Project Report for Nitrogen: An Operating System for a Portable Game Console

MEng Computer Systems Engineering, Third Year Project, University of Manchester

Jaime Van Hunskerken

May 2019

Project Supervisor: Steve Furber

1 Abstract

This report concerns an operating system kernel written for a portable video game console, the Nintendo DS. This kernel implements a number of features common to other operating systems, some of which are new to this platform. This report describes the development of the kernel, with emphasis placed on the following topics:

- The identification of motivating factors and the determination of project goals.
- The planning process and the changes made to the plan during development.
- Descriptions of each core feature of the kernel, from the perspective of the desired behaviour as well as the implementation required to obtain it.
- Comparisons with competing approaches to the problems addressed by this kernel, and justification for the chosen solutions.
- The testing process and the confirmation of working implementations.
- A reflection upon the work accomplished.

Contents

2 Context

2.1 Background

Operating systems form a fundamental part of modern computing. It is desirable to run many different applications on a single unit of hardware simultaneously, with the capability for those applications to interact with one another, and with that interaction being controlled in such a manner that applications are not able to cause other applications to misbehave by altering their state in ways that are not accounted for. Hardware does not inherently offer this functionality, though it does offer primitives whch allow system software to implement this functionality. Different hardware configurations also exist, and they may use incompabile register interfaces that would cause a program written for one such interface to be nonfunctional when using another interface. It is desirable for application software to interact with an abstract programming interface rather than directly manipulating hardware registers, as this enables the software to run on multiple hardware configurations and also to avoid interfering with the operation of other programs. The system software which fulfils these two aims is known as an operating system kernel[1].

The design, development and implementation of operating systems is therefore important to computing. An operating system is concerned with presenting abstract services to the applications it intends to manage - such as programming interfaces for utilizing resources such as memory, storage, button inputs and graphical output. At the same time, those high level interfaces must be designed in a way that the underlying hardware is well equipped to service. Like many problems in computer science, the challenge of designing an operating system is one of balancing these competing concerns to deliver a robust abstraction.

This project implements such a kernel. The kernel is designed to run on the Nintendo DS portable game console and its successors. All commercial software on this device runs without the aid of an operating system, as the unit is only designed to run a single program at a time. However the unit is equipped with the necessary hardware to implement an operating system kernel.

2.2 Related Efforts

Many operating systems are in use today. A popular example is Linux, an operating system kernel which can run on a number of different hardware architectures. A variation of Linux, Clinux, has been ported to the Nintendo DS under the name DSLinux[2]. DSLinux can run much ordinary Linux software. However there are significant limitations. The DS has little memory by the standards of a modern computer, and DSLinux lacks memory management functionality such as memory paging. Limited to a small amount of memory, many programs will not run at all. The total lack of memory management, in fact, means that applications can freely access other applications' memory address spaces and interfere with their operation.

2.3 Motivation

The lack of an official operating system for the Nintendo DS makes the development of a new operating system appealing. Though unofficial efforts have brought an operating system to this platform, they lack important operating system features such as memory management which the device is equipped to support. However, the memory management hardware on this platform is very different from and more limited than most devices that are designed to run operating systems like personal computers[3]. This presents a unique challenge for the implementation of this feature and related features. The viability of the project is also quite high. The platform uses a significant amount of proprietary hardware for input/output operations, but this is well documented. The CPU on which the device is based uses an instruction set architecture, ARM, which is very popular for low power mobile devices and is one which I am personally familiar with. This platform is a popular target for writing software due to possessing an active community and a well-maintained set of development tools including compilers, linker scripts and libraries.

2.4 Project Goals

This project aims to bring an operating system kernel to the Nintendo DS platform. This kernel shall be called Nitrogen[1]. This kernel shall be capable of loading and executing user programs as processes. The kernel will implement a system call interface that allows user processes to invoke behaviour in the kernel via APIs. The processes will have access to button inputs and graphical output via kernel APIs rather than directly interfacing with hardware registers. The kernel shall implement a form of multitasking to ensure that multiple processes can execute simultaneously on a single hardware unit. The kernel will provide a system call that allows a process to yield execution and allow the kernel to execute another process instead. The kernel shall provide APIs that allow processes to interact with one another, such as by sharing data. The kernel shall implement memory protection so that processes are not able to interfere with one another's memory address spaces except where explicitly permitted. The kernel shall provide system calls that allow processes to create new execution threads and new processes. The kernel shall provide system calls that allow processes to allocate and deallocate additional memory. The kernel shall provide system calls that allow processes to share memory with other processes, enabling another more efficient mechanism for inter-process communication.

