. 29 Jan. 2015.
- Intel 64 And IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*. 1st ed. 2015. Web. 7 Feb. 2015.
- Intersil 82C59A CMOS Priority Interrupt Controller*. 1st ed. 1997. Web. 8 Mar. 2015.
- ISO/IEC 9899:2011*. 1st ed. 2015. Web. 16 Dec. 2014.
- Kang, Jonathan. 'Why Haven't CPU Clock Speeds Increased In The Last 5 Years? - Quora'. *Quora.com*. N.p., 2009. Web. 1 Mar. 2015.
- Lee, Edward. 1st ed. 2006. Web. 3 Mar. 2015.
- Liu, Yukun, Yong Yue, and Liwei Guo. *UNIX Operating System*. Beijing: Higher Education Press, 2011. Print.
- Renberg, Adam, and Erik Helin. 'The Little Book About OS Development'. *Little OS Book*. N.p., 2015. Web. 2 Jan. 2015.
- Rubini, Alessandro, and Jonathan Corbet. *Linux Device Drivers*. Sebastopol: O'Reilly & Associates, 2001. Print.

- Scott, Andrew. 'Andrew's Page: Teaching'. *Scott*. N.p., 2014. Web. 5 Nov. 2014.
- Silberschatz, Abraham, Peter B Galvin, and Greg Gagne. *Operating System Concepts Essentials*. Hoboken, NJ: John Wiley & Sons Inc, 2011. Print.
- Stallings, William. *Operating Systems*. Upper Saddle River, N.J.: Prentice Hall, 2001. Print.
- Stevens, W. Richard. *Advanced Programming In The UNIX Environment*. Reading, Mass.: Addison-Wesley Pub. Co., 1992. Print.
- Sutter, Herb. 'The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software'. *Gotw.ca*. N.p., 2005. Web. 20 Mar. 2015.
- Tanenbaum, Andrew S. *Modern Operating Systems*. Englewood Cliffs, N.J.: Prentice Hall, 1992. Print.
- Tennis, Caleb. *A Peek At Computer Electronics*. The Pragmatic Programmers, 2007. Print.
- Timmer, John. 'Are Processors Pushing Up Against The Limits Of Physics?'. *Ars Technica*. N.p., 2014. Web. 4 Mar. 2015.
- Zhang, Xiao, Sandhya Dwarkadas, and Kai Shen. *Towards Practical Page Coloring-Based Multi-Core Cache Management*. 1st ed. 2009. Web. 20 Mar. 2015.



# Appendix

## Appendix A: Commonly Used Terminology

*page*: a block of contiguous physical memory, typically 4096 bytes

*monolithic*: when referred to in the context of kernel design, a single, static, unchangeable block of code, often with modules that are laid out in a certain hierarchy

*microkernel*: a dynamic kernel design in which the kernel is built out of loadable modules

*cr0*, *cr3*: system registers specific to the x86 architecture; they control various processor properties

*static*, *volatile*, *const*: C code type specifiers, generally unrelated to the concept presented

