MOS : A Multitasking Operating System

Michael Rochester
Candidate: 23344
University of Sussex

Supervisor:
Martin Berger

April 26, 2015

Contents

List of Figures

Chapter 1

Introduction

The overall aim for this project is to produce a simple Multitasking Operating System (herein called MOS), following 5 key tenants, to study structure, allocation, and management of different classes of resource. This section provides a brief introduction to Operating Systems, Discusses the project aims, and looks at the approach take.

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.

1.1 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.

1.1.1 The Bootloader

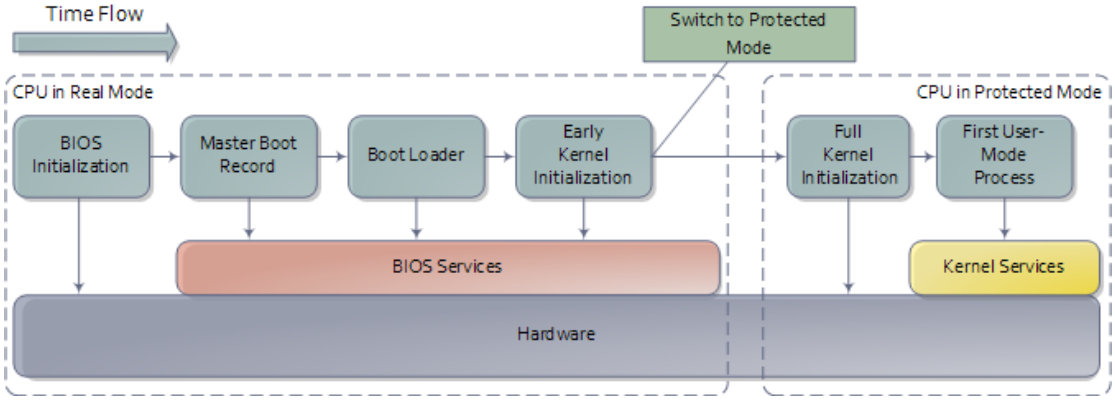


Figure 1.1: An Example of a boot cycle. The bootloader is in control while the CPU is in Real Mode.

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.

1.1.2 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.

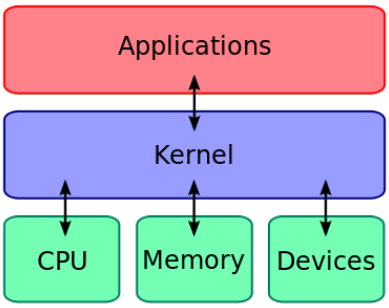


Figure 1.2: The Kernels Role is to make using the hardware easier for programs.

1.1.3 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.

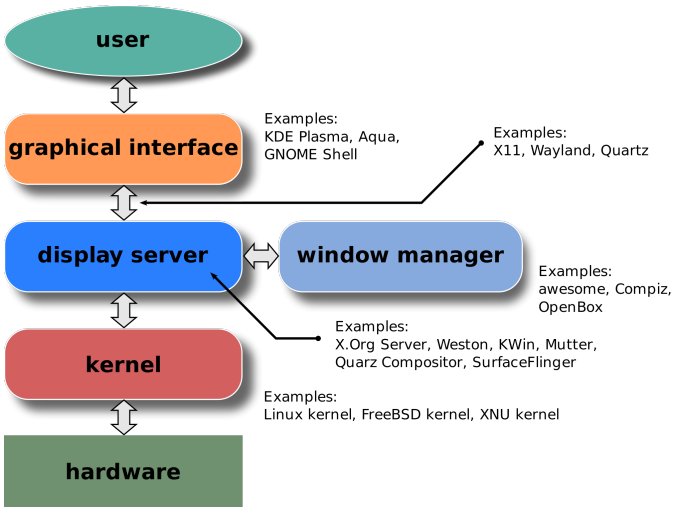


Figure 1.3: An example windowing server architecture.

Figure 1.3 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.

1.2 Project Aims

The project was begun with a number of Primary and Extended Objectives, however as the project progressed, new objectives were discovered. These new objectives were prioritised over other extended objectives. This lead to some of the original extended objectives being left incomplete.

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:

- [Complete]Add some hardware drivers to load programs off disk and have some persistence.
- [Partially Complete]Implement the entire c standard library.
- Move into pixel graphics mode.
- Use Multiple core's to have 'true' concurrency.
- 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).

New Objectives:

- [Complete]Create A user manual.
- [Complete]Create A Developers manual.
- [Complete]Create set of scripts for getting the source code, Building the project, and deploying it to a computer.

Chapter 2

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 [1]. 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;

2.1 Public Interest

For this project no user information is gathered and the project poses no risk to a users health, privacy, or well-being.

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.

2.2 Professional Competence and Integrity

The project was created as an undergraduate Final Year Project for a computer science course. It pulls knowledge from many of the modules taught as part of this course.

During the completion of the project The project engineer, Michael Rochester, will meet with a supervisor to ensure that he maintains professional competence and considers ideas and viewpoints he may not have considered.

2.3 Duty to Relevant Authority

A structure of meetings, and other feedback channels have been established between the project engineer, Michael Rochester, and the supervisors to ensure he is working in accordance with the University of Sussex guidelines.

2.4 Duty to the Profession

The project engineer, Michael Rochester, will carry out this project to the highest standard possible to maintain the reputation of the profession, furthermore all materials produced as part of this project will be freely available to any member of the profession to help further the understanding of the project domain.