[1]A reference to the Nintendo DS's internal development name, Nitro.

The kernel shall allow processes to "block", preventing them from being executed until a need is met (such as another process concluding, or some data to finish copying) so as not to waste CPU resources. Where it is convenient, the kernel will base its functionality off existing efforts (such as the POSIX interfaces implemented by popular operating systems like Linux) though the kernel does not necessarily attempt to remain compatible with these interfaces. If presenting a different interface is more convenient to implement or use then this interface shall be chosen instead.

3 Development

3.1 Planning

From the start, this project was planned to have flexible goals. The plan deliberately sets out a sensible maximal amount of work that may be accomplished on this project. Work that was be completed in earlier weeks would serve as the basis for work accomplished in later weeks. The fundamental and critical elements of operating system development would be completed early so that, if there were any trouble implementing them, there would be more time available to borrow for them at the expense of less important features.

- Weeks 1-3
  - Loading and executing a user program
- Weeks 4-5
  - Develop a mechanism for user programs to issue system calls to the kernel. Implement basic system calls such as reading the button state and printing debug messages.
- Weeks 6-8
  - Implement a co-operative multitasking system. Processes will have access to a ”yield” system call which, when called, will return control back to the kernel. The kernel will then execute another process.
  - Implement inter-process communication. The mechanism shall be based on POSIX signals,which are capable of interrupting processes mid-execution. Processes and the kernel can issue signals.
- Weeks 9-11
  - Implement a preemptive multitasking system. The kernel will regain control of the system if no yield is called within a certain period of time, and schedule another process for execution.
  - Add process isolation. Processes are isolated from one another by utilising memory protection. If a process tries to access memory which it lacks permission to use, a signal will be sent to it by the kernel.
- Weeks 12-14
  - Implement syscalls for gaining access to graphics hardware. User processes will be able to draw a bitmapped image to the screen.

  - Syscalls will be added for dynamically allocating memory, similar to POSIX mmap. Processes will be able to set permissions for other processes to use dynamically allocated memory, thereby enabling the use of shared memory for efficient inter-process communication without system calls.

- Weeks 15-16
  - Processes will be able to block. Blocking processes will not be scheduled for execution. Processes will have access to system calls that will cause them to block, such as hardware-assisted timer and memory copy operations.

- Weeks 17-19
  - Miscellaneous input/output operations will be available to user programs. This includes access to the touch screen interface of the console and access to files on storage devices attached to the console.

3.2 Methodology

The development methodology had a heavy emphasis on incremental building. A new feature was planned to be added to the kernel over a period of a few weeks, and with it some basic user programs which take advantage of. This makes sure features are tested early, and makes it easier to isolate and identify problems if they arise. It also makes it possible for a less functional but still viable solution to be presented if development takes longer than expected, or if unexpected hurdles are encountered during the development of the solution. This also makes it possible for the design to be revised during the development process. If an element of the design demonstrates itself to be a bad idea, this design can be revised before any other aspects of the system come to depend on it.

3.3 Development Environment

The development environment for this project consists of the build tools and the execution environment. The primary piece of software that comprises the build software is the DevkitARM[4]. This is a suite of related software that is used to build software for a variety of platforms that lack free official toolchains, including the Nintendo DS. DevkitARM includes GNU compilers for C and C++, the GNU assembler for ARM, the GNU linker for linking compiled objects into an executable, and linker scripts for target platforms that allow one to link binaries correctly. DevkitARM also provides essential libraries for the DS such as libnds, which provide a basic runtime environment for C and C++ programs to run. DevkitARM also includes miscellaneous tools that make it possible to build software for its target platforms. For example most software written for the DS platform is not distributed as a raw executable (an ELF file) but rather in a custom container format which contains multiple binaries, metadata and an embedded filesystem (an SRL file). DevkitARM provides the ndstool program to package the linker output in an SRL that software launchers on the DS are able to execute.