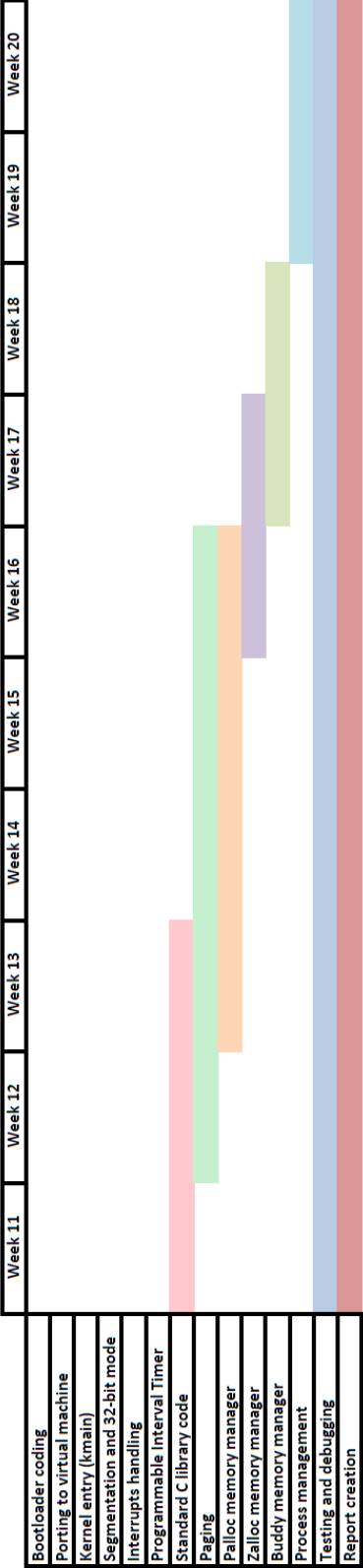
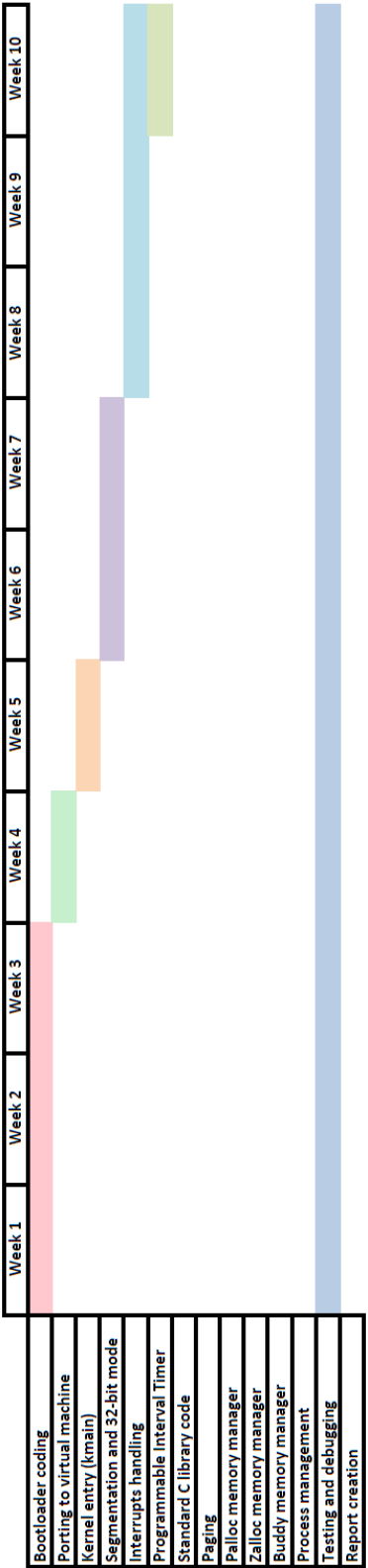
*uint8\_t*, *uint32\_t* and derivatives: variable type of a forced size; the *u* specifies that the value is treated as an unsigned number, the *8/16/32* represent the number of bits the variable has

*struct*: a C code type which groups one or more variables together

*hexadecimal*: base 16 notation, which consists of the digits *0* to *9* plus the letters *A* to *F* inclusive

*stack*: a data structure which adds or removes entries only at one end, called the top; can be visualised as a pile of elements

# Appendix B: Project Gantt Chart



## Appendix C: Project Proposal

# Creating an x86 kernel

Ivan Stanev

### Abstract

Operating systems adopt a variety of architectural designs in order to fully utilise a system's potential. Kernels sit at the core of an operating system and have an enormous responsibility to efficiently cover its users' needs. Common design choices include: the monolithic kernel, a layering scheme with components sitting conceptually on top of each other; the microkernel, which relies on multiple independent components communicating through special messages; and a mixture of both -- the hybrid approach. Often the edge cases prove insufficient for today's innovative technological progress but hybrid kernels also tend to become over-complicated and bloated in pursuit of supporting a variety of systems. This paper aims to outline an array of existing kernels and to show the process of building a hybrid kernel from scratch in a variety of phases. The final product is expected to display a thorough research and understanding in efficient modern kernel design.

### 1. Introduction

The design and implementation of an operating system creates huge software engineering issues. Many architectural styles have been adopted with avid supporters promoting their own models as the *de facto* standard for a successful and extensible kernel (Tanenbaum and Torvalds, 1992). This project focuses on a particular kernel implementation, from scratch, based on the x86-64 processor architecture (Figure 1.1) (Intel Corporation, 2014). The aims are to realise the author's views and knowledge on the topic, to cover only the main points on the available literature due to its sheer volume, and to analyse the strengths and weaknesses of prevalent architectures. The project strives to implement a customised approach with emphasis on the current technological advances.

The core of running and customising the functions of any computer is provided by the kernel and as such each kernel must be powerful and complex. Therefore the goal of the project is to create the basic building blocks from scratch: a bootloader to run the kernel, inter-process communication, interrupt handling, context-switching and memory management, as well as 64-bit platform support and multitasking.

