Operating System Report

Daniel Stokes

Table of Contents

Introduction

This report documents the implementation, structure and features of my Operating System (OS) which was written in partial fulfillment of the requirements for Advanced Operating Systems. Section 1 of this report describes the features that are contained in the current version. Section 2 explores the process of developing the OS and delves into the implementation details of the OS. Section 3 addresses the usage of the OS from an application perspective. Section 4 describes the test suite used to verify the OS correctness. Finally, Section 5 addresses some extra features that I would like to add but was unable to within the time constraints.

Section 1 - Features

Architecture Support

My OS targets 32 bit x86 CPU architectures with up to 4 GiB of system RAM. It also relies on peripheral components for generating timer interrupts as well as obtaining keyboard input from the user. There are future plans to support loading applications from external disks but this does not exist in the current implementation.

Original Specifications

The original specification for this assignment described a range of interfaces the final OS must fulfil. These are found in *src/kernel.h* in the OS. All these operations except the initialization functions *OS_Init()/OS_Start()/OS_Abort()* and *OS_InitMemory()* utilise syscalls to separate application execution from kernel execution. It is illegal to call the initialization functions from a user application.
The first set of functions were the initialization functions *OS_Init()/OS_Start()/OS_Abort()*. The *OS_Init()* function was implemented and initialises all subsystems in the OS. The *OS_Start()* starts a login shell and enables interrupts. The *OS_Abort()* function is reached when a kernel assertion fails, in which case execution is permanently suspended until a power cycle occurs. Currently the OS does not support power operations, though this may be done as future work. The implementation for these can be found in *src/kernel.c.*
The next set of functions were the process management functions *OS_Create()/ OS_Terminate()/OS_Yield()/OS_GetParam()*. The *OS_Create()* function sets up a new process and adds it to the appropriate scheduling queue then returns control to the caller. The new process is not run immediately, even if its scheduling priority is higher. The *OS_Terminate()* function terminates the currently running process. It does not do any cleanup inside the function, this is instead delayed until the next process with sufficient resources is scheduled. This function never returns. *OS_Yield()* yields the CPU to another process in accordance with the scheduling specifications outlined in *kernel.h*. When the process is rescheduled the program will

continue from this function. *OS_GetParam()* gets the param set when the process was created. The implementation for these can be found in *src/processes/process.c.*
The next set of functions were the semaphore primitives *OS_InitSem()/OS_Wait()/OS_Signal()*. These functions manage semaphores in the manner described by the requirements. Additionally, there is an extension to these functions provided in *src/sync/semaphore.h* that allows them to be run in *RELAXED* compliance mode. In this mode semaphores can be initialised with negative values and signal on threads that have not previously called *OS_Wait()* on the corresponding semaphore. This allows applications to wait on the results of a different process without having to busy loop on the result of a FIFO queue. The implementation for these can be found in *src/sync/semaphore.c.*
The second set of synchronization primitives are the FIFO primitives *OS_InitFiFo()/OS_Write()/ OS_Read().* These functions implement a non-blocking First-In-First-Out queue in accordance with the specification. The implementation for these can be found in *src/sync/fifo.c.*
Finally, there are the memory management functions *OS_InitMemory()/OS_Malloc()/OS_Free().* The *OS_InitMemory()* function is called by *OS_Init()* and should not be called anywhere else. It initialises both the physical and virtual memory subsystems. This includes obtaining information from the memory map generated during the boot process and initialising the kernel virtual heap space. The *OS_Malloc()* function allocates a virtual address range as specified by the requirements in *kernel.h*. It also assigns physical pages to the virtual addresses at the time of allocation as the OS currently does not support allocating physical pages only on a page fault. *OS_Free()* frees a virtual address range as well as the physical pages associated with it. This function will do a best effort check that the pointer is valid, however it relies on a magic number, so an application should not rely on this behaviour. This currently may require allocating some extra memory, so in rare cases this may fail to free a valid pointer.