Beside this, a simple text editor was used to create and modify source files. The source files that comprise the kernel were predominantly written in C++, with certain key sections written in inline ARM assembly. C++ was chosen because it shares the same core semantics and lightweight runtime of C that make it suitable for kernel development, but adds a number of useful constructs like template code generation. ARM assembly must be used in some circumstances where the precise state of the machine must be precisely manipulated in order to achieve correct results such as when switching between two processes' contexts. By using inline assembly, it is possible to share important information - such as offsets of data within structures - between C++ and assembly in a safe and readable manner. Raw assembly files were used where direct interaction with C++ code were not necessary. Familiar tools such as Make are used to automate the invocation of DevkitARM tools to build the source files into a fully built SRL.

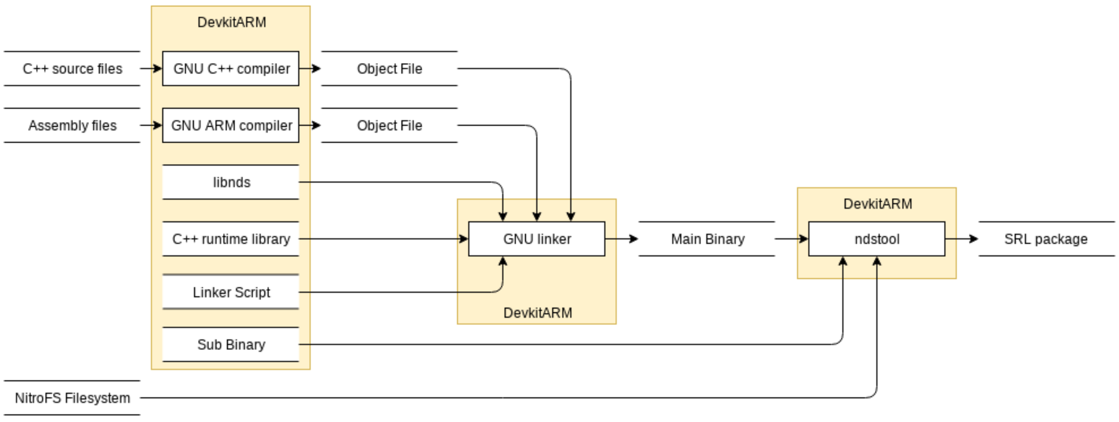

Figure 3.1: Illustrated data flow for the build system.

Two distinct execution environments were used. The first is real hardware. When an appropriate launcher software is installed on a DS unit, they can launch SRL files from an attached storage device. All models of the DS contain a proprietary cartridge slot. Adaptors known as linkers[1] allow users to read data from a Micro SD card using this slot. However, many revisions of the DS also contain a regular SD card slot, which is more convenient than using a specialized linker. The primary unit I used for development is a Nintendo DSi, using an ordinary SD card. However, this technique has some limitations. The edit-compile-debug cycle is rather slow when using real hardware. The SD card must be removed from the console, the binary file under test must be replaced, the SD card must be reinserted and then the console must be rebooted before the test can be conducted. Additionally, the console has no dedicated debugging support. The only way to debug programs running on real hardware is to display text on the screen. There do exist variations of the hardware which have debugging support such as the IS-NITRO-DEBUGGER but these are prohibitively expensive to obtain.

[1]Not to be confused with linker software used to build binary files.

The second execution environment was an emulator. As a popular platform for playing video games, a number of emulators have been developed for the Nintendo DS with the intent of playing those video games on a PC. However, two of these - NO$GBA[5] and DeSmuMe[6] - both feature accurate emulation of most hardware in the console and also feature debuggers. NO$GBA has builtin debugging, while DeSmuMe provides a stub which GDB can connect to for debugging. DeSmuMe proved to be my preferred emulator as it runs natively on Linux and supports GDB, my preferred debugger. Because of these factors, I heavily preferred DeSmuMe to real hardware when developing and debugging my kernel. However, DeSmuMe is also not perfect. As an emulator designed primarily for playing games, certain hardware features such as memory protection are not implemented. Implementing memory protection would harm performance, a key concern for emulators intended to play video games, and software which takes advantage of memory protection is extremely rare. The only way to test certain kernel features that rely on memory protection is by using real hardware instead. This is what was done.