Each of the dedicated sections describe the project's context and plans. The Background section provides a discussion and overview on operating systems, particularly on their kernel implementation, and the design decisions made by their authors. The Proposed Project section

describes the author's own approach in creating a hybrid kernel and utilising modern hardware. Programme of Work covers the breakdown of individual tasks and their spacing in time, including a Gantt chart for reference. An outline of the required material is noted in the final section Resources Required.

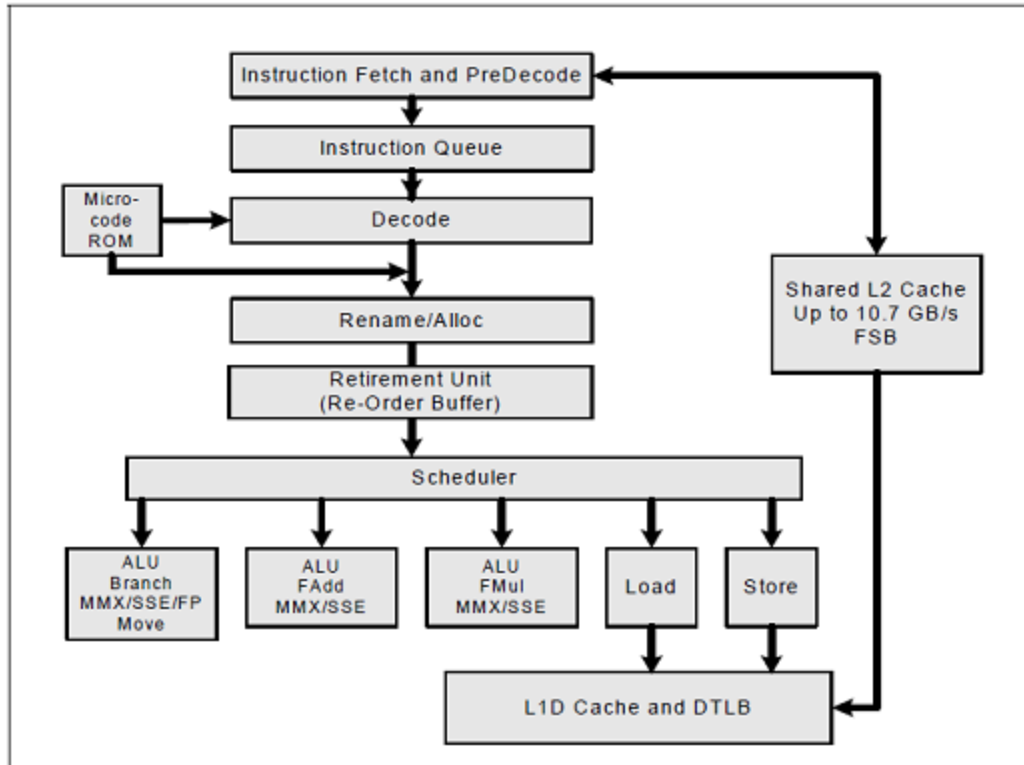


Figure 1.1 Intel Core processor architecture

## 2. Background

The kernel is the essential software structure that sits at the lowest layer (or inner ring, depending on architecture) of any operating system. It hides the differences in hardware architecture from upper layers. The kernel also provides a variety of core functions like scheduling and memory management. However, it must comply with the underlying system's instruction set architecture and considering the variety of hardware that is present nowadays, a kernel must be made portable between platforms. As with any software process, a particular design architecture is employed in the kernel structure, each with its own benefits and drawbacks.

Monolithic kernels encompass all functionality (memory management, file systems) in a single space, called the kernel space. This kernel space runs with full system privileges; in other words, it accesses all resources available. Another space, the user space, is completely separated from the kernel and accesses its functionality through system calls. Figure 2.1 shows the conceptual view of a monolithic kernel. Monolithic operating systems are Unix kernels, such as FreeBSD, SunOS and Solaris, and DOS kernels, such as MS-DOS.