My Extensions

The most significant extension to the OS was the support of Virtual Memory. This includes a range of other extensions that complement this. In my OS virtual memory is per application, with each application sharing its own address space but containing a shared kernel memory area. This utilises a higher half kernel approach which places the kernel at *0xC0000000* and the user application in the area below. This means that each application shares the address space in the range *0xC0000000-0xFFFFFFFF* and differs for the address space *0x0-0x7FFFFFFF*. The stack for each application grows downwards from *0x7FFFFFFF* (up to a maximum of *DEFAULT_STACK_SIZE* as defined in *src/processes/process.c*) while the rest of the address space is available for use via the *OS_Malloc()* API. To complement virtual memory, there is a physical memory subsystem for allocating pages. During boot the BIOS is queried for a memory map which is passed to the kernel. This subsystem then processes this memory map to find memory valid 32 bit physical memory addresses. A bitmap is then built to represent all valid pages in the system, which is used for allocating and freeing memory.
The kernel also exposes an API for interacting with the console. The first API is a *printf()* like interface that allows for printing format strings to the console. In addition, the console API also supports changing colours for more expressive representations. Finally, there is also support for

reading input from the user using the *readline()* interface in *src/peripherals/keyboard.h*, which prompts the user with the provided string and then reads a line of input. Currently, this is the only method of interaction with the user, but I am looking to extend this in the future.
To allow the OS to perform an actually useful function a shell must be provided. The OS currently comes with a simple shell interface that prompts the user for a command. Currently, the only commands supported are the TEST, UPTIME, EXIT and HELP commands. The TEST command runs the test suite described in Section 4. The UPTIME command prints the current OS uptime in a time unit of the users choice. The HELP command prints the valid commands, while the EXIT command terminates the shell and enters an idle state.
The OS also comes with some scripts to help with GDB debugging. The script *qemu-launch.sh* can be used to run the OS using Qemu. The *.gdbinit* file can be read by gdb to establish a connection to the running OS in qemu. It also adds a range of helpful breakpoints for debugging. These breakpoints integrate with the builtin assertion framework, allowing the debugger to break on warnings and failures during execution. For more information on these scripts you can look in *README.md*.

Section 2 - Implementation Details

This section will describe the implementations for come of the core systems of the OS.

Booting

Boot Sector

The boot process begins with the boot sector being loaded and executed by the BIOS. The implementation used in my OS is based primarily on the x86 boot code we were given at the start of the project. I made a couple of small modifications to this sector to get it running. The first modification was to increase the loaded kernel offset to *0x10000* from *0x1000* since at the lower position it would overwrite the boot sector. The second change I made was to make the size of the kernel to load variable. This is set as part of the build process based on the size of the actual compiled kernel. Finally, I added some code to get a memory map from the BIOS. This code loads the memory map at *0x8000* just before the actual loaded kernel and after the boot sector which is loaded at *0x7c00*. The code for doing this is taken from the OS dev wiki[1]. In order to fit this into the boot sector I had to remove many of the prints in the boot sector. The implementation of the boot sector can be found in *boot/*.
Once the boot sector is complete the CPU is initialized in 32 bit protected mode at ring 0. The boot sector ends by jumping to *0x10000* where it loaded the kernel.
One limitation of the boot sector as it exists currently is that it can only load a kernel containing at most 128 sectors (64KiB) so my kernel is somewhat size constrained. Currently, if I compile

[1] http://wiki.osdev.org/Detecting_Memory_(x86)

the kernel with the *-O3* flag I exceed this value, but the *-Os* flag leaves plenty of space for growth at the cost of some performance.