3.4 Loading and Executing Programs

Writing, loading, and executing a user program entails the following:

1. Compile or assemble a program.
2. Open the file containing the program.
3. Copy the contents of the file into a region of memory.
4. Begin executing at the entrypoint of the program.

In practice, there were a number of hurdles involved in accomplishing this.

Firstly, the development tools must be modified in order to compile a user program. The C++ runtime library defines the entrypoint to any program in the _start routine. The _start routine that DevkitARM provides contains various hardware initialization routines that should not be executed for user programs. Therefore it is necessary to write a new _start routine. Though the routine was modified at later times to introduce new features, the initial behaviour was very simple. The basic _nelf_start[2] routine is chosen to be the entrypoint for programs compiled to run on this kernel, and this routine simply calls the nelf_main function of a C++ program without any other initialization taking place. This is similar to how the _start routine calls the main function in most C++ environments.[3] The _start routine in an ordinary C++ environment also executes constructors and destructors for global objects before calling main, but this feature is not used by any programs I wrote and was not implemented by the _nelf_start routine.

A new linker script must also be made. A linker takes object files which contain symbols and combines them into a binary file, which is an image of a program's memory

[2]Nelf is a shorthand for Nitrogen ELF

[3]A different function signature was chosen because it becomes useful to implement threading later.

address space. Symbols are named locations in memory which usually contain machine code but may also refer to storage for the program stack, heap or global variables. A linker script defines the memory regions a memory address space contains and describes where symbols should be placed. For example, the DevkitARM linker script used for the kernel defines that the program stack should be placed in the DS's fast DTCM RAM, while the heap should be placed in slower main RAM. Since user programs should not be interacting with the hardware memory address spaces in a direct manner, a new linker script was written which is unaware of this. All symbols in the user program are defined to inhabit a single contiguous region of memory, which the kernel is free to load wherever it likes.

Opening the file was relatively simple. Once the user program is built, it can be added to the NitroFS filesystem of the kernel SRL. Then the kernel can to read the contents of the binary file.

The next hurdle came in the form of copying the file to memory. Most modern CPUs contain memory management units which implement virtual memory. Virtual memory allows an operating system to present a view of memory which is distinct from its physical layout. This means the memory view for a given process can be precisely as is defined in its binary image, which enables it to use global variables through absolute addresses. Without virtual memory it may not be possible to load processes in this way - for example if they define symbols at memory addresses where no physical memory is mapped, or where another process is already loaded. With no virtual memory a solution must be chosen, of which there are two common approaches.

1. Compile and write position independent executables (PIEs). Position independent executables do not contain any code whose behaviour depends on what location in memory the program is loaded to.

2. Have the kernel program loader perform load-time relocation. Programs are modified when they are loaded so all symbols, and all absolute addresses are updated to point to the new location of symbols.

First I considered position independent executables. The GNU C++ compiler supports position independent executables with a compiler flag. The implementation is as follows. One CPU register (usually R9) is reserved to hold a pointer to a Global Offset Table (GOT), which is an array of addresses. Symbols which cannot be accessed by program counter relative offsets are performed by looking up entries in the GOT. For the GOT to contain the correct addresses, the C++ runtime library must update its contents when the process first executes and before global symbols are used. This is usually done either by the _start routine or a routine invoked by _start.

