# A Multitasking Operating System

Michael Rochester
University of Sussex

Supervisor:
Martin Berger

November 5, 2014

# Contents

# List of Figures

# Introduction

We live in a world surrounded and controlled by machines, and software is our way of telling those machines how to act. As our machines and needs become more complex, so must the software that runs them. An Operating System is a software package that allows for others to easily create complex behaviours by providing simple high level abstractions over the intricate low level hardware.

The project goal is to understand the mechanisms and technologies involved in running an x86 computer and allowing for dynamic memory allocation, pseudo concurrent execution of code, and co-operative usage of the screen in user space applications.

The project has been split into a number of discrete 'goals' as listed below.

**Primary Objectives:**

- Create a bootloader to boot a kernel.
- Create a kernel that should be able to:
  - Load multiple 'programs' into memory.
  - Execute multiple 'programs' pseudo simultaneously using time sliced scheduling.
  - Allow multiple 'programs' asynchronous access to areas of the terminal screen.
  - Allow for the management of the execution state of each 'program'.
  - Allow each 'program' access to a unique memory space.
  - Allow 'programms' to asynchronously send data to each other.
  - Handle all common cpu interrupts and resolve them correctly.
  - Handle keyboard input and allow programs to 'collect' keyboard inputs.
- Create a small set of 'programs' with which to test the kernel.

**Extended Objectives:**

- Add some hardware drivers to load programs off disk and have some persistence.
- Move into pixel graphics mode.
- Use Multiple core's to have 'true' concurrency.
- Implement the entire c standard library.
- Port the 'Python' runtime to my OS (here I have chosen Python simply because there is a well documented 'todo' list for porting python to new OS's).

This report will outline the requirements for the project and the plan for creating an Operating System.

# Professional Considerations

As this project falls within the bound of Computer Science, It will comply with the British Society of Computing (BSC) Code of Conduct. Although there is no contractual obligation to follow this code, it is important to recognize and comply with them to ensure the integrity of the project.

Each of the four main sections of the Code of Conduct will be discussed in order;

## Public Interest

From the Code of Conduct: [1]

*You shall:*

- *have due regard for public health, privacy, security and wellbeing of others and the environment.*
- *have due regard for the legitimate rights of Third Parties*.*
- *conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement*
- *promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise.*

The project complies with this section of the Code of Conduct. No user information is gathered and the project poses no risk to a users health, privacy, or wellbeing.

The project makes use of third party material, however only within the licenses set out by those third parties that have been implicitly agreed to by all project members. Furthermore, it is explicitly marked where third party technologies were used.

## Professional Competence and Integrity

From the Code of Conduct: [1]

*You shall:*

- *only undertake to do work or provide a service that is within your professional competence.*
- *NOT claim any level of competence that you do not possess.*
- *develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.*
- *ensure that you have the knowledge and understanding of Legislation* and that you comply with such Legislation, in carrying out your professional responsibilities.*
- *respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work.*
- *avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction.*
- *reject and will not make any offer of bribery or unethical inducement.*

The members of this project each comply with this section of the Code of Conduct.

The level of competence of each member of the project was assessed and found fit to carryout this project tasks.

During the completion of the project each member will meet with a higher supervisor to ensure that they maintain professional competence and discuss viewpoints and criticisms they may not have considered.

## Duty to Relevant Authority

From the Code of Conduct: [1]

*You shall:*

- *carry out your professional responsibilities with due care and diligence in accordance with the Relevant Authoritys requirements whilst exercising your professional judgement at all times.*
- *seek to avoid any situation that may give rise to a conflict of interest between you and your Relevant Authority.*
- *accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision.*
- *NOT disclose or authorise to be disclosed, or use for personal gain or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation*
- *NOT misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others.*

The members of this project each comply with this section of the Code of Conduct.

A structure of meetings and other feedback channels have been established between the project members and their relevant supervisors to ensure each member is working within the bounds of their relevant authority.

## Duty to the Profession

From the Code of Conduct: [1]

*You shall:*

- *accept your personal duty to uphold the reputation of the profession and not take any action which could bring the profession into disrepute.*
- *seek to improve professional standards through participation in their development, use and enforcement.*
- *uphold the reputation and good standing of BCS, the Chartered Institute for IT.*
- *act with integrity and respect in your professional relationships with all members of BCS and with members of other professions with whom you work in a professional capacity.*
- *notify BCS if convicted of a criminal offence or upon becoming bankrupt or disqualified as a Company Director and in each case give details of the relevant jurisdiction.*
- *encourage and support fellow members in their professional development*

The members of this project each comply with this section of the Code of Conduct.

## Requirement for ethical review

This Project is a technical exercise in understanding the low level relationship between hardware and software, due to this, the project does not require human participation, testing, or feedback. As such, The Project does not require an ethical review, and there are no ethical concerns with this project.

# Understanding Operating Systems

Before this project can even begin, an understanding of what a Operating System is and does is required.