2.5 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.

Chapter 3

Requirements Analysis

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 MOS. Although It was known in advanced that MOS 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.

Modern Operating Systems are 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, allocating screen space to the programs, and accesing the hard drive. 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.

3.1 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 MOS.

3.1.1 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 3.1.)

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 3.2.)

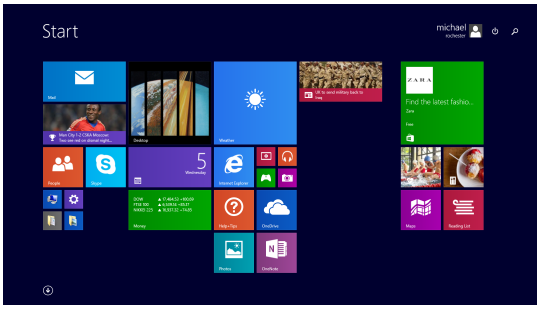


Figure 3.1: Windows Program Launcher Interface

Figure 3.2: Linux Program Launcher Interface

3.1.2 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 3.3 and 3.4 show the process management screen in both the reference systems.

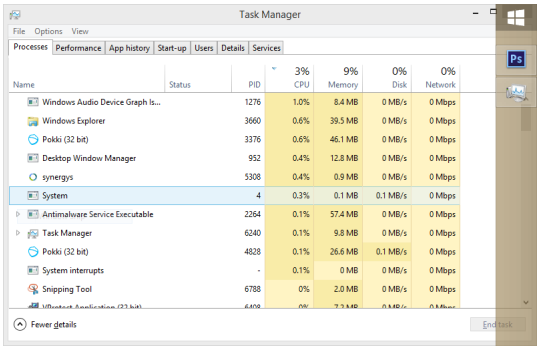


Figure 3.3: Windows Program Manager Interface

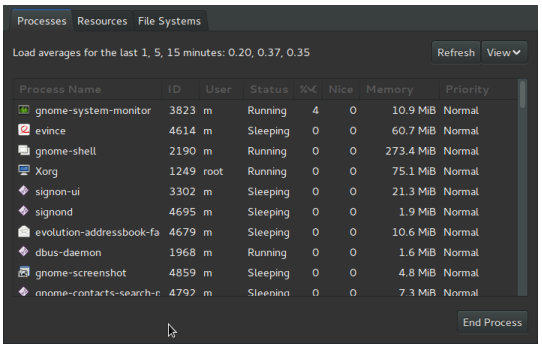


Figure 3.4: Linux Program Manager Interface

3.1.3 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/unmaximize 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 4.1 and 3.6 give example of these features.

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

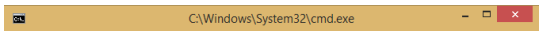


Figure 3.5: Windows Title Bar Example

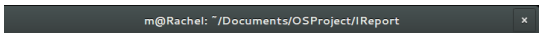


Figure 3.6: Linux Title Bar Example

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 3.7 and 3.8 give example of these Navigation features.

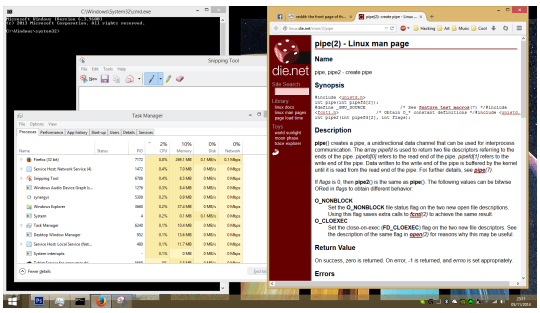


Figure 3.7: Windows Program Navigation Interface

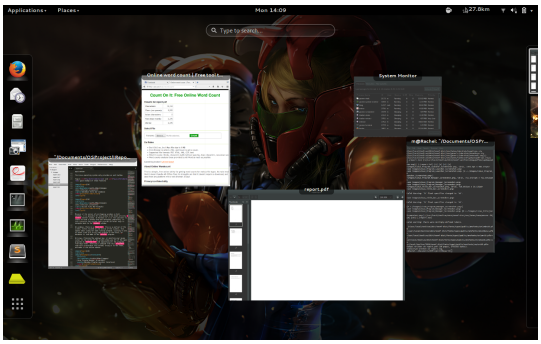


Figure 3.8: Linux Program Navigation Interface

3.2 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.

3.2.1 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.

Creating a new process

To create a new process, Linux provide the 'fork' system call. [2]

Fork will spawn a new processes with a perfect copy of the calling processes memory and cpu state. This effectively means that fork will return twice, once into the caller and once into the child. It will return 0 to the child and the child pid to the parent.

Changing executable

To change the executable that your process is running Linux provides execve. [3]

Execve will replace the calling processes memory with that of the target executable and then jump to the __start label in the target executables code. This allows for switching into new executables and can be used along side fork to spawn new programs.

Destroying processes

To destroy a process, Linux provides the 'kill' system call. [4]

In Linux, kill is not just for killing processes. It allows the caller to send a signal to the target process. Frequently the signal sent is SIGKILL or SIGTERM which both destroy the target processes in different ways.

Pausing and Restarting processes

To pause and continue processes in Linux, the kill system call is re used, but now the caller sends the SIGSTOP signal.

3.2.2 Managing memory in Linux

Linux provides two functions for memory management, one to allocate memory and one to free it.

Allocating memory

Linux provides the malloc call to allocate memory. [5]

Malloc is passed the size of memory need in bytes and will return a pointer to the heap that is at least of the size requested.