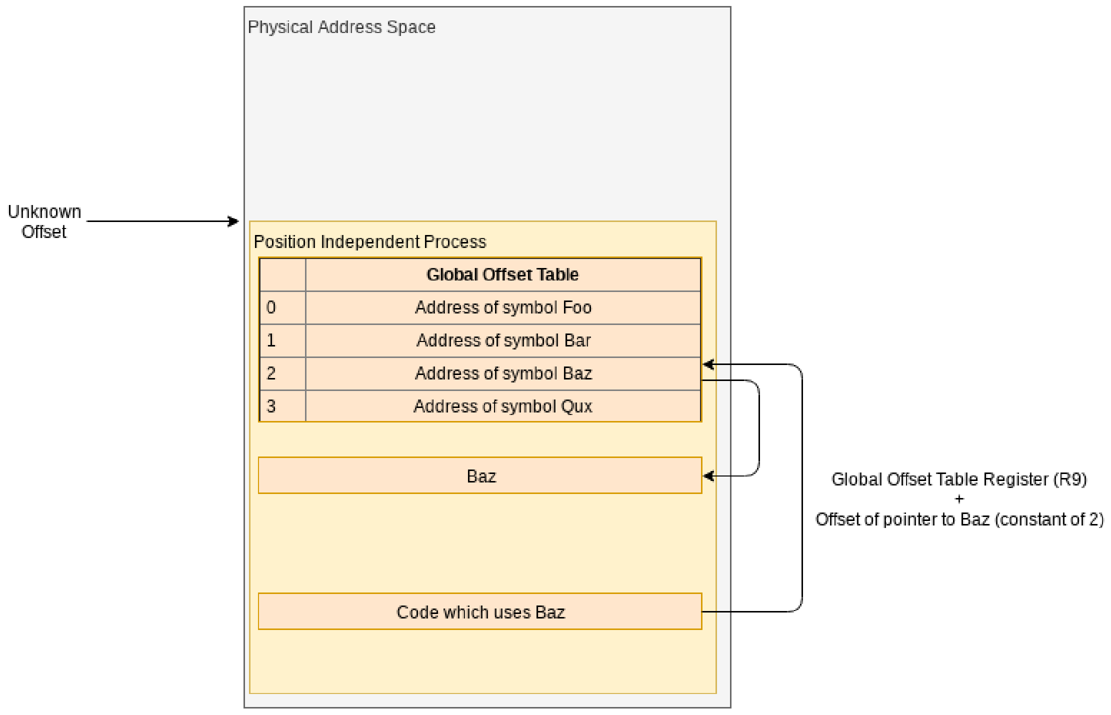

Figure 3.2: Usage of global offset table.

Next, I considered load time relocation. Relocation information is included in object files when a source file is compiled or assembled. Relocation information specifies locations in the object which refer to global symbols and describe how they should be updated if the symbol is relocated to another address in memory. This information is used by the linker when object files are linked into binaries, but is usually not included in binaries because it is not necessary. Fortunately the GNU linker can be invoked with a flag that allows relocation information to be retained in binaries. The information is then used by the loader.

Ultimately, I settled on using load-time relocation. Performing relocations is simpler than updating the GOT because it can be performed in C++ code by the kernel rather than ARM assembly by the user program before its C++ runtime environment has finished initializing. Additionally, accessing symbols directly rather than indirectly via a GOT leads to the user program using more efficient and easier to understand and debug. GOTs have other advantages when using shared libraries, but this is outside the scope of this project[7].

Performing relocations is relatively simple. A segment is reserved within the binary file which contains a table of metadata. Each entry in this table contains a pointer to a location with the binary along with an integer code which describes the operation that needs to be performed when the program is relocated[8]. The ARM ABI defines over a hundred different relocation codes, but only one was implemented as only one was ever emitted by the compiler.

The final step is to actually have the CPU execute the program. The most basic way to accomplish this is to simply set the program counter to the entrypoint of the executable with a branch instruction. However, this overwrites the kernel's execution context, which is not desirable. Instead, a "context" structure is created for both the kernel and the process which records the register states. When the process is loaded. When the process is loaded, this context is initialized with the appropriate stack pointer and program counter. A routine was written that saves the CPU register state to the kernel context structure and loads it from the process context structure. At the same time, the routine switches the CPU mode from System to User. This switch ensures that the CPU cannot perform certain operations which only the kernel should have access to, like modifying the system control coprocessor.