Operating Systems come in all different shapes and sizes, however usually they contain three distinct parts:

The kernel is the software that drives the hardware. It provides nice abstractions and interfaces to control the very fiddly underlying machine.

The Bootloader is the code that is run first. It gets things that the kernel might need set up, for example, getting the CPU into the correct operation mode.

The Final part is a collection of programs and other software that come with the Operating System. Although some people would argue this is not so much part of the Operating System, on some systems (namely Microsoft's Windows) a huge number of these programs are built right in, and seem from the outside to be part of the kernel.

### The Bootloader

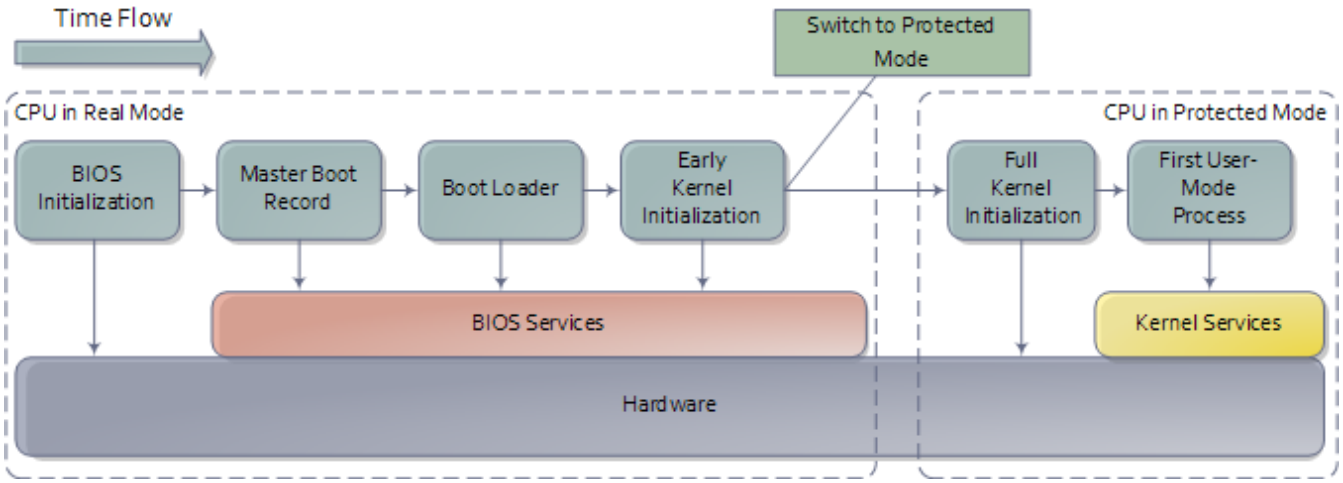



Figure 1: An Example of a boot cycle. The bootloader is in control while the CPU is in Real Mode.

The bootloader's job is to load the kernel. First off, the CPU has to begin executing code somewhere, most of the time this is done by the BIOS, which detects our bootloader and starts it running.

The majority of the rest of the bootloader's job is to get the hardware ready for the kernel to take over. Tasks like detecting the available physical memory of a machine is up to the bootloader, once it has gathered such information, the loader will store it somewhere the kernel can find it.

The final steps of the bootloader are to get the CPU into the right mode. It does this by first setting up some important 'tables' in memory. One of these tells the CPU what to do if an interrupt is produced. Another tells the CPU where it can find physical addresses in RAM when virtual addresses are looked up.

Finally the bootloader can jump to the start of the kernel and it's job is finished.

## The Kernel

Once the bootloader has passed control to the kernel, the kernel begins to prepare for executing programs. It will create tables of information, most of which will be empty to begin with, but they will quickly be filled. Some of the more important tasks are keeping track of which processes are currently executing and deciding which should go next, as well as ensuring that all processes think they have enough memory to operate correctly.

Once the kernel is happy that everything is in order, it will begin execution of the first process. For the rest of the time the computer is on, 'processes' will be run. These processes are pieces of code that perform some form of task, each having a unique environment. As far as a process is concerned, it owns the whole computer. However, as programmers are aware that each process does not own the whole computer, they can program complex interprocess interactions, with the kernels help of course.

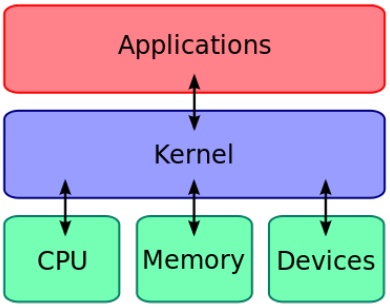



Figure 2: The Kernels Role is to make using the hardware easier for programs.

## The Built in Programs

After the kernel starts process 0, process 0 is free to spawn many other programs. Many of these programs will come 'built in' to an Operating System to create a richer platform for users.

Some would argue that these programs are not part of the Operating System its self. For example, in the Linux terminology, a kernel plus a collection of programs is referred to as a 'Distro' or distribution of Linux. In Windows however, many of these programs are well integrated into the kernel and are considered part of the Operating System.