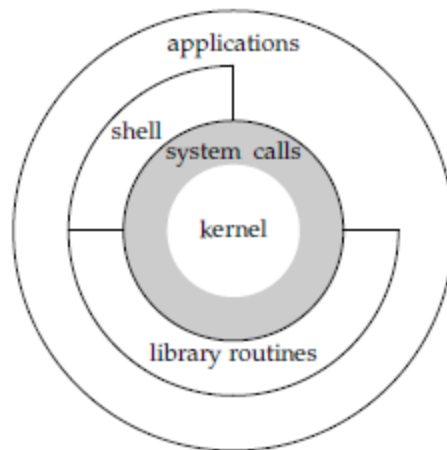


Figure 2.1 Monolithic architecture

Microkernels provide minimal functionality by handling the hardware devices and offering system calls to access their internal mechanism. Any other components of the system run in user space and communicate with the kernel through message passing facilities. Microkernels arguably exist to work around monolithic kernel limitations and to provide portable and extensible modules but they do not come without their drawbacks. Asynchronous message passing requires synchronisation mechanisms, which implies processing and ordering overhead on the system (Beck, 2002). Figure 2.2 displays the microkernel architecture. Famous microkernels include Mach, Minix, Chorus (Hulse and Earle, 2003). A notable feature of the Mach kernel, not present in Unix versions prior to its release, is multi-processing support (Silberschatz, Galvin and Gagne, 2011). However, pure microkernels are disputed to be commercially impractical because they are inefficient (Rusinovich, Solomon, Ionescu and Pietrek, 2009).

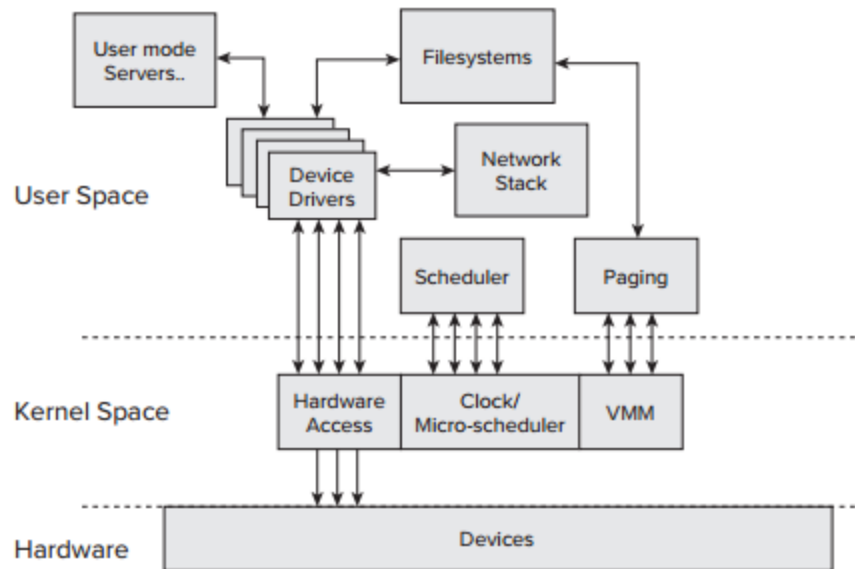


Figure 2.2 Microkernel architecture

As such extreme cases exist, there is always a middle ground approach and many commercial operating systems adopt a hybrid architecture. Windows NT (as seen on Figure 2.3) fragments the kernel space into different modules with distinct functions -- an executive layer containing process and thread management, virtual memory, input and output, etc.; a driver manager; a hardware abstraction layer; a kernel layer with interrupt handling and processor management. In this architecture, the kernel abstracts the processor and provides low-level synchronisation, whereas the executive layer contains the services running in a multithreaded environment with full memory support.

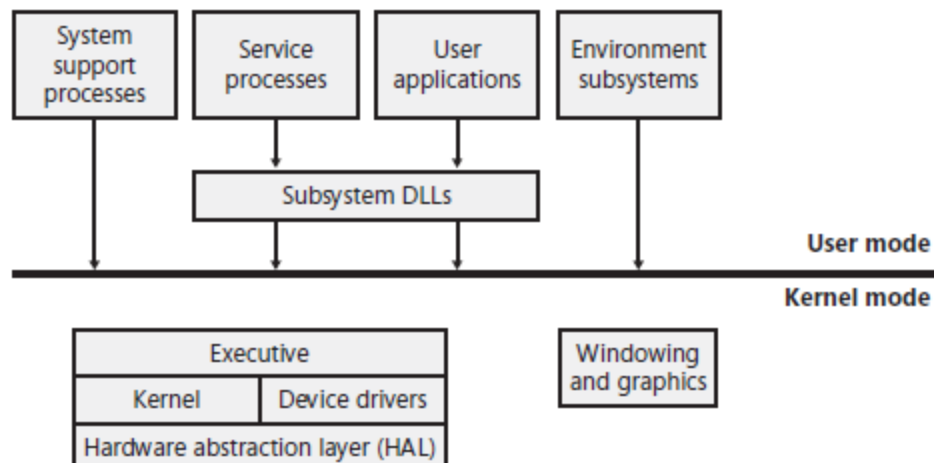


Figure 2.3 Simplified Windows NT architecture

The Linux operating system, once a monolithic system, now adopts an approach where all new components are strictly enforced into user space if not critically necessary to be in kernel space. Linus's operating system follows the microkernel design of modularity but uses function invocation instead of message passing. Linux is a Unix-like system that provides a Unix API and is POSIX compliant, but is not a direct successor of it; it only follows its original design principles and does not break standardised interfaces (Love, 2005). The Linux architecture can be seen on Figure 2.4.