Deallocate memory

Linux provides the free call to deallocate memory.

Free is passed a pointer to memory that was received from malloc, it frees this memory allowing for it to be allocated during future calls to malloc.

3.2.3 Managing screen space in Linux

The screen space is not managed by the Linux kernel and so is not useful as a study for MOS.

3.2.4 Inter Process Communication in Linux

Communication between processes is an important part of creating complex behaviours in user space programs. Linux provides some simple calls to create channels of communication between processes.

Creating a pipe.

Linux provides the 'pipe' system call to create pipes. [6]

'pipe' created a one directional communication channel and provides the user a handle to this channel.

Reading and Writing to a pipe.

To use this channel, Linux provides 'write' and 'read': [7] [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 protocols for transmitting more complex data.

3.3 Analysis conclusion

It was found that in end user design, many systems share similar design features. These features are very rich and operate do to a wide collection of tool, utilities, and applications The most important of these feature should be mimicked in new systems to assist new users in understanding how the the system works, However due to time constraints MOS will target much simpler versions of these features.

From the programmers perspective. The system API's or modern Operating Systems have been well thought out and have stood the test of time, however sometimes represent behaviours far more complex than that which MOS targets. The studied API's will be mimicked both to follow tried and tested ideas, and also so that program developers feel at home developing for MOS.

Chapter 4

Design

As with the Analysis, the Design phase was separated into End User and Program Developer sections. However here the structure of the kernel itself was also considered and planned. The designs presented here are the final designs for the system, some earlier versions can be found in XXXXX.

Design for the end user focused on similar tools, looks, and controls to existing systems; The Linux CLI was referenced for a lot of the simple command line tools as MOS' text mode graphics share a common interface to Linux terminals. MOS programs also targeted a clean workspace, avoiding unnecessary ascii drawing and focussing on delivering the service provided in a quick and clean manner.

Program Developer design was constructed with a 'many simple tools' structure. Each system call the standard library contained was made to perform one task, rather than bunching many tasks into one more complex call. This allows developers to maintain simple, intuitive control over their resources.

4.1 The End User

The End Users interaction directly with MOS will be limited, as the majority of their time will be spent interacting with the user space programs running on the system. Because of this, MOS will ship with a number of user space tools for managing the system

The following actions will be possible using the built in tools.

- Process Creation.
- Process Management.
- File/directory Creation.
- File/directory Management.
- File editing.

4.1.1 Process Creations : MASH

MASH will be MOS' built in shell. Using MASH, users can launch any other process.

MASH will support passing parameters to launched processes, and a persistent file history. It will be displayed to the user as a single line at the bottom of the screen, with a *promptwheretheusercanentertheircommand*.

Since MASH will cover Process Creation. All other actions that user can perform with build in tools are achieved through different built in programs. The user can use MASH to launch these programs passing any desired parameters.

4.1.2 Process Management : processes

The program called processes will serve as the users tool for killing processes.

The proposed design for this program 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     |