Kernel Entry Point

Once the boot sector has jumped to the loaded kernel there are still a number of things that need to be set up before we can switch to running C code. The most important thing is setting up virtual memory. Since code executing in virtual memory mode needs to be linked against the correct address this means we need to set up the kernel address space in assembly before we jump to C code which will be compiled assuming it resides at virtual address *0xC0010000* (see the discussion on upper half kernel in Section 1). The reason for the offset by *0x10000* is simply for convenience as it allows a simple mapping between physical and virtual address. As well as the kernel code we must also map any other physical addresses into the virtual address space. The first such value is the gdt that was setup by the boot sector. The kernel is compiled with an identical copy which can then be loaded using the *lgdt* once virtual memory is enabled. The second value is the video memory buffer at *0xb8000*. This range is used to write to the console by the OS, so it must be mapped to the virtual address space. This is done by reserving an empty page in the kernel .bss space which is then skipped during the mapping process and replaced with the address for video memory. The next value we must preserve is the memory map the BIOS loaded for us. To achieve this we once again reserve space in the kernel and this time we simply copy the values one into this region of the kernel and throw away the original copy.
Finally, the most complicated value to map is the page directory and page tables themselves. Mapping these values seems quite complicated on the surface since you must somehow map the physical pages to known virtual addresses which can then be accessed and modified in the future to update the page tables. However, one can observe that the format for a page directory is very similar to that of a page table. This allows one to simply have an entry in the page directory reference itself. In my implementation I chose to use the last page directory entry which means the upper 4MiB of the virtual address range refers to the page directory and page tables. This fits nicely with the upper half kernel approach since it ensures that the user still can use the entire lower address space.
One difficulty in the upper half kernel approach is that the code changes location when you switch to virtual memory. In order to fix this we start with two mappings for the code, one is an identity mapping which sets the virtual address equal to the physical address and one mapping with the desired virtual address. Once the switch is done we can now jump to the desired offset in the upper half and then remove the old mapping from the page directory. Since the current kernel is assumed to be less than 4MiB total space we can do this by wiping the first page directory entry. At this point the kernel is operational and we can jump to C code. For more information on this section see the *kernel_entry.asm* file.

Kernel Initialisation

Once virtual memory is set up the assembly jumps to the *entry_point()* which is the entry point for the C code. This function calls *OS_Init()* and then *OS_Start()*. *OS_Init()* starts by registering

the start up thread as the entry process. This ensures that there is a process state attached for initialisation functions that expect it. It also registers it as the idle process. Next it clears the screen and sets the colour to green on black for some 1980s feel. It then initialises the IDT, which sets up the Interrupt Service Routine (ISR) and Interrupt Request (IRQ) handlers, and then registers the syscall IRQ. Next we initialise the memory subsystem via *OS_InitMemory()*. Next we register the timer and keyboard IRQ handlers and set up any state needed. The timer is initialised to run at a frequency of 10000Hz. Finally, we initialise the synchronization and process subsystems.
In *OS_Start()* we launch the shell, which is the entry point for user interaction and then enable interrupts and enter an idle loop. Since we are registered as the idle process in *OS_Init()* control will return here whenever there are no tasks to run. To prevent the CPU from busy looping we use the *hlt* instruction to suspend execution until an interrupt arrives (likely a keyboard or timer interrupt) which may then schedule a new process.

Memory Management

Memory Map

During the initialisation phase we look at the memory map provided by the BIOS and parse it into a usable form. We extract all the usable ranges that fit in a 32 bit integer (currently only 32 bit physical addresses are supported) and store them in an array. We expose three interfaces for interacting with this. The first allows the kernel to get the amount of usable memory. The other two allow mapping from a physical address to a canonical page ID. These page IDs are sequential and ignore holes in the memory allowing us to use a bitmap for physical memory management. For more information look in *src/memory/meminfo.c*

Physical Memory Management

Physical memory is managed through the use of a bitmap. This bitmap contains one bit for every physical page in the usable memory space. The bitmaps are indexed by the page IDs provided by the memory map API. To allocate a page we scan the bitmap to find a set bit then clear it. We then use the API to convert the ID of the bit into a physical address we can return to the user. Likewise to free a page we convert the physical address to a page ID and set the corresponding bit to mark it as free.
In order to ensure that we do not allocate one of the kernel pages to a user application we must ensure that any pages used by the kernel are marked as used during startup. Fortunately, using the *linker.ld* script we can get the linker to provide the information of what virtual address range the kernel takes up. If we remember that during the boot process we mapped the kernel to *0xC0010000* and the physical address begins at *0x10000.* This means we can subtract *0xC0000000* from the virtual address provided by the linker to find the corresponding physical address.
For more information on look at *src/memory/physical_memory.c*

Virtual Memory Management

The page directories and page tables are managed by the virtual memory subsystem. This is one of the more complicated pieces of code in the OS since it is responsible for creating and freeing new process address spaces. The basic job of this function is to fulfil two types of queries. The first is to allocate physical pages to a virtual address range by updating the necessary page directory and page table entries and the second is to set up and tear down the virtual address space of processes.