The Mac OS X combines a mixture of a Mach-derived kernel and the Berkeley Software Distribution Unix kernel (see Figure 2.5). The Mac OS is a full-fledged Unix implementation, despite its hybrid approach (Levin, 2013). The kernel is named XNU and employs message passing facilities but often with pointers due to the common address space of the servers (Levin, 2013). Although the division of layers is conceptually strict, in reality many OS X services span the borders of multiple components (Levine, Halvorsen and Clarke, 2011).

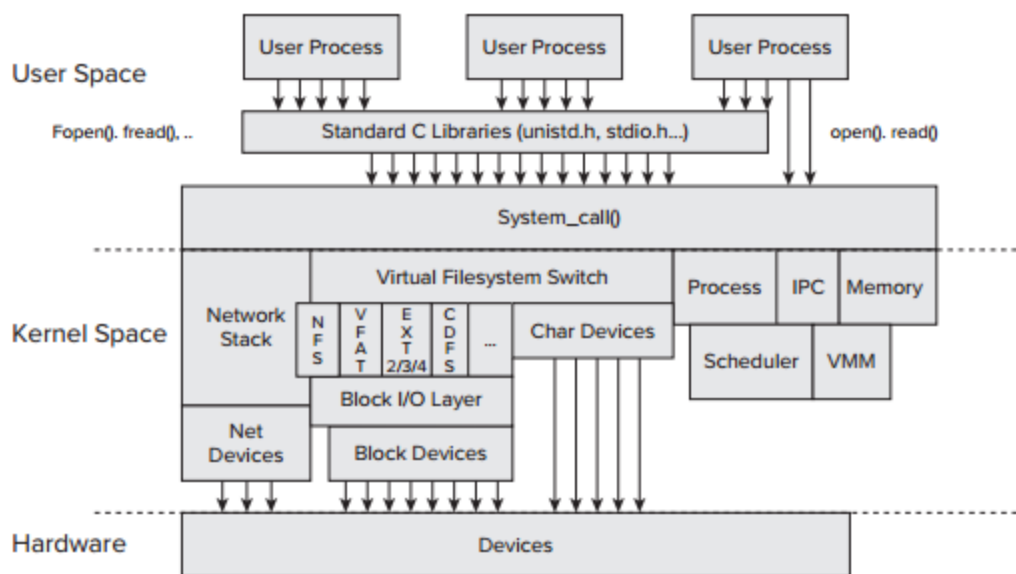


Figure 2.4 Linux architecture

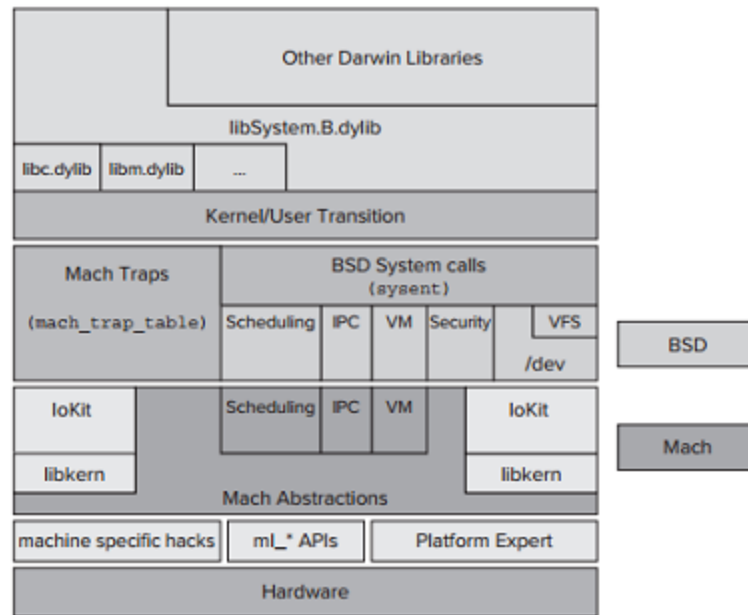


Figure 2.5 Mac OS X Darwin architecture

Notable non-profitable operating systems exist and have more of an educational value (Return Infinity, 2013) (Lionel, 2014) (Smith, 2013).