Because this requires directly manipulating registers in a precise manner, ordinary C++ code which does not expose the details of registers cannot be used. The setjmp and longjmp functions[12] provided by the C++ standard library can be used to save and restore register contexts to jump between them, but the details of the register buffer are not exposed outside of the library. This makes it impossible to set up the initial state of the process context. Additionally it is not specified whether the CPU mode is switched by this function and there is no method to manually do so. Therefore it is necessary to write this routine in ARM assembly.

3.5 System Calls

As user processes should not be able to use hardware resources themselves, there must be a mechanism for user processes to communicate with the kernel to request certain behaviour. These are broadly known as system calls. The hardware therefore needs to provide some mechanism to call prespecified code in a privileged execution mode from an unprivileged execution mode in a manner that does not allow the process to execute itself with those privileges. On ARM CPUs this is accomplished with a dedicated instruction, SWI[4]. Using the SWI instruction invokes an SWI handler routine in a privileged execution mode, which is installed by the kernel. The SWI handler stores the process' state, restores the kernel's stack, checks the validity of the call and executes it before returning to the user process.

A dedicated system call handler was written for this purpose. This was quite difficult, as it required the the process' CPU register state to be preserved and later restore in a fashion similar to the . Any mistake in this procedure will cause the process to behave unpredictably, with the nature of the register corruption difficult to determine from the behaviour. Use of DeSmuMe to precisely inspect the register state during every step of execution of the SVC handler was essential in the debugging of this routine.

System calls are not usually called directly by programmers. On Linux the syscall function can be used to issue system calls by providing the integer index of the syscall and any additional arguments[10]. However, this interface is not safe. The compiler cannot check that the supplied arguments are of the correct type for the given system

[4]This stands for Software Interrupt. This is a synonym for SVC or Supervisor Call.

call. The C standard library therefore provides a number of wrappers around syscalls which are more convenient and safer. However, it is inconvenient and time consuming to write these wrappers. It would be much more convenient if safe interfaces could be utilized with automatically generated wrappers.

This kernel provides a type safe interface to issue arbitrary system calls directly, with automatically generated wrappers. Firstly, a number of functions which should be exposed via system calls are declared in the kernel's source. Then, a C++ template is used to place the exposed function calls in a list and another C++ template is used to place pointers to these functions in an array. The SWI handler uses this array to select the function to call when a user program executes an SWI.

Another C++ template is used by user programs to issue SWIs. A function template is written which takes the name of one of the kernel functions as an argument and in turn generates a wrapper function which executes the appropriate SWI instruction. Importantly, the automatically generated function used in user code has the same signature as the implementation function defined in the kernel. This means that trying to call one of these wrappers with incorrect arguments will result in a compile time type error.

```
// Shared code enumerating the functions available with system calls.
using SwiList = TypeTraits::ConstantList
<
    &exitFromNelf,
    &usrDbg,
    &getKeys,
    &yield,
    &spawnThread,
    &msgSend,
    &msgRcv,
    &getFrameCtr,
    &acquireFrameBuffer,
    &releaseFrameBuffer,
    &launchExecutable
>;

// Kernel code placing pointers to these functions in an array
template <size_t, typename>
class SwiTable;

template <size_t sz, auto ...constants>
class SwiTable<sz, TypeTraits::List<TypeTraits::Constant<constants>...>>
{
    void (*pointers[sz])() = { ((void(*)()) constants)... };
};

constexpr extern
SwiTable<0x80, SwiList> swi_table{};

// User code calling a system call with an automatically generated wrapper.
swiCall<usrDbg>("Hello from userspace!");
```

This mechanism makes it safer to invoke system calls and makes it trivial to introduce new system calls. All that needs to be done is to add another function to the definition of SwiList, and then the function can immediately used with an automatically generated wrapper in user code.

3.6 Multitasking

For the kernel to multitask, it must maintain contexts for multiple executing processes simultaneously. These contexts include information such as the CPU register state, the memory permission tables and the ID of the process. This data is stored in a Process Control Block, or a PCB. The PCBs are stored in a data structure known as the process table. As efficiency is not a priority for this kernel, a general purpose allocator (std::allocator) and data structure (std::set) provided by the C++ standard library were selected to form the process table. It was a simple task to adapt the existing kernel and process register contexts into PCBs in a process table.