There are two parallel interfaces for allocating address ranges, the normal application allocation and the kernel allocation. These functions are very similar, except that the kernel family can make certain assumptions about the state. One future piece of work would be to merge these paths since currently there is a significant amount of duplication.

To allocate a range the function will first check that a directory entry exists and if it does not it will allocate a page for it. Once the OS has allocated the page table it will query the physical memory management API for a free physical page and insert it into the corresponding page table entry. This process is repeated for every page in the requested address range. Finally once all of this is done the updated page table entries are flushed using the *invlpg* instruction.

Freeing is effectively the inverse of this. We first free the page by wiping the corresponding page table entry, we then check if the page directory contains any valid entries and free the associated page if all memory associated with it is free. Finally, we once again flush all the entries.

The second API is for allocating a new process address space. To do this I introduce the concept of scratch pages. These pages are temporary virtual addresses that are allocated while setting up or tearing down a new process address space. This means that a process must have free virtual address space as well as free physical pages to either setup or teardown another process. When allocating a scratch page we allocate a virtual address from the current process memory and then map it to one of the target process' pages. During setup this is used for the page that will eventually become the new process' page directory. To do this we allocate a virtual page and a physical page together in the current address space. We then clone the kernel address space into the upper section of the page directory. This means that the new process will share the kernel address space of all the processes, since all physical pages are shared. Importantly, we allocate all page directory entries for the kernel address range up front so that the same physical page tables are shared so that any update in one address space propagates to all others. After this is done we must set up the stack for the new process. This is done by allocating physical pages to the requested number of pages at the top of the user address space. Note that this requires a second scratch page since the page directory is not already allocated. Once we have finished all the setup for the new address space we wipe our scratch page reference to the physical page (without freeing the physical page) and return the physical page directory to be loaded into the *cr3* register when the new process is started.

Freeing a process address space is somewhat similar, we first allocate a scratch virtual address and associate it with the physical page from the terminated process. We then loop through all directory entries finding and freeing any valid page tables. For every page table we associate it

with a scratch page and then loop through all its entries freeing any leaked pages the user did not free. Once all these pages have been freed we can free the page table physical page. We can reuse the same scratch address for all page tables so we only need two addresses for freeing a process. Finally, once all the application pages are freed (skipping the kernel pages which are all shared) we can free the page directory and release our scratch pointers.
For more information see *src/memory/virtual_memory.c*

Heap Management

To manage memory allocation we have a virtual heap object. This object represents a virtual address space and provides functions for allocating a freeing memory in accordance with the requirements for *OS_Malloc()* and *OS_Free()*. It can also be used for allocating kernel memory. The heap takes a range of virtual addresses that it can allocate from, functions to allocate and deallocate addresses in this range (these hook into the virtual memory management subsystem described previously) and allocation flags indicating the access permissions of memory allocated via the heap. In the heap initialisation a pool of free list nodes is preallocated for future allocations. This is suboptimal and should be reworked in the future, since it can lead to situations where memory cannot be freed due to insufficient free list nodes. Currently the system uses one free list node per allocatable page as an estimate of how many nodes will be needed.
The allocation function accepts a size and then allocates memory via the first-fit algorithm. This runs by scanning through the free list to find the first range with sufficient space to allocate. Once a valid range is found it will allocate any new pages using the function provided when the heap was created, forwarding the flags to the virtual memory management subsystem.
Likewise, to free a page we iterate through the free list to find the correct place to insert. Once found it will either insert a new node or merge with its neighbouring node if possible. At this point it will free any pages that are no longer referenced. This happens if the page is entirely contained within the freed region, or when one of the freed region's neighbours contains the rest of the page at which they border each other.
To determine how much memory to free we utilise the two bytes specified by the requirements to store the $\log_2$ of the allocated size, since it is a power of 2. Since this only requires 5 bits, I also store a magic value in the upper byte to indicate that the allocation is valid. This allows the free function to catch errors where an invalid pointer is freed.
For more information see *src/memory/virtual_heap.c*

Process Scheduling

The process scheduling supports three types of process. In general each operation has a common and type specific part. The type specific implementations are in the respective files in *src/processes/* and the common implementations are found in *src/processes/process.c*.