Figure 3 shows how even showing simple windows on the screen requires many levels of different programs interacting with each other. This architecture is favored amongst some computer users as it is more flexible and customizable to their own needs.

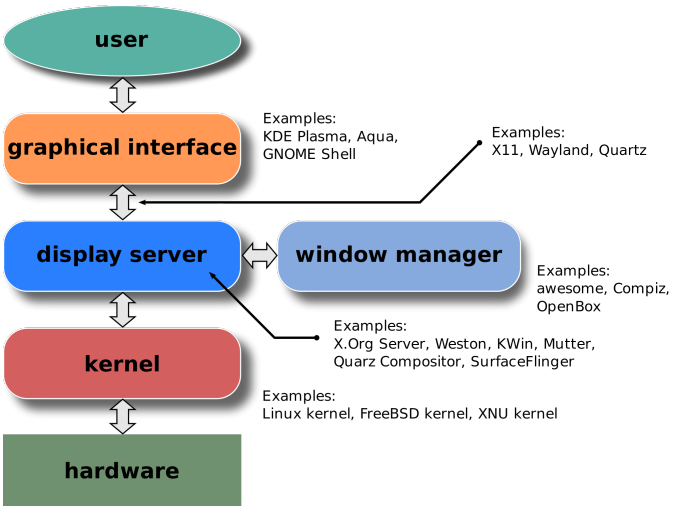



Figure 3: An example windowing server architecture.

# Requirements Analysis

In this chapter the Requirements Analysis will be discussed. The Requirements were decided upon by analyzing other modern Operating Systems and then designing the project Operating System to mimic the most useful and needed components. Therefore this chapter will first discuss the analysis of modern Operating Systems, and then the design of the project system.

The nature of an Operating Systems is such that there are two 'user groups'.

The 'End Users' are the people using the Operating System and its programs to assist in achieving an assortment of tasks. To them, the Operating System provides a few simple mechanisms to interact with the system, E.G. Launching programs, closing programs, and switching between the 'focused' program. Most of the inner workings of the Operating System are hidden from the end user, and they mainly interact with the 'User space programs', which are not defined by the Operating System Developer.

The 'Program Developers' use an Operating System as a platform to create the programs the 'End User' will use. To them, the Operating System is a Resource Manager and Scheduler and provides some crucial functionality for complex behaviours like creating new processes, Allocating dynamic memory, and allocating screen space to the programs. Many of the complexities of the Operating System (especially hardware control) are abstracted away from the Program Developers, however it is expected that they have some awareness of how the system is structured to successfully integrate with the kernel.

# Modern Operating Systems

In preparation of producing an Operating System, current generation systems were studied in detail. The features and interfaces of the systems were analysed to assist in the designing of the project Operating System. Although It was known in advanced that the project Operating System was not going to be complex enough to incorporate all features of the reference systems, it was important to study what current systems do, such that their most important features could be mimicked.

## The End User

The reference systems that were studied with the End Users in mind are: The Linux distribution remix 'Gnomebuntu' and the Microsoft Operating System 'Windows 8'. These systems were chosen to get a spread in our sample. The systems are both popular desktop systems, however: One, (Linux) is stemmed from an open source mentality, where the source code is freely available and the system is free, and the other (Windows) is produced by a company with the aim to provide paying business, and home customers with a usable system.

The parts of the reference systems that were analysed from the end users perspective are; program launching, process management, and program interaction as these are the main sections that will be implemented in the project Operating System.

### Launching programs in Linux and Windows

In both reference systems, the 'windows' key is used to expose the program launching interface.

In Windows, the interface is a customizable, horizontal-scrolling, tile-based launcher. Each tile can be set to launch an application, or even launch a sub-activity of certain applications. Furthermore each Tile can give basic information of the internal state of the programs, even when the program is not properly running. (see Figure 4.)

In Linux, the interface is a alphabetized list (with icons) of installed programs, sorted into pages. Pages can be navigated with vertical scrolling, or by jumping directly to a page represented by dots to the right of the screen. Linux also provides a 'quick launch bar' (on the left of the screen) that can be accessed by simple pushing your cursor to the left edge of the screen. This allows for quicker launching of your favorite programs (see Figure 5.)

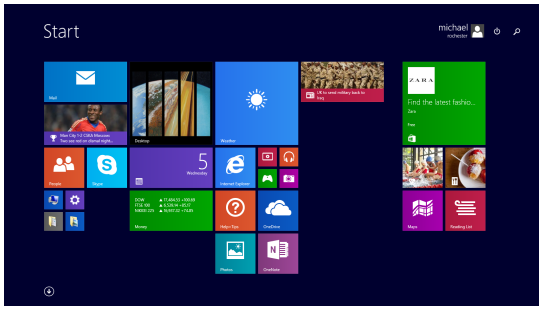



Figure 4: Windows Program Launcher Interface

Figure 5: Linux Program Launcher Interface

### Managing running programs in Linux and Windows

Both reference systems provide a way of interacting directly with the process manager to give the user power over which programs are running on their system.