Press k to kill the current [>] process.
```

As shown, this tool will show the user information about each running processes and afford them the ability to kill these processes.

4.1.3 File/Directory creation: mkdir and touch

MOS will provide two programs for creating files and directories. touch will create empty files and mkdir will create directories. Both programs will operate in a similar way, taking an absolute path as their only parameter. Neither will create anything unless the parent directory of the wanted object exists, and the parameter is well formed (with a training / for directories).

4.1.4 File/Directory Management: rm and mv

For file and directory management MOS will provide two functions.

The first is rm. rm will try to delete the object at the obsolete path path passed as its only parameter. This path must be will formed and the object must exist however rm does not mind if the object is a file or directory and will delete either. If a directory is selected, all child nodes of that directory will be delete recursively.

The second is mv. mv will take two parameters, the first will be the path to an existing file system object (file or directory) and the second will be the path to an existing directory. mv will move the specified object (and all of its children in the case of a directory) into the target directory.

4.1.5 File Editing: notes

MOS' provides a simple multi-line text editor for editing text files. The program takes one parameter which is a path to the file you wish to edit. If the file does not exist notes will create it.

Notes supports common cursor key commands for navigating the cursor and saves when ctrl+s is presses.

4.2 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 similar to that of the Linux kernel that was analysed.

4.2.1 Managing processes

MOS will provide a very similar set of system calls to Linux, however their behaviour 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 MOS was designed to follow the 'fork and exec' pattern. Therefore MOS provides two system calls:

'fork()' will return the child pid to both the parent and the child, allowing for the processes to distinguish themselves 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, MOS 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 MOS 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.

4.2.2 Managing memory

The memory managemnt API will be Identical to that of Linux. Therefore MOS 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.

4.2.3 Managing screen space

As the Linux screen management system cannot be mimicked in the MOS, 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.

4.2.4 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.

4.3 The Kernel

The kernel was split into a number of discrete modules, each performing one simple task, or over seeing the access to one peice of hardware. These modules were originally designed to be self contained, not relying on any other modules, however it was quickly realised that some modules could be brought 'closer together' to help streamline the kernel. The most notable case of this is the scheduler and the terminal manger, the scheduler contains the process entries for each running application, including the io descriptors, these io descriptors are required by the terminal to operate, therefore the scheduler allows write access to these data structures 'cross module'.

The Kernel Loader can also be seen as a kernel module however it is special in that it does its job once at boot, and then does nothing for the remainder of the computers uptime.

The kernel design was a living structure throughout the project as it was hard to predict the exact nature of each module. The following modules will be implemented to support all suggested features:

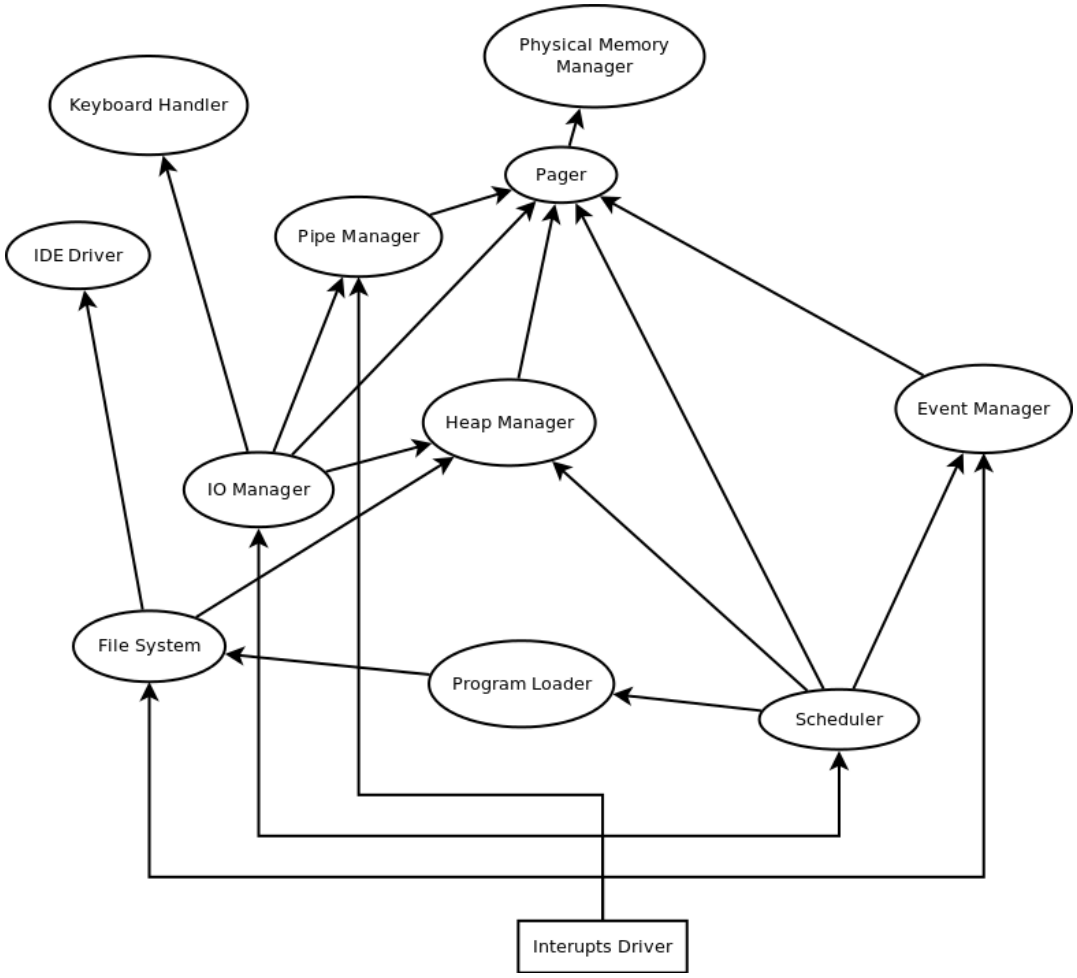


Figure 4.1: Kernel Module Dependencies

4.3.1 The Kernel Loader

As this module is different from the others we will discuss it first.

The Kernel Loader can actual be seen as more of a kernel in its own right. It is the code that is loaded by our Boot loader (GRUB). The loaders job is to to coreectly configure the hardware in the CPU's Real Mode and then switch over to Protected Mode for the rest of the uptime of the CPU.

It has its own smaller Physical Memory Manager, IO Manager and XXX, To assist in its task. It then maps the kernel to the correct location in memory and jumps to the kernels entry point.

Since the kernel only executes in Protected mode, it has less direct access to physical memory, therefore the kernel loader must initialise the memory map for the Physical Memory Manager module and map this memory map into a suitable Virtual memory location.

4.3.2 The Scheduler

The schedulers task will be to maintain and update all relevant information about all running processes. This means it is responsible for creating, destroying, and switching processes. It will also provide write access to parts of the process information to other modules where necessary.

The scheduler depends on the Program Loader to load new programs from the hard disk, The Event manager organise timed events such as sleeping a processes, The Pager to provide a persistent memory space, and The Heap manager to provide memory space for passing parameters to new processes.

4.3.3 The Program Loader

The program loaders job is to prepare a Program for execution; Loading the binary into memory in an appropriate manner, setting all process table fields to correct values, and ensuring all other system modules correctly reflect the execution of a new program.

The program loader depends on the File System to load programs off disk.

4.3.4 The Event Manager

The event managers job is to track timed events. It will allow for the registration of certain types of events with an event deadline. Once the deadline is passed the Event manager will perform the requested action. This module requires that its main function 'events()' be called with some frequency.