Exokernels are an interesting research topic as they tend to restrict the kernel only to resource protection so external applications manage resources by themselves (Engler, 1998). However, they are vaguely researched, rarely implemented and literature about them is scarce.

### 3. The Proposed Project

The proposed kernel aims to cover the best of everything available. It is a mixture of monolithic and microkernel architectures with modern software development techniques. It has a great emphasis on packing tightly functionally-related code and providing interfaces, as seen in object-oriented design and service-oriented architectures (Yegge, 2011). Although initially built for x86-64, the kernel is aimed at supporting different architectures, thus modularity is important. Simplification and minimalism are key concepts but not at the expense of security and reliability. Distributed operation and multiprocessor support is preferred but not at the cost of efficiency and speed. As with all projects, the creation process must be split into consecutive development stages. Note that further clarification on ambiguous components is presented in section five -- Further Discussion.

#### 3.1 The bootloader

The bootloader's main goal is to load the kernel. Depending on the size of the bootloader, it is split into two or more parts if necessary, with the initial bootloader loading the second one (a



process called chain-loading). The project's goal is to create a minimal bootloader within the Master Boot Record (MBR) of the device containing the operating system (flash drive or CD). Since the MBR has a strictly defined format and must be 512 bytes by convention (Carnegie Mellon University, 2007), it is kept to minimum functionality -- it runs the kernel's *main()* function. This is done by finding the kernel, loading it in memory and running it.

Address		Description	Size in bytes
Hex	Dec		
0000	0	<b>Code Area</b>	<b>≤ 446</b>
01B8	440	Optional disk signature	4
01BC	444	Usually null: 0x0000	2
01BE	446	<b>Table of primary partitions</b> (four 16-byte partition structures)	<b>64</b>
01FE	510	55h	<b>MBR signature: 0xAA55</b> <b>2</b>
01FF	511	AAh	
<b>MBR total size: 446 + 64 + 2 =</b>			<b>512</b>

Figure 3.1 Master Boot Record structure

### 3.2 main()

The *main()* function is the core of the kernel, which manages the inner workings. At this point information about the system is obtained through the BIOS functions available (i.e. amount of RAM) (IBM, 2006). If this piece of code is sufficiently small (440 bytes), it is pushed back to the bootloader code to preserve the logic.

### 3.3 Basic library functions

At this point an API is built based on standard C functions. The kernel aims to provide memory manipulation functions such as *memset()* and *memcpy()*. This API attempts to provide a custom interface to future application builders, similar to what Microsoft provides for its Windows operating system (Microsoft, 2014). The core of this API includes system functions and user-level functions with the concept of extensibility in mind.

### 3.4 Interrupts and Interrupt Handling

The kernel must handle hardware and software interrupts by responding accordingly to them. Thus, an interrupt descriptor table (IDT) is created to identify handlers for a variety of interrupt types. The first 32 interrupts are processor generated and necessarily require to be handled.

### **3.5 Memory Paging**

The page directory with page tables is created at a temporary fixed address. Once the kernel is sufficiently large, the directory gets positioned right after the kernel code. Page table size is 4 KiB with 1024 entries, with the page directory containing another 1024 4 KiB entries, totalling to a 4 MiB paging scheme. This is achieved through creating the paging scheme and enabling paging in the processor. Considering future extensions that support 64-bit code, the page tables are possible to be 2 MiB in size.

### **3.6 Timer Interrupts**

As with any preemptive environment, the on-chip timer is utilised to provide scheduling mechanisms. Therefore the programmable interrupt timer shall be interacted with.

### **3.7 Scheduling**

The utilised timer interrupts allow for scheduling. At this point the basis of a thread is conceptually outlined (i.e. what type of data is determined to be stored between context switching) and the mechanism of context switching is implemented with a specific underlying algorithm.

### **3.8 Memory Management**

No operating system is complete without correct memory handling and management. Heap allocation techniques are implemented at this stage and the power of the kernel is unleashed.

### **3.9 64-bit Support**

At this point attempts to convert the kernel into 64-bit are made. The kernel shall support full 64-bit application code.

## **4. Programme of Work**

The writing of such a large-scale project requires critical time management and it is highly important that deadlines are met and even, if possible, completed in advance. Conceptually, the timeline can be divided into a research stage and a development stage.