In each system, the process manager is very similar, as the tasks they perform are identical. Both systems provide a window that lists processes and some relevant information about each process like PID, cpu usage, memory usage, disk usage, and network usage. It was found that the majority of these fields are concerned with what resources a process is using. This is the case because an Operating System's purpose is to manage the physical resources available to the software.

Each system also provides the ability to change what information is provided for each process, these extra informations include many other physical resource usages (I.E. Nice value, or cpu time) as well as a more in depth view of the default informations (I.E. separating 'memory usage' into: Virtual, Resident, Writeable, Shared, and Xserver memory.)

In each system this list was scrollable, allowing for many more processes that can fit in the window, furthermore the window becomes horizontally scrollable if the information exceeds the width of the screen.

Both systems provide a simple way of pausing or ending processes to manually free up resources being used by those processes.

Figures 6 and 7 show the process management screen in both the reference systems.

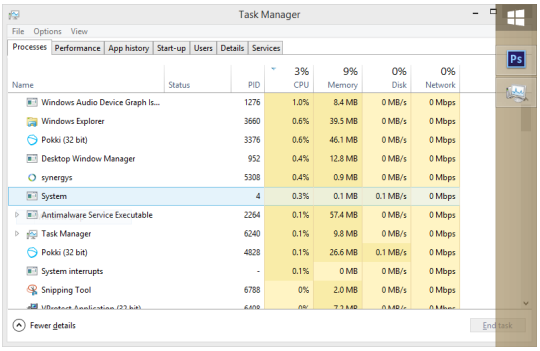



Figure 6: Windows Program Manager Interface

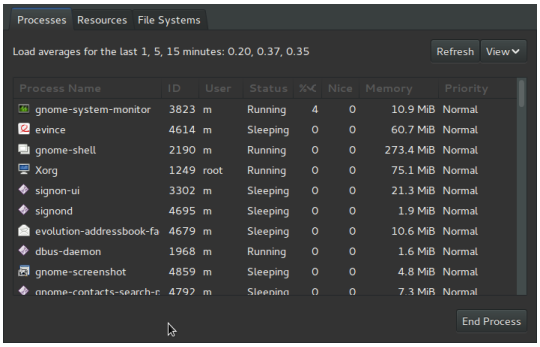



Figure 7: Linux Program Manager Interface

### Using programs in Linux and Windows

Interacting with programs is what users spend the majority of their time on a computer doing, because of this, it is a pivotal design aspect in modern Operating Systems. Some systems can be very different in how they allow you to interact with programs. Although the two reference systems were similar in how they present the programs to you, they differed when it came to navigating between programs.

In both the reference systems, the programs were presented in 'floating', overlapping windows. These windows share the screen space by having the 'active' window draw on top of all other windows. The system provides a way or moving, hiding, and resizing the windows to customize your workspace. Both systems provide similar mechanics for achieving this. They each provide a 'title bar' that can be grabbed and dragged around to move the window. Each also provide the ability to grab the very edge of any side of any window, or the corners, to resize the window to the desired size. Finally, each system provides buttons to manipulate the windows more directly. This is where the systems begin to differ.

The Windows Operating System provides three buttons; The first Button allows the user to minimize the window to hide it from the work space. The second button is to maximize/un-maximize the window, this makes the window occupy the entirety of the screen. The final button exits the application.

The Linux Operating System only provides an exit button.

Figures 8 and 9 give example of these features.

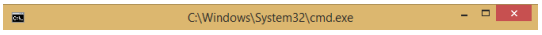



Figure 8: Windows Title Bar Example

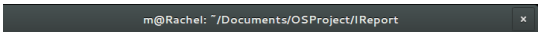



Figure 9: Linux Title Bar Example

Because of the nature of overlapping windows on both systems, it is possible for one window to be drawn entirely 'under' another window. In Windows it is in fact possible to choose to hide a window from the workspace completely. In this situation the systems both provide different ways to navigate back to the 'hidden' window.

On windows, There is a 'task bar'. This is a portion of the screen, (often located at the bottom of the screen), that stores small icons for each running program. Clicking on one of these icons will bring that program to the front, un-minimize it, and make it the 'active' program.

On Linux, Pressing the windows key, or pointing your mouse cursor to the corner of the screen shows you all of the open programs by 'zooming out' and separating all the programs from each other. Selecting a program from this 'exploded' view will un-minimize this window, bring it to the front, and make it the active window.

Figure 10 and 11 give example of these Navigation features.