Periodic

Periodic process scheduling is controlled by three primary variables, on top of the periodic process scheduling plan outlined in the requirements. The first is the *ppp_index* variable which indicates which slot in the PPP we are currently in. The next is the *next_periodic_start*. This variable stores the time slice at which the next periodic process should be scheduled. Note that this is since the previous process slot was supposed to finish rather than from when the process started execution, so if a device process runs long, the periodic process will receive a shorter slot. The final variable is the *yielded* flag, which tells the scheduling whether the current periodic process has yielded the CPU during the current time slot. This is always set to true during a slot marked idle.

Sporadic

Sporadic process scheduling is managed by the circular list *sporadic_scheduling_list* which uses the *scheduling_list* member of the PCB. The head of the list is the currently running sporadic process. Whenever yield is called the head is advanced to the next ready sporadic process which is then scheduled. If there are no sporadic functions ready to be run the idle process is run.

Device

Similar to the sporadic process scheduling the device process scheduling is managed by the list *device_scheduling_list*. This list contains the device processes in the order they need executed. When a timer tick occurs the time slice is compared to the first ready device process in the list to see if the device process needs to run. If it does the process will be scheduled. When a device yields the CPU it will calculate its new wakeup time based on the current time slice and insert itself at the correct location in the scheduling list. If a device function is blocked it is not reinserted, instead it is left at its current position in the queue and immediately resumes if it becomes unblocked.

Process Control Blocks (PCBs)

Each process has a PCB associated with it. This contains a range of useful/necessary values needed by various parts of the kernel. Each process is allocated a PCB based from the *pcb_pool* array, where the index into the array corresponds to the process' PID - 1 (except for the idle pcb which is separate).

Process Switching

Process switching is done via the *sched_common()* function, which is called by one of the type specific scheduling functions with a new pcb to schedule. It then sets the state of the current process to READY if necessary and sets the new current process. It then executes the *context_switch()* function which backs up the current state and then loads the stored state of the

new pcb. It must also switch *cr3* and *flags* registers to ensure the new process is running with the correct memory space with the correct privileges. Once the switch is complete, the new process sets its state to executing and then checks for any terminated processes that need cleaned up. Usually this will be the process that was just switched from.

Termination

When a process is terminated it will remove itself from any scheduling structures associated with its type but will not free the PCB. This ensures that the process is never scheduled again in the future. The pcb is then added to the stopped process queue to be cleaned up by a later process. Finally, it yields the CPU and should never be rescheduled.

New Process Creation

Creating a new process is the most complicated task in the OS at present. In the section on Virtual Memory Management I have already discussed the steps for initialising a new process' address space. The process scheduling subsystem is responsible for allocating and initialising the processing and then preparing the process for being scheduled for the first time. First we allocate a free PCB from the PCB pool. This works by scanning the pool for the first free entry with an invalid PID. This marks the entry as unused so the process may allocate it. To allocate it we simply set the PID to its index in the pool plus one. Once allocated we initialise all the fields in the PCB, delegating to the initialisation for the appropriate types' initialisation functions to add the pcb to the scheduling structures. Once this is done we set up the process address space and then call *fork_process()*. This function sets up some state for the new process so when it is scheduled it is ready to run. It also specifies that execution should begin at the *new_proc_entry_point()* function and passes the pcb to this function so it knows which pcb to setup. This function then sets up its heap for the new address space and then calls the function the user requested. When the user function returns we simply call *OS_Terminate()* to clean up the process.

Synchronisation

Semaphore

The semaphore implementation is relatively straightforward. The most complicated part is blocking the processes when the semaphore is not available. The main flow for *OS_Wait()* involves checking if the semaphore value is above 0 and if not we block the process. When the process wakes up it checks if the semaphore is above 0 and if not it goes back to sleep. If the process finds the semaphore with a positive value it will decrement the count and store a flag in the pcb to indicate that it signalled the semaphore and returns. In order to block processes on the semaphore we must have a blocked list associated with the semaphore. When a semaphore is initialised we use the semaphore index to access an entry in a static array. This entry

contains the blocked list and the integer for the semaphore. The blocking process involves appending the blocked process to the list for the semaphore and then yielding.
The *OS_Signal()* function is relatively straightforward, if the compliance mode is *RELAXED* or the current process is marked as holding the semaphore then we increment the semaphore count and unblock any waiting processes if the new count is positive. We then mark the process as not holding the semaphore and return. The unblock logic is slightly more complicated than the blocking logic since we want to favour high priority processes first. To do this we loop through the blocked list and find the highest priority process. Once found we unblock the process and, if it has higher priority, yield the CPU to it. For more information see *src/sync/semaphore.c*