A process scheduler must also be written. The scheduler is a procedure which determines when processes should be executed. One again, the aim is to prioritise functionality over performance - so the simplest scheduling algorithm was chosen, a round robin algorithm. This essentially means that all processes are allocated are placed in a list and executed sequentially, returning to execute the first process again after the final process is finished executing.

To implement this, there must be an ordering among the processes. The chosen data structure for the process table already includes an ordering that makes it possible to iterate over the PCBs within it. However, this will iterate over all PCBs within the process table. This is undesirable, as it is preferable to ignore processes which do not need to be executed. Processes are therefore considered to inhabit three states - running (currently being executed), ready (can be executed) and blocked (should not be executed). Two intrusive doubly linked lists called execution lists are maintained between PCBs, for ready and blocked processes. As each PCB can only inhabit one of the lists at a time, each PCB only needs to contain one pair of intrusive pointers. The scheduler can now simply traverse the ready list to find all the processes which should be scheduled for execution, and schedule them in order of traversal. The list is doubly linked so that list consistency can be maintained when removing elements from the middle of the list. In other words, processes can be moved between the ready and blocked lists at any time as necessary. However, this functionality was never utilized and events that would cause a thread to block or unblock were never implemented.

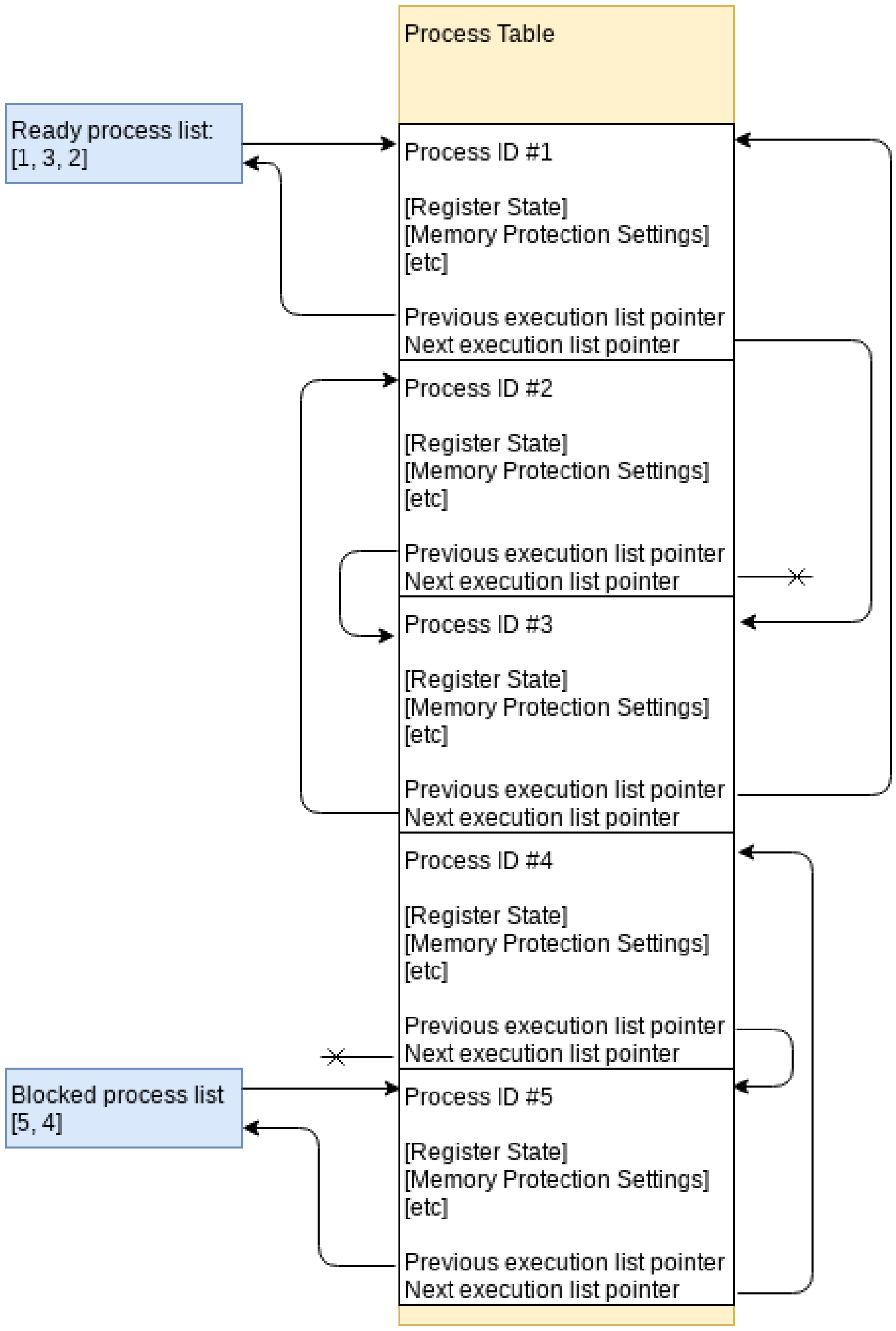
Process Table
Ready process list: [1, 3, 2]
Process ID #1
[Register State]
[Memory Protection Settings]
[etc]
Previous execution list pointer
Next execution list pointer
Process ID #2
[Register State]
[Memory Protection Settings]
[etc]
Previous execution list pointer
Next execution list pointer
Process ID #3
[Register State]
[Memory Protection Settings]
[etc]
Previous execution list pointer
Next execution list pointer
Process ID #4
[Register State]
[Memory Protection Settings]
[etc]
Previous execution list pointer
Next execution list pointer
Blocked process list [5, 4]
Process ID #5
[Register State]
[Memory Protection Settings]
[etc]
Previous execution list pointer
Next execution list pointer