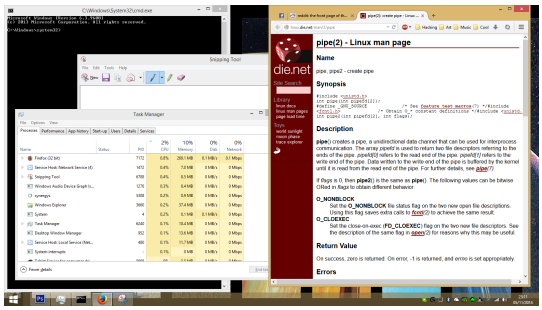



Figure 10: Windows Program Navigation Interface

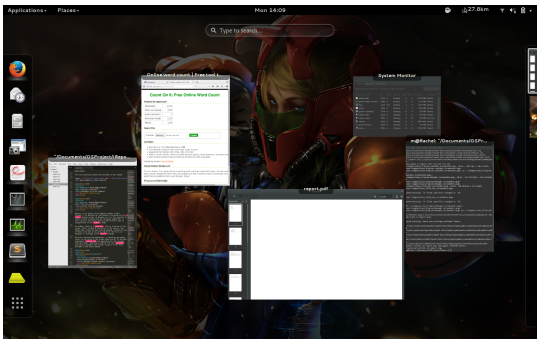


Figure 11: Linux Program Navigation Interface

## The Program Developers

To understand how program developers interact with Operating Systems The Linux distribution remix 'Gnomebuntu' was studied. Only one system here was studied as Linux is an open source system. This meant that the system could be looked at in more depth.

The parts of the Linux system that were analysed are the; Process management system, the memory management system, the windowing system, and Inter Process Communication.

### Managing processes in Linux

Linux exposes a number of system calls to the program developers to allow them to create, destroy, and pause processes, and also change the current 'executable' to allow for new programs to be run.

These features were considered independently of each other except for creating a new process and changing the current executable, as these calls are so commonly used together.

#### Creating a new process

To create a new process, Linux provide the 'fork' system call.

From the man page:

*fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.*

*On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.* [2]

This call can be used at any time to duplicate your process. The return value can be used to discriminate between the child and the parent, meaning that you can program different behaviours for each. However, for more complex behavior, the program developer may want the child process to become an entirely different executable.

To change the executable that your process is running Linux provides execve.

From the man page: *execve() executes the program pointed to by filename.*

*execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.* [3]

This call, as indicated, will completely replace a processes execution environment with one of a target executable, effectively switching to the new executable. It is used heavily with 'fork' to allow one program to spawn another.

#### Destroying processes

To destroy a process, Linux provides the 'kill' system call.

In Linux, kill is not technically just for destroying processes though.

From the man page: *The kill() system call can be used to send any signal to any process group or process.*[4]

However it CAN be used to kill processes, and as it turns out, this is the main use for this system call.

The caller provides kill with a PID and a signal. Different signals cause different behavior however the one that is relevant to the project is -9, which kills the process.

### Pausing and Restarting programs in Linux

To pause and continue programs in Linux, the kill system call is re used, but now with different parameters.

To pause a program, a call to 'kill -STOP <PID>' is made. This will prevent that program from being scheduled again.

To resume execution, a call to 'kill -CONT <PID>' is made.

### Managing memory in Linux

Linux provides two functions for memory management that were looked at together due to their close nit nature.

Malloc allocates the application memory and free deallocates it.

From the man page:

*The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized.*

*The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.* [5]

These two functions are almost always called in pairs, as it is common to want to explicitly free memory that has been allocated. However, all memory is freed if a program is terminated.

### Managing screen space in Linux

Managing screen space in Linux is a little different from managing other resources as the screen manager is a user space application, and technically speaking the screen cannot be 'consumed'. That is to say, if a program is using on part of the screen, that does not exclude anther program from thinking its occupying the same space. This means that the system can never 'run out' of screen space, as all windows can 'occupy' the same space.

As this is not how the project Operating System will be structured, there is not much we can take from the way Linux (or any other major modern Operating System) manages its screen.

### Inter Process Comunication in Linux

Communication between processes is an important part of creating complex behaviours in user space programs. Linux provides some simple calls to create chanels of communication between processes. The main call is 'pipe'.

From the man page:

*pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.* [6]

Here pipe is used to create a channel of communication. To use this channel, Linux provides 'write' and 'read':

From the man pages:

*write(pipe, data, size) writes up to size bytes from the buffer pointed data to the file referred to by openpipe.* [7]

*read(pipe, data, size) attempts to read up to size bytes from pipe into the buffer data.* [8]

These system calls will read and write using the pipe, allowing process to send data to each other. Note, it is by design that the pipe only transmits bytes, it is down to the user space program to define any communication protocalls for transmitting more complex data.

## Analysis conclusion

It was found that in end user design, many systems share similar design features. The most important of these feature should be mimicked in new systems to assist new users in understanding how the the system works.

From the programmers perspective. The system API's or modern Operating Systems have been well thought out and so new systems would do well to start from modern systems ideas and work from there.

# User Design

This section discuses the considerations of designing the system from the users perspective.

## The End User

The End Users interaction with The Operating System will be limited, as the majority of their time will be spent interacting with the user space programs running on the system. However the Operating System will provide a number of system level tools to the user.

The following tools will be provided.

- Process Creation.
- Process Management.
- Process Interaction.

These tools will be provided with three contexts:

### Process Creation Context

The process creation context will allow the user to view installed programs and run them from a command line interface.