The event manager depends on the Pager to provide a persistent memory space.

4.3.5 The Pager

The pager is the core module that provides memory to the system, it controls the CPU's page tables, mapping physical memory to virtual memory. The pager is also responsible for giving out unique sections of memory to all other kernel modules. By assigning this job to the pager, the kernel memory management is centralised, making it easier to move sections of memory around during development.

The pager depends on the following Physical Memory Manager for locating unallocated physical memory.

4.3.6 The Physical Memory Manager

The physical memory manager maintains a map of the physical memory (RAM) and can provide new physical memory pages to the pager.

The Physical Memory Manager interacts directly with hardware and so has no dependencies.

4.3.7 The Heap Manager

The heap manager allows other kernel modules to use the well known malloc and free calls, providing them with small, temporary (more temporary than the memory provided by the pager) sections of memory of any size.

The heap manager depends on the Pager to provide a persistent memory space.

4.3.8 The File System

The file system will create a file/directory tree abstraction over the IDE drives present on the system. It will provide two different mechanisms for accessing its tree; one for the kernel and one for the user space programs.

The kernel functions will be to read entire files/directories into memory, or write sections of memory into files. These functions are designed for speed and elegance and the kernel always wants to load/save entire files to disk at once.

The user space abstractions are centred around giving out handles to files, and letting programs read those programs like character pipes. These abstractions focus on requiring a significantly lower memory footprint, and building input and output piece by piece.

The file system depends on the IDE Driver for reading and writing to the Hard Disks.

4.3.9 The IDE Driver

The IDE Driver will be responsible for performing reads and writes to the Hard Drives. It will also need to contain some levels of caching as hard drives are typically very slow compared to the rest of the system.

The IDE driver interacts directly with hardware and so has no dependencies.

4.3.10 Pipe Manager

The pipe manager will maintain information on any pipes that user space programs require. These pipes will also be utilised and the main IO mechanism for user space programs.

The pipe manager depends on the Pager to provide a persistent memory space.

4.3.11 The IO Manager

The IO manager will control all IO for user interaction with processes, as well as any outputting the kernel may do. It must maintain a current rendering of all running processes screen space, and deliver keyboard input to user space processes at the users command. It should also maintain different kernel space and user space windows, allowing for the user to switch between the kernel and user process view.

The IO Manager depends on the Pipe Manager for delivering and receiving IO to the user space processes, the keyboard driver for collecting key presses, The Pager to provide a persistent memory space, and the heap manager for allocating window space to processes.

4.3.12 Keyboard Driver

The keyboard Driver handles keyboard input to the system, Its main job is to correctly pass signals along to the IO manager such that they can be sent to user space processes, however it may also detect system commands (like ctrl+tab and F1) and call the respective module to perform the expected action.

The Keyboard driver interacts directly with hardware and so has no dependencies.

4.3.13 The Interrupt Handler

The interrupt handler is the module that will correctly catch and process all interrupts, this may be seen as the controller of the kernel to some extent as it is both how the kernel receives information from hardware (E.G. when a key is presses, or a hard drive finishes an operation) and the entry point from the user space into the kernel (all system calls occur through the interrupt handles.)

Because of this it depends on many of the previous modules to provide services to user space, deliver information around the kernel, and control the state of the system. Namely, The File System, IO Manager, Pipe Manager, Scheduler, Keyboard Driver, and Event Manager.

Chapter 5

Project Implementation

This section discusses the implementation of each kernel module in turn along with implementation choices and reasoning for each decision.

What wont be discussed here is implementation of the various user space programs delivered with MOS, this is because they are out of the scope of the project and although they were required to make MOS a functional system, they are not interesting from the perspective of this project.

Furthermore the user space standard library will not be discussed. This section of code is a simple user space wrapper of system calls. The system calls them self are the interesting parts, and these will be discussed with the module that contains them.

We will discuss the more complex and central modules firs, and then move onto the simpler modules after.

5.1 The Scheduler

5.1.1 Data Structure

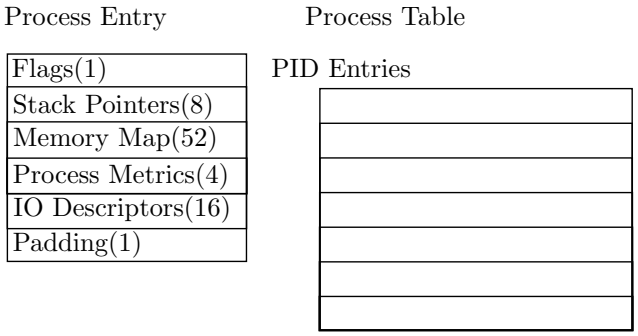


Figure 5.1: The Process Table Structure

The Scheduler has a number of data structures to keep track of relevant information. The central structure is the Process Table. This is a fixed size array of Process Entries. Each process entry stores all information pertinent to the process execution (More process information like 'envireonment variables' may be stored on the disk). This table acts as the main record for processes, the position in the table corresponds to the PID of each process.

This structure is however very slow when searching for processes with specific attributes, and also contains no information about which process should be scheduled next. For these reasons the Scheduler also contains a number of linked lists, each storing a list of processes that are 'interesting' in some way.