FIFO Queue

Like the semaphore, the FIFO Queue is quite straightforward. It also doesn't have to worry about blocking processes. When a fifo is initialised we loop through a pool of FIFO states and finds an unused FIFO.The requirements are ambiguous as to exactly how a FIFO queue can be released so I defined a separate function *release_fifo()* that released an allocated fifo by setting the head and size pointers to *-1*.This fifo is then returned to the user for use.
The *OS_Write()* function writes into the fifo *buffer* at index *head* and then increments the head pointer. If the fifo is not full it also increases the *size*. The *OS_Read()* function reads from the location *size* entries before *head*, wrapping to the end if this index is negative. The size can then be decremented by one. This implementation forms a basic ring buffer allowing the head to eat the oldest element automatically, avoids needing to dynamically allocate the queue and allows all operations to be *O(1)*. If size is zero *OS_Read()* fails. For more information see *src/sync/fifo.c*

Interrupts

The interrupt subsystem is responsible for managing hardware interrupts. There are two similar but distinct cases that my implementation distinguishes. There are the Interrupt Service Routines (ISR) which are raised by the CPU itself and usually represent an illegal operation that requires immediate attention. Overall my approach to handling these was to simply catch the error and terminate the calling process with a message. The only one that required special handling was the Page Fault. Since currently I allocate physical pages up front this is actually illegal at the moment and indicates an illegal memory access. In this case I simply printed a slightly different message to indicate that there was an illegal memory access rather than just a Page Fault. One other thing of note regarding ISRs is that if an application causes a stack overflow they will trigger a page fault. However, when the ISR tries to process the fault it will push registers onto the stack and cause a triple fault. If this happens the CPU will reset. While in theory it is possible to fix this, the simplest approach requires ring 3 mode. In this case we can have a separate stack for ring 0 that has not overflown.
The second class of interrupts I support are the Interrupt Requests (IRQ). I support two variants: interrupt gates and trap gates. Interrupt Gates behave identically to ISRs and differ from Trap Gates primarily in the fact that interrupts are disabled by default for Interrupt Gates but not Trap

Gates. There is a third type of gate called a Task Gate that is currently not utilised. These gates are defined in a table called the Interrupt Descriptor Table (IDT) which is setup as part of the initialisation procedure. By default both the CPU ISRs and the PIC IRQs are mapped to the same interrupts, to fix this the initialisation procedure must also send some commands to the PIC to remap its interrupts to 32-47.
Each gate has a hard coded entry function in *src/interrupts/interrupt_handlers.asm* which push some flags to the stack, backup all the registers and then call the handler in *src/interrupts/interrupt_handlers.c*. The ISR handler simply checks the pushed flags against a hardcoded table, whereas the IRQs are configurable and any user can subscribe to handle the specific interrupts. Currently there are 32 interrupt gates assigned to the PIC, and one trap gate for syscalls at 0x80.

Syscalls

Like IRQs the syscall interface allows the user to subscribe to handle specific syscalls. To invoke a syscall the user stores the syscall type in eax and up to two parameters in ebx and ecx. The user then invokes the trap gate via *int 0x80* instruction. The syscall interrupt handler then calls the registered handler. The result of the syscall is returned in the *eax* register to the caller. For more information see *src/processes/syscall.c*.