The proposed design for this context is given by the following example screen.

```
 Installed Programs:
   1 | Notes Application
   2 | Pong Game
   3 | Name Application
   4 | Time Application

Please type an application number followed by a <return> to launch a program:
   >
```

This context will expand as more programs are installed, and may support scrolling or a second column of programs if the number of programs exceeds the size of the screen.

This context was influence by the designs of modern Operating Systems analysed earlier in this document. As the Operating System does not support pixel graphics or mouse pointers, no icons can be used and pointing and clicking on a program to launch it is impossible. This lead the design to be more refined and text based.

### Process Management Context

This context will allow the user to view currently running programs and manage them by pausing or stopping execution.

The proposed design for this context is given by the following example screen.

```
Running Programs:

  PID     | Name     | CPU Usage     | Memory Usage  | Status     |
  0       | init     |           0 % |          8 KB  | Paused     |
> 1       | time     |           2 % |          2 KB  | Running    |
  2       | notes    |           1 % |         25 KB  | Running    |
  5       | pong     |          80 % |        102 KB  | Running    |
  12      | name     |           0 % |          1 KB  | Paused     |

'>' Indicates the 'selected' program.
'f1' will pause the selected program.
'f2' will unpause the delected program.
'f3' will kill the selected program.
```

This context will allow the user to modify the state of programs. This can be useful if a program is using two many system resources or is not responding. It will expand into a scrollable area when the cursor gets to the bottom of the screen if there are too many applications.

This context was also influenced heavily by the analysed systems however once again with out mouse pointer support it was impossible to mimic it direct. The columns of information however were chosen from the analysed systems.

### Process Interaction Context

Here the user will use the programs that are running on the system. They can view the output of programs and send input to the programs using the keyboard.

The proposed design for this context is given by the following example screen.

```
Joe Bloggs                          12:24
***********004****-PONG-****014*********                Notes
*                                     *          |
*                                     *          | Shopping list
*  \#                00               *          | - Eggs 12
*  \#                00               *          | - milk 2 Pints
*  \#                                 *          | - bread (kingsmill)
*                                   \# *          |
*                                   \# *          |
*                                   \# *          |
****************************************
```

Here the context differers the most form the analysed systems as the project Operating System will not allow the overlapping of program areas.

## The Program Developers

As program developers are very familiar with how to create programs for other systems, the API for the project system was designed to be very simerlar to that of the Linux kernel that was analysed.

### Managing processes

The project Operating System will provide a very similar set of system calls to Linux, however their behavior may be simpler.

These features were considered independently of each other except for creating a new process and changing the current executable, as these calls are so commonly used together.

#### Creating a new process

Similar to Linux the process creation pattern the project Operating System was designed to follow the 'fork and exec' pattern. Therefore the project Operating System provides two system calls:

'fork()' will return the child pid to both the parent and the child, allowing for the processes to distinguish them selfs by comparing this value to the value of the 'get_pid()' system call. On failure it will return -1.

'exec(pid)' takes the integer value of the target program to be loaded. It does not return. On failure the current process will not continue.

#### Destroying processes

Unlike in Linux, where destroying, pausing, and restarting processes uses one system call, the project Operating System will use three separate calls.

'kill(pid)' will kill the target process, freeing all its memory and removing it from the scheduler.

#### Pausing and Restarting programs

Pausing programs is important for efficiency, therefore the project Operating System provides the following calls.

'pause()' and 'pause(pid)' will pause the target process until 'wake()' is called. With 'pause()' the current process will be paused and immediately revoke controls to the scheduler.

'wake(pid)' will wake the target process and allow it to be scheduled again. It will NOT cause the target process to be executed next.

### Managing memory

The memory managemnt API will be Identical to that of Linux. Therefore the project Operating System will provide two system calls:

'malloc(size)' will allocate a memory locations of size, and return a pointer to that location.

'free(pointer)' will deallocate the memory at a pointer.

### Managing screen space

As the Linux screen management system cannot be mimicked in the project Operating System, a new one entirely was invented. This lead to one system call:

'setio(px,py,wx,wy)' will attempt to allocate the portion of the screen using the starting coordinate (px,py) and the size (wx,wy). If this failed (that portion of the screen is already allocated) the call will block. Otherwise it will reserve this portion of the screen for this process.

### Inter Process Communication

The Interprocess Communication process will be very similar to Linux's system with one key addition.

'pipe(size, &readpipe, &writepipe)' will set up a communication channel of the requested size. On failure it will return one of a number of errors: -1 will indicate that the pipe cannot be created due to a lack of space, -2 will indicate that the system pipe table is full. This call will fill readpipe and writepipe with descriptors that can be passed to all the other pipe calls.

Once a pipe is set up three calls will be able to interact with it:

'write(pipe, byte)' will write the byte into the pipe. On failure it will return an error message; 1 will mean the pipe is full and 2 will mean the pipe is of the wrong type (a read-pipe not a write-pipe).

'read(pipe)' will return a positive byte representing the byte at the front of the pipe, or a negative error message; -1 will mean the pipe is empty and -2 will mean the pipe is of the wrong type.