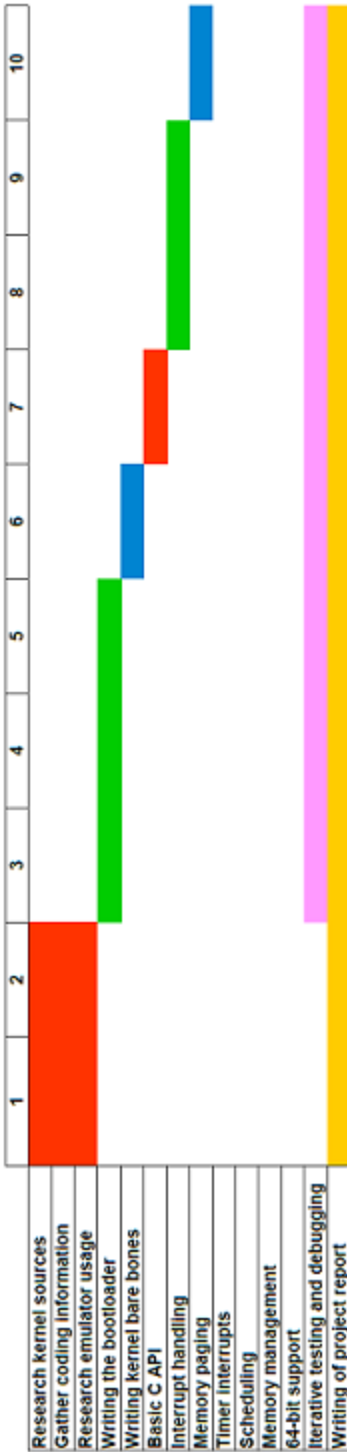


Figure 4.1 Gantt chart (1-10 weeks)

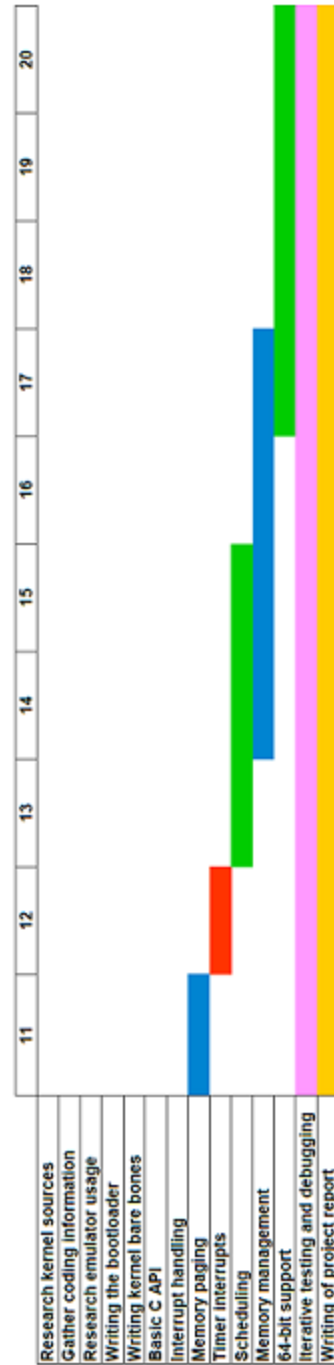


Figure 4.2 Gantt chart (11-20 weeks)

#### **4.1 The research stage (pre-Michaelmas Term and Weeks 1 to 2)**

The research stage is all about gathering the necessary resources for building the kernel. An investigation into existing kernel source code shall be made to analyse common systems' internals. Topics of research include: algorithms, emulator usage, design, coding practices, terminology, reverse engineering. Analysis of architectural approaches will be made to determine efficient or non-existing concepts. Research begins pre-university time to allow for sufficient time to analyse the available information.

#### **4.2 The development stage (Weeks 3 to 20)**

During this period the kernel's development begins. Figures 4.1 and 4.2 display the arrangement of different components throughout the two university terms. Debugging and testing is performed throughout the whole development phase, as well as the documentation of the project itself.

#### **4.3 The final product**

The final product expected is a kernel that is booted by its own bootloader and has the following features: a custom API with system and user functions; interrupt handling, including timer interrupts; memory paging and memory management for both the kernel and external applications. Additionally, the project strives to support 64-bit code, which is well-desired feature.

### **5. Resources Required**

The requirements for this project are multiple. This is why they are split into several sections. The availability of information is indefinite, therefore no issues are expected from the software and other requirements. As for the test machine, the author attempts to use his personal Packard Bell laptop with Intel Core i3 processor (x86-64) and 3GB physical memory.