The central list is the Active Process list; This list contains all processes that either want CPU time, or are known to want CPU time in the future. The order of this list is important because when the scheduler receives a time interrupt, it will schedule the first process on this list that is awake now, moving the old process to the end of the list.

Using a linked list to store the active processes makes it very fast to select a new process to run, and as the scheduler maintains a handle to both ends of the list, both getting the first item in the list and putting something at the back of the list is O(c) time.

The Decision to keep sleeping processes in this list is due to fairness of execution. The Scheduler policy is that every process will receive one time-slice on the CPU before any processes gets a second time-slice. When a process sleeps, it is making the declaration that it will want CPU time later, just not right now. If we were to leave it out of the scheduling queue until it was woken back up by the events module, and then add it to the queue, it would always be waiting too long. Alternatively, if we always scheduled it immediately after it woke up, then it might get more slices than it should. The solution it to put it in the scheduler queue while still asleep. Once it is at the front, it will stay there until it gets woken up. If it wakes too early, it will still have to wait until each other process has had its fair share of time.

The Scheduler also maintains a number of other lists, each containing processes with 'specific' attributes. There is a list for any paused processes, a list of input capable processes, and list of output capable processes. Using seperate lists for each of these 'interesting' attributes means there is never any need to search the process table for this information, also the lists[by being utilised as queue's] can be seen to keep the order of these processes birth (or the order in which they gained the specific attribute stored).

5.1.2 The Process Entry

Figure 5.2: The Process Entry Structure

5.1.2.1 The Flags

The Flags in a process entry store boolean information:

- init : This process has been created, but not initialised for execution.
- dead : This process is dead; this entry may be reused at any time.
- skip : This process should not be scheduled for its next one time slice.
- paused : This process is currently asleep.
- has_input : This process can be sent keyboard input.
- is_hidden : This process has a reserved portion of screen space but is currently hidden.
- forked : This process is a forked child of another (this will get cleared if the process calls exec).

5.1.2.2 The Stack Pointers

The stack pointers are the only registers that need to be stored and restored when performing a context shift, this is because all other registers are paused onto the stack by the CPU during an interrupt. They are stored here and are only valid while a process is not currently having a time slice.

5.1.2.3 The Memory Map

Figure 5.3: The Memory Map Example

The memory map stores the physical pages of memory that this process has been allocated. Most entries are direct pointers to the physical addresses of the page being used (or arrays of such addresses]. However, the heap_physical_page is actually the physical page that stores a list of physical pages for the processes heap (see diagram above). This is due to the large memory demand that some processes have.

5.2 The IO Manager

The IO manager manages all input and output to and from the userspace applications, All system calls that corrispond to printing tot he sceen of accessing keyboard input lead here.

5.2.1 Keyboard Input

When a userspace program wants it access keyboard input, it first creates a pipe (using the pipe system calls as discussed later) and registers this pipe with the kernel. This pipe descriptor is held in the IO part of the process table.

During this registration, the IO manager also adds the pid of the requesting process to its 'inputable' list (using the same process lists implementation as the scheduler does for its list.). This list will store all the processes that have requested keyboard input in the order that they made the requests.

The use of this list means that when the user requests that keyboard input be switched to a the next process, the IO manager simply has to push the currently focused processes into the back of this list, and pop the next one from the front. This means that switching the focused processes is always O(1).

When a key is pressed and the keyboard driver decides the key press is for the user space, it passes the character to the IO manager which simply writes the character into the input pipe provided. If the process is sleeping (waiting for input) the IO manager will request that the scheduler wake the process up.

5.2.2 Screen Output

The output part of the io manager has more to do, since it is managing a finite and very scarce resource. To consider the screen manager 'fair' we say that; Any request for a portion of the screen by a user space process must by accepted and fulfilled, A portion of the screen can only be under the control of one user space processes at any one time, and The user should have freedom to decide which processes they can interact with at any time.

The major problem with these rules is that rule one and two don't get along, how can we always allow user space processes to get the part of the screen they want, and also never allow overlapping processes. The solution chosen is to hide 'old' processes if they overlap with new ones.

To acquire a portion of the screen the user space processes must register with the kernel a number of things; A pipe for printing in 'progressive' mode, The coordinates for the top left corner of the requested are, and A width and height for this area.