'waitToRead(pipe)' will return a positive byte representing the byte at the front of the pipe or a negative error message; -2 will mean the pipe is of the wrong type. Note, if the pipe is empty, this call will block the process until the pipe is written to.

# Project Plan

This project constitutes a significant amount of work, some of which is of undefined difficulty (it will be impossible to predict ahead of time how much time it will take to complete). However there is also a significant amount of time to complete it, if time is managed properly and work effort is consistent.

Because of this, an FDD (feature driven development) style of project management was chosen.

This style is where a feature that can be completed (all prerequisite features are complete) is selected, analysed, designed and then implemented. Once it is completed and tested, the process is repeated.

This style keeps project engineers focused on similar tasks, and prevents them from bouncing around tasks left, right, and center.

The FDD style requires that project engineers have a set schedule and during scheduled work hours focus on the task at hand.

The downside of an FDD project is it is difficult to get an estimate of when the project will be completed. However this does not matter so much as FDD is more concerned about quality progress over superficial deadlines.

The Project engineer has proposed this schedule for the project work (Figure includes non-project work hours to demonstrate correct time allocations.):


Figure 12: Weekly Schedule. Project work shown in Blue, Extra project time shown in Purple.

# Log

### Week 1

- Created basic design docs.
- Rewrite existing code to be suitable for this project.

### Week 2

- Detect Upper Memory.
- Test Stack Jumping.
- Simple page frame allocator.
- Bound all interrupts to stubs.
- Created simple kernel threading.
- Created a 'printf' like utility.
- Supervisor Meeting 1: This week we discussed the state of the project, some goals to hit in the upcoming weeks and the proposal report due in next week.

### Week 3

- Implemented Paging.
- Moved to a higher half kernel.
- Loaded my first 'user space' program.
- Supervisor Meeting 2: This week we discussed how process creation was handled in linux and other Operating Systems, and how to move forward with concurrency.

### Week 4

- implemented fork system call.
- implemented exec system call.
- Supervisor Meeting 3: This week we discussed how IPC was handled on many different system and how I could think about going ahead with IPC.

### Week 5

- major bug fixing.

## Week 6

- created proposal for IPC.
- implemented IPC.
- began work on interim report.
- Supervisor Meeting 4: This week we discussed how to continue work on IPC and the next steps of using the IPC to abstract general process I/O.

## Week 7

- finished work on interim report.

# Appendices

The following Three pages are a copy of the Project Proposal document.

# Project Proposal

## Title: A Multitasking Operating system

**Student: Michael Rochester** **Supervisor: Martin Berger**

### Project Aims:

The purpose of this project is to produce a simple multitasking system in order to fully understand the working of OS's at the lowest level. I am interesting in this topic as an operating system sits at the heart of every computer. It defines the way that we as users interact with the system, and defines what kind of application we can run, this makes it one of the most important and central programs in computer science, and getting it right is not easy.

### *Primary Objectives:*

- Create a bootloader to boot a kernel.
- Create a kernel that should be able to:
    - Load multiple 'programs' into memory.
    - Execute multiple 'programs' pseudo simultaneously using time sliced scheduling.
    - Allow multiple 'programs' asynchronous access to areas of the terminal screen.
    - Allow for the management of the execution state of each 'program'
    - Allow each 'program' access to a unique memory space.
    - Allow 'programms' to asynchronously send data to each other.
    - Handle all common cpu interrupts and resolve them correctly.
    - Handle keyboard input and allow programs to 'collect' keyboard inputs
- Create a small set of 'programs' with which to test the kernel.

### *Extended Objectives:*

- Add some hardware drivers to load programs off disk and have some persistence.
- Move into pixel graphics mode.
- Use Multiple core's to have 'true' concurrency.
- Implement the entire c standard library.
- Port the 'Python' runtime to my OS (here I have chosen Python simply because there is a well documented 'todo' list for porting python to new OS's).

### Relevance.

Computer science is the study computers, how they work, and what we can use them for. Operating Systems are the core of any computer. They define the what and the how of computer functionality. They sit at the very heart of all computer sciences.

### Resources required:

- A computer for code development (already acquired)
- A i386-elf compiler for code compilation (already acquired)
- A i386 x86 emulator for code execution (already acquired)

## Optional resources:

- A Physical i386 cpu and related hardware for code execution (already begun attempting to acquire). Although it would be nice to boot my kernel directly on the hardware, It is not necessary for the completion of the project.

## Background Reading:

The C Programming Language (2nd edition) – Brian E. Kernighan & Dennis Ritchie.

This book provides a details description of the C programming language. As this is the language I have chosen to implement my OS in it is crucial I understand it thoroughly.

Modern Operating Systems (Jul 2013) – Andrew S. Tanenbaum.

This book describes key concepts and Algorithms behind operating systems, It will be a useful resource on understanding how current systems solve problems.

http://wiki.osdev.org/Main_Page (and all linked pages.).