Peripheral Devices

Timer

The timer is initialised to run at 10000 Hz and is registered to IRQ 0 (mapped to interrupt gate 32 when the PIC remapping was done). Every tick a counter is incremented, and every 10 ticks (1 ms) the timer invokes the process scheduling preemption function to trigger a process switch. Additionally, the timer provides an API to get the current uptime since the OS was booted. This uses the number of ticks to get the time in either *us* (always in multiples of 100) or *ms*. For more information see *src/peripherals/timer.c*

Keyboard

The keyboard is mapped to IRQ1 (mapped to interrupt gate 33 when the PIC remapping was done). Whenever a keystroke is made the handler reads the character typed from the port 0x60. This returns a scancode value which can be used to translate into an actual character by the *sc_ascii* lookup table. Then, if a process has currently asked for a line of input from the console it will echo it to the screen and add the character to the current line buffer. If the character is a backspace it will remove one character from the buffer and clear that character from the screen. If the character is an enter, the handler will write a new line to the console and unblock the process waiting for a line of input, passing it the completed line of text. The keyboard exposes the *readline()* function for processes to ask for this input. This function first checks if any other processes are waiting for input. If so the current process will be blocked until the running

process has finished reading from the console. Once the function is given access it prints the prompt provided by the process to the screen and then waits for a line of text to be typed. Everytime a character is typed it is added to a line buffer in kernel memory. If this buffer is too small a new one twice the size will be allocated and the contents copied across. A factor of 2 is used for performance since this makes it so that we will have to repeatedly reallocate the buffer on every key press. A multiplicative factor is used rather than adding a fixed amount so that there is only one allocation per order of magnitude of characters typed, since adding 10 items after 256 characters typed is much more likely to need a reallocation than after 1 character, since we know the user is already typing a lot. Additionally, since the allocation scheme always allocates in powers of two, asking for the next multiple of two will be the most optimal, since we would otherwise allocate the same amount, but have to realloc more frequently. Once the line is complete the waiting process is unblocked and it allocates a buffer in the process' memory and copies the completed line from the kernel memory buffer into the user memory. This buffer is then returned to the caller. For more information see *src/peripherals/keyboard.c*.

Section 3 - Applications

Currently my applications all reside in the *programs/** folder. The only two applications at present are my test suite and the shell. All applications will be launched by a shell command as this is the only application directly invoked by the kernel. In the current architecture there is very little to distinguish between applications and the kernel since they must all be compiled together due to the lack of a disk driver for loading applications from disk separately. Additionally, these applications have full ring 0 permissions since having a ring 3 user mode is not implemented at present. However, in principle applications should only interact with the kernel through the use of the syscall API, or OS wrappers around syscalls. This currently includes all the user functions in *src/kernel.h* as well as other user facing APIs such as *readline(), release_fifo()* and *set_sem_compliance()*. However, there are some functions such as *print()* which do not currently have a syscall API but are still useful for applications to use. In future I plan to extract out these APIs into a standard library that applications can compile against to achieve all these operations. This standard library should have functions of a similar or identical signature as the kernel functions used currently so it can serve as a drop in replacement.
Currently the PPP is set up once on initialisation as per the requirements (currently randomly), however in future I would like to change this policy to make application development more flexible.

Section 4 - Testing the OS

My OS comes with a test suite application runnable by the shell. This suite contains a range of test cases targeting different aspects of the OS. Primarily the tests focus on the external user facing subsystems, in a blackbox manner. The primary subsystems targeted are the memory, synchronisation and process scheduling subsystems. The tests can be found in *programs/test_suite/*

Memory subsystem

The following tests cover the memory subsystem and can be found in *programs/test_suite/malloc_tests.c*

Test Name	*Test Function*	*Test Description*
Small Malloc Tests	test_malloc_small()	This test tries allocating two small buffers and checks they are non-overlapping as well as readable and writable.
Large Malloc Tests	test_malloc_large()	This test tries allocating two large buffers and checks they are non-overlapping as well as readable and writable. It also checks that freeing the buffers resets the state
Underflow Regression Tests	test_malloc_underflow_regression()	This test addresses an underflow bug that existed in an early implementation of malloc where allocating in a specific manner could cause a failure
OOM Malloc Tests	test_malloc_oom()	This test attempts to allocate more memory than is available and checks the allocation fails.
Many Malloc Tests	test_malloc_many()	This test allocates many same sized allocations at once and checks the memory is not overlapping and is cleaned up correctly
Variable Malloc Tests	test_malloc_variable()	This does many allocations, increasing the size by one for every allocation. It also checks memory consistency.
Fill Malloc Tests	test_malloc_fill()	This allocates many odd sized buffers until the allocation fails then frees all the successful allocations.
Interleaved Malloc Tests	test_malloc_interleaved()	This test allocates many randomly sized allocations and then frees a random subset of them. It then does more allocations in the freed memory. This repeats many times
Bad Free Tests	test_bad_free()	This test covers a range of invalid free scenarios and checks that free fails correctly.