The IO manager first appends this process to its 'outputable' list and populates the io part of the respective processes table entry with the remainder of the metrics, filling the column and row potion with 0 and 0.

It then checks for any over lapping processes, these are processes that are in the 'outputable' list who's screen are collides with the new request. For any overlapping processes, the IO manager first pauses that processes, giving it the $is_hidden flag. The itusesmalloc()torequestaportionofthekernelsheapspaceof$

The above process ensures that, from a processes perspective, it always has full control over a portion of the screen. This is because when it doesn't have control of that part of the screen, it is paused and cannot be executed.

When the user requests to see one of the hidden processes, the IO manager repeats the process above, but instead checking for overlapping with the processes that the user want to bring to the front. Once any overlapping processes have been correctly hidden, it copies the snapshot of

the new process back tot he correct area of the screen and unpauses it, moving it back to the 'outputable' list.

5.3 The Flat vs List interlude

During the discussion of the scheduler and the IO manager I have mentioned the use of lists to store PID's in a number of situations. These lists are ultimately an exceleration structure over the process table its self. In the first design, the scheduler and IO manager dealt only with the process table, however as noted above this table is very slow to use for many operations, namely searching for attributes.

For example, when scheduling a new processes, the scheduler would have to do a linear search across the entire processes table, including all dead processes. This meant that the longer the system ran (and the more processes that had been run) the slower the algorithm, even if all these processes were already dead.

Secondly, when a process requested an over lapping portion of the screen, the IO manager would also have to search the entire process table to see if any other processes overlapped.

The solution was to introduce a number of single linked lists, and maintain pointers to the head and tail of these lists.

By storing the process information in lists, we could accelerate the algorythm in many different places.

The scheduler nologer had to search the process table atall, it simply needed to pull the next process out the front of the respective list.And although the IO manager still needed to search its list, the algorythm now needed only to search the number of processes that have output, not every processes that has ever been run.

But if lists are so good, why not remove the process table all together?

The answer is two fold; Data duplication and direct addressing.

Processes may find them selfs in many of the systems process lists, An active process with input and output is in atleast 3 lists at any time. If these lists directly contained the process entries, they would have to be coppied many times arround the syterm, which would make keeping themup to date slow and difficult. By having the process entries in one place and linking them into the lists indirectly, we can prevent duplication and maintain nice centralisation of informaiton.

Secondly, we often want to perform an action knowing the PID in advance; like setting a process flag of updating the IO part of a process. By storing the Process Entries in an array indexed by PID. These actions are always constant time.

Below is an over view of the the kind of accelrations this system creates.

n is the total number of processes run for the entire uptime of the system p is the number of active processes o is the number of outputable processes i is the number of inputtable processes

it always holds that: o ¡= p i ¡= p p ¡= n

Figure 5.4: Runtime speed-ups of flat vs list.

5.4 The Hard Driver: Drivers and Filesyste,s

5.4.1 The IDE Driver

The hardware access portion of the IDE driver was not written by me as indicated in the code, I simply adapted it to fit my system.

I did however add a cache above the driver to massively increases read write times.

The caching was necessary because hard drives are slow. But also the manner in which they are accessed is not idea, in an IDE device, the only way of getting data on or off the disk is in 512 byte segments. This means if we want to write one byte, we must read 512 bytes into memory, change the target byte, then write all 512 bytes back to the disk. This is very slow. Especially as it is common to want to write sequential bytes to disk; writing 512 bytes would require 512 reads of disk, then 512 writes to and from the same section of the disk.

5.4.1.1 Data Structure

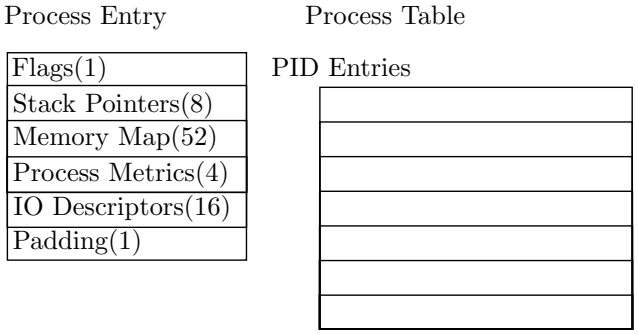


Figure 5.5: The Process Table Structure

The primary structure here is the cache table, as above, it consists of a number of descriptors and a corresponding number of 512 sections of memory.

The caching layer provides the functions request$_r$*eadandrequest*$_w$*rite*.

When these functions are called, the cache first searches the cached blocks for a block corresponding to the requested read/write address. If it finds it, it returns the pointer to the cached block. If that block is not cached, it reads it off the disk into a new block and then returns the pointer to the new block.

If there are no new blocks available, the cache writes all modifies blocks back to disk, wipes the entire cache and then tries again.

It may seem inefficient to wipe the entire cache every time there cache fills however; the cache is very small, and purging it in the worst case only takes as long as writing the bocks back to disk, it is very common for a file to be read/write (and therefore be moved into the cache) only once in the entire life of the system, so a large amount of the cache wont be needed again any time soon, the potential algorythm for a more sophisticated swap of old blocks (perhaps a working set clock algorthm) can take more time to execute every time we need one more block than simply throwing away the whole cache, Finally, at every system context shift (every 10 ms) the cache is written and purged anyway.