This website provides detailed explanations of each component of an operating system from the perspective of someone implementing it. So far it has been an invaluable resource to understand how each component should behave.

## Log:

### *Week 1:*

Created basic design docs.
Rewrite existing code to be suitable for this project.

### *Week 2:*

Detect Upper Memory.
Test Stack Jumping.
Simple page frame allocator.
Bound all interrupts to stubs.
Created simple kernel threading.
Created a 'printf' like utility.
Supervisor Meeting 1:

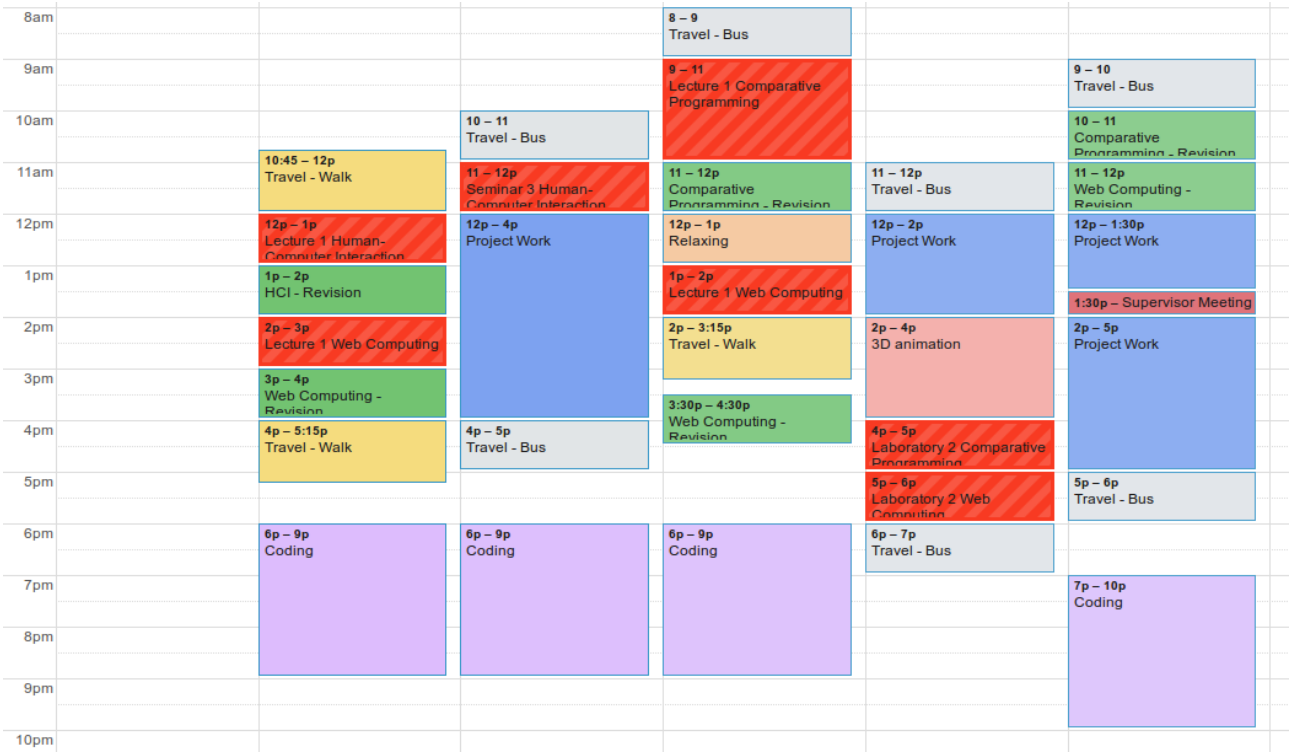
This week we discussed the state of the project, some goals to hit in the upcoming weeks and the proposal report due in next week.

### *Week 3:*

Implemented Paging.
Moved to a higher half kernel.
Loaded my first 'user space' program.

## Time table:

Follows is my current time table. My designated Project work time is in Blue, however time during the Purple 'Coding' blocks will also be used for project work where necessary.



# Bibliography


[1] Code of conduct — membership — bcs - the chartered institute for it. http://www.bcs.org/category/6030. Accessed: 2014-11-5.

[2] fork(2): create child process - linux man page. http://linux.die.net/man/2/fork. Accessed: 2014-11-3.

[3] execve(2): execute program - linux man page. http://linux.die.net/man/2/execve. Accessed: 2014-11-3.

[4] kill(2): send signal to process - linux man page. http://linux.die.net/man/2/kill. Accessed: 2014-11-4.

[5] malloc(3): allocate/free dynamic memory - linux man page. http://linux.die.net/man/3/malloc. Accessed: 2014-11-4.

[6] pipe(2): create pipe - linux man page. http://linux.die.net/man/2/pipe. Accessed: 2014-11-5.

[7] write(2): to file descriptor - linux man page. http://linux.die.net/man/2/write. Accessed: 2014-11-5.

[8] read(2): read from file descriptor - linux man page. http://linux.die.net/man/2/read. Accessed: 2014-11-5.