#### **5.1 Hardware Requirements**

HR1. A test machine with Intel x86 processor architecture in order to run the kernel.

HR2. One or more development machines on which the kernel is written.

HR3. External storage (CD, flash drive) for porting the kernel from the development machines to the test machine.

#### **5.2 Software Requirements**

SR1. Emulation software (Bochs, VirtualBox) to quickly test the kernel in a virtual environment.

SR2. Cross-compiler, assembler and linker (GNU Binutils) to build the kernel for a particular architecture.

SR3. Development environment and text editors (Notepad++, Sublime Text) to write code for the kernel.

SR4. Debuggers (GNU Debugger) to find issues with written code.

SR5. Version-control system (Git, SVN) and/or cloud storage (Google Drive, Dropbox), which is essential when writing a large project that expect multiple edits.

### **5.3 Other Requirements**

OR1. Books on Operating Systems (by Tanenbaum, Silberschatz), hardware, software; tutorials and manuals (Intel Software Development Manual); articles and research papers (ACM).

## **6. References**

Beck, M. (2002). Linux kernel programming. 1st ed. London: Addison-Wesley.

Carnegie Mellon University, (2007). Writing a Bootloader from Scratch. [online] Available at: <http://www.cs.cmu.edu/~410-s07/p4/p4-boot.pdf> [Accessed 15 May. 2014].

Engler, D. (1998). The Exokernel Operating System Architecture. [online] Bar Ilan University. Available at: <http://u.cs.biu.ac.il/~wiseman/2os/microkernels/exokernel.pdf> [Accessed 22 May. 2014].

Hulse, D. and Earle, A. (2003). Trends in Operating System Design: Towards a Customisable Persistent Micro-Kernel. [online] Available at: [http://www.osdever.net./documents/persistent\\_microkernel.pdf](http://www.osdever.net./documents/persistent_microkernel.pdf) [Accessed 14 May. 2014].

IBM, (2006). Inside the Linux boot process. [online] Ibm.com. Available at: <http://www.ibm.com/developerworks/library/l-linuxboot/index.html> [Accessed 18 May. 2014].

Intel Corporation, (2014). Intel® 64 and IA-32 Architectures Software Developer's Manual. 1st ed. [ebook]  
[pp.http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf](http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf). Available at:  
<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf> [Accessed 13 May. 2014].

Levin, J. (2013). Mac OS X and iOS Internals. 1st ed. Indianapolis: Wiley.

Levine, J., Halvorsen, O. and Clarke, D. (2011). OS X and iOS Kernel Programming. 1st ed. New York: Apress L. P.

Lionel, (2014). Beryllium OS. [online] GitHub. Available at:  
<https://github.com/Lionel07/Beryllium> [Accessed 15 May. 2014].

Love, R. (2005). Linux kernel development. 1st ed. Indianapolis, Ind.: Novell Press.

Microsoft, (2014). MSDN Library. [online] Msdn.microsoft.com. Available at:  
<http://msdn.microsoft.com/en-us/library/ms123401.aspx> [Accessed 22 May. 2014].

Return Infinity, (2013). BareMetal x86\_64 OS. [online] Returninfinity.com. Available at:  
<http://www.returninfinity.com/baremetal.html> [Accessed 15 May. 2014].

Russinovich, M., Solomon, D., Ionescu, A. and Pietrek, M. (2009). Windows internals. 1st ed. Washington, DC: Microsoft.

Silberschatz, A., Galvin, P. and Gagne, G. (2011). Operating system concepts essentials. 1st ed. Hoboken, NJ: John Wiley & Sons Inc.

Smith, J. (2013). JS-OS. [online] GitHub. Available at:  
<https://github.com/JSmith-BitFlipper/JS-OS> [Accessed 16 May. 2014].

Stevens, W. (1992). Advanced programming in the UNIX environment. 1st ed. Reading, Mass.: Addison-Wesley Pub. Co.

Tanenbaum, and Torvalds, (1992). Tanenbaum-Torvalds Archived Debate. [online] Groups.google.com. Available at:  
<https://groups.google.com/forum/#!topic/comp.os.minix/wlhw16QWltf%5B1-25-false%5D> [Accessed 18 May. 2014].

Yegge, S. (2011). Steve's Google Platforms Rant. [online] Plus.google.com. Available at:  
<https://plus.google.com/+RipRowan/posts/eVeouesvaVX> [Accessed 14 May. 2014].