This 10ms purge happens to both ensure any changes a user space processes has made to the disk is actually written and is not sat in a cache, but also becuase it is assumed that the next

user space processes will want to access different portions of the disk and will therefore want a clean cache.

This cache is extremely efficient and reduces the number of read/writes and therefore the time its takes to read files by a huge factor, however there is still one slight slowdown, and as the hard drive is a systems biggest bottle neck, it was seen as appropriate to address it.

It is very common for the same block to be requested repeatedly, this is due to the nature of the file system discussed below. Most the file system structures are sequential bytes, and the data its self is in sequential 4096 byte sections. This means that its very common for one 512byte harddrice block to be read/written for every byte (512 times) before we are done with it. This is only interesting because whenever a read/write is requested from the cahce, it searched its cached blocks for the appropriate block.

This search is slow. Especially as the block we need is almost always (511 out of 512 times) simple 'the same one we used last time'. Because of this, the final algorythm remembers which block it served last request, and always checks that block first. That means that 511 out of 512 times the algorythm is O(1) instead of O(n), and the other 1 out of 512 time, its O(n+1) instead of O(n).

5.4.2 File System

The file system (called MRFS from here on out) used was a modification of a piece of second year coursework. This coursework required the create of a file system along with a set of file read write interfaces. These parts will therefore only be discussed briefly. A new set of abstractions were created from scratch for the user space system and these abstractions will be discussed in more detail.

These new abstractions were created due to the unsuitability of the old abstractions for user space. The old abstractions will be refered to as the kernel abstractions as they are used through the kernel. These kernel abstractions are unsuitable for the userspace as they involve calling malloc, and returning a pointer to this memory. Its fobbiden to return a kernel pointer into user space, and having the kernel malloc into the user space is not very good practice. therefore a userpace set of abstractions that provide handles to the TODO

Although MRFS was taken from a previously complete project, it was significatly modified TODO

5.4.3 File Directory Structure

Figure 5.6: Runtime speed-ups of flat vs list.

MRFS uses an inode structure, this means that every object on the disk is represented and described by an inode. Ecah inode can be either a directory or file in the implemented system. Directorys have other file system objects as children whereas files have data blocks as children. Inode structures are very common structuresand were therefore esy to find information on and implement. MRFS implements uses bitmaps for storing which inodes and data blocks have been alocated on the disk. It contains no caching as caching was done on a lower level (see IDE driver above). Every inode must be a child of annother inode except for the root inode, called root or '/'. All inodes also must have names, unique to their parent node. this means that any file system object may be referenced by its name along with its inode number.

5.4.4 File Directory Kernel Abstractions

The kernel abstractions are simple in nature, most take a directory path and a file name, and operate upon that file. Those that operate on directories still take 2 strings, one as the path to the parent directory and one as the name of the directory wanted.

These abstractions all work 'all at once' meaning that they deal with files as a whole; reading a whole file into memory, writing a section of memory as an entire file. This is method is fit for the kernel as it never needs to read/write sections of files off disk.

5.4.5 File Directory User Space abstractions

These abstractions are the endpoints for all file/directory system calls. The follow the open, read, close pattern, except that there is no need to close a file. To open a file or directory, the full absolute path to the object is requried. The file system then returns a structure describing the file; its Inode number, its size, its names size, and two pointers, one into its data and one into its name. This structure is the same for both directories and files. The poiters into the data are initialised to point to the end of the fileor directory.

Once a userspace program has aquired this structure it is used with all the remaining calls. These calls all work on a 'on at a time' method. Meaning that writing or reading from a file must be done one character at a time. Reading from a directory returns one file structure at a time, where each is a valid file handle to one of the children of the directory.

These abstractions mean that storing this information and any mallocing must be done at the user-space applications discression. It also allows for eisily reading parts of a file, and updating the file without destroying all the old ccontents

Chapter 6

Log

6.1 Week 1

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

6.2 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.

6.3 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.

6.4 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.

6.5 Week 5

- major bug fixing.

6.6 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.

6.7 Week 7

- finished work on interim report.

Chapter 7

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:

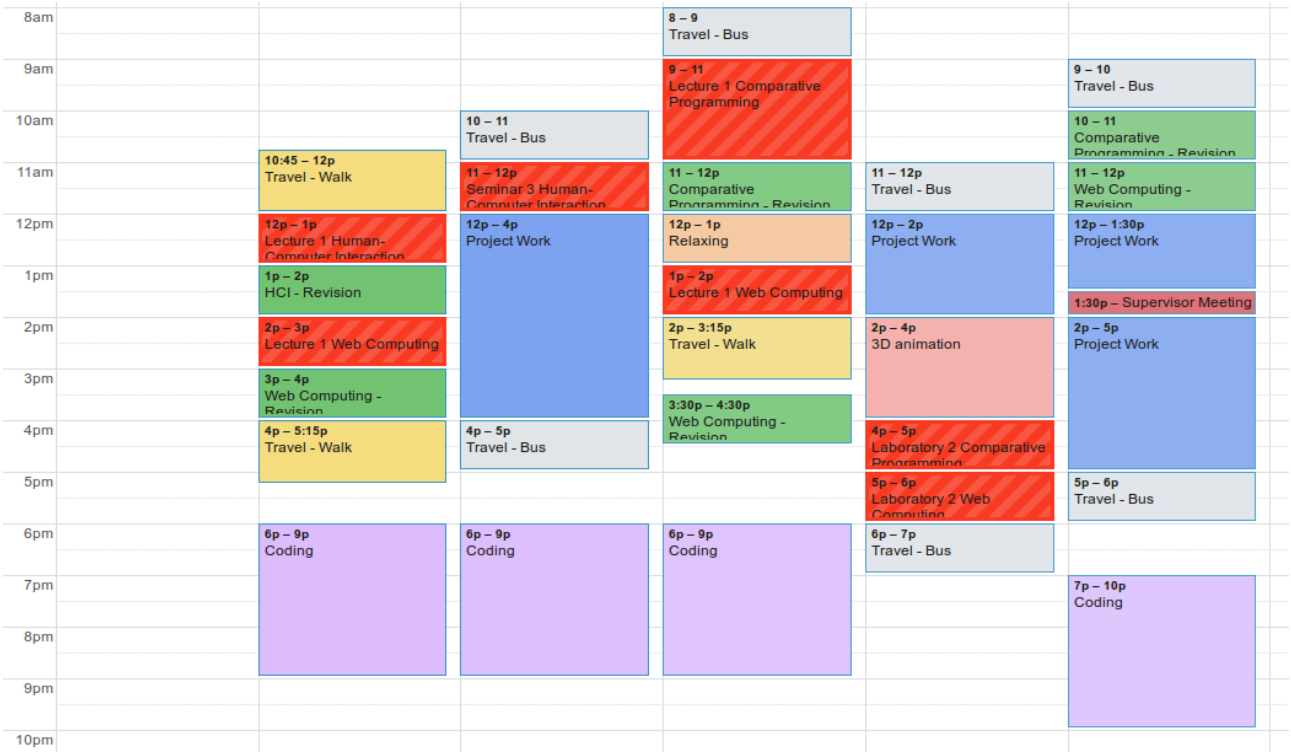
This week we discussed the state of the project, some goals to hit in the upcoming weeks and the proposal report due in next week.

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.