Synchronisation Tests

These tests cover the synchronisation subsystem and can be found in *programs/test_suite/fifo_test.c* and *programs/test_suite/semaphore_tests.c*

Test Name	*Test Function*	*Test Description*
Mutual Exclusion Semaphore Tests	test_semaphore_mutex()	This tests using a semaphore for the purposes of mutual exclusion.
N-Way Exclusion Semaphore Tests	test_semaphore_n_blocked()	This tests using a semaphore for allowing multiple processes access to a resource
Semaphore Compliance Tests	test_semaphore_compliance()	This tests that semaphores meet the STRICT compliance mode specification.
SPSC FIFO Tests	test_spsc_fifo()	This tests using a FIFO queue with one producer and one consumer
MPSC FIFO Tests	test_mpsc_fifo()	This tests using a FIFO queue with many producers and one consumer
MPMC FIFO Tests	test_mpmc_fifo()	This tests using a FIFO queue with many producers and consumers
Invalid FIFO Tests	test_invalid_fifo()	This tests a range of failure cases regarding the allocation and use of FIFOs.

Process Scheduling

These tests address process scheduling and aim to check that scheduling is done correctly. These tests can be found in *programs/test_suite/process_scheduling_tests.c*

Test Name	*Test Function*	*Test Description*
Periodic Scheduling Tests	test_periodic_scheduling()	This test verifies that periodic functions correctly sleep for the requested time. It also checks that they are preempted at the end of their time slot even if they don't yield
Sporadic Scheduling Tests	test_sporadic_scheduling()	This test checks sporadic functions are done in order and dont switch until the previous yields
Device Scheduling Tests	test_device_scheduling()	This tests that device functions are run at the correct intervals and that they are able to preempt sporadic processes

Invalid Process Tests	test_invalid_process()	This test covers a range of negative test cases relating to spawning processes.

Section 5 - Future Work

In addition to various improvements to the implementations of the features above there are a range of extra features that I would like to implement in the future.

User mode

The first feature I would like to support is a ring 3 user mode. In user mode applications would be unable to execute privileged instructions such as those to modify the virtual memory page tables. It can also prevent the user from modifying protected pages in memory, such as the kernel code and heap. The only way to do this would be via the syscall interface used for most application facing functions at present. However, to gain full benefit from this we would need to be able to load the application into non-protected memory so as to prevent access to the kernel, which should be done with the Dynamic Application Loading feature described below. As a result, I have not implemented Ring 3 mode yet.

Disk Driver

The next feature I would like to implement is a disk driver interface that allows loading from disk. Currently we are limited to loading 64KiB from disk during the boot process, and there is no support once the CPU is switched from real mode to protected mode. Adding a driver will allow the Dynamic Application Loading feature below and will also be an important feature for making useful applications, since currently no persistent state can be stored between restarts.

Dynamic Application Loading

The Dynamic Application Loading feature is useful for expanding the set of applications and allowing the OS to support a broader range of applications. In theory once this is done my OS would be capable of executing any 32 bit x86 application that does not use dynamic libraries, without the need to compile them into the kernel. This would also help eliminate some of the security risks with compiling the application with the kernel.

Multi-stage bootloader

Currently the kernel has a limit of 64KiB (128 sectors) that can be loaded from disk. This is due to a limitation in the current boot sector read logic. Additionally, there are features such as the

memory map that require real mode but do not currently fit in the boot sector. To solve these problems I would like to write a multi-stage boot loader that loads a smaller intermediate executable prior to loading the actual kernel. This gives more flexibility in the boot process to allow for a more feature rich experience. Another benefit is that when paired with Dynamic Application Loading the kernel may reside in a file system so it is not trivially locatable. This would be much more difficult to do in the limited 512 bytes available in the boot sector.

Reboots

Currently the OS does not support software reboots. This would be a useful feature once the kernel has some persistent state such as when there is a disk driver. To implement this it is necessary to use machine specific protocols such as ACPI since there is no straightforward method for it. I have not had a chance to thoroughly investigate what is required to implement this feature.