Figure 3.3: Diagram of process table and execution lists

There also needs to be a mechanism for creating new processes to execute. Two were selected. The first is a method of spawning threads. Similarly to Linux's obsolete[5] threading model, this kernel simply implements threads as distinct processes which share certain amounts of context like the memory protection tables. This makes it very easy to implement threading, reusing all the same code that is used to implement multiprocessing. To spawn a new thread, an existing thread need only use the spawnThread system call. The declaration of this function is as follows.

```
Pid spawnThread(u16 (*start)(u32 closure, Pid parent, Pid self), u32 closure);
```

There are two arguments to this function. The first is a pointer to a function which takes three arguments - an integer "closure" argument, the process id of the parent thread, and the process id of the newly spawned thread. This function is called when the thread first runs. The second is the closure, which will be passed to the start function when it is called. The closure may be converted to a pointer and dereferenced so the start function can retrieve arbitrary data of any size[6]. This is very similar to the pthread_create function used by POSIX threads[9].

For each thread to run the chosen function, the four arguments (start address, closure, parent process ID and own process ID) are loaded into the thread's registers before execution just like the program counter and stack pointer are. The program entrypoint of _nelf_start then simply calls the function with those arguments. If a null address is passed to the _nelf_start routine, such as when the process is initially loaded by the kernel, then the process simply calls nelf_main instead with the remaining arguments instead. Indeed, the nelf_main function has the same signature as a thread start function. This technique keeps the _nelf_start routine simple and concise, and this is the main motivation for using a divergent declaration for the main function from an ordinary C++ environment.

Beside threading, there is a system call for making new processes. This function simply takes a filename as an argument, which refers to a binary file in the NitroFS filesystem. The kernel then loads and executes the program in exactly the same way as the initial program was executed.

The simplest mechanism for multitasking is co-operative multitasking. In this system, a process executes a yield system call. This system call suspends execution of the current process by returning to the process scheduler. This is accomplished by simply loading the register state from the kernel scheduler context from within the system call function. The scheduler can then simply choose another process to execute. This is very simple to implement, but has some limitations. If a misbehaving process never yields, then no other process is able to execute.

To solve this issue, preemption is used. Preemptive multitasking interrupts allocates a period of time for processes to execute and interrupts them when that time period ends. This is accomplished using a hardware timer. When the process is scheduled to run, the timer begins to count down. When the countdown concludes, an interrupt is issued to

[5]This implementation was replaced because it does not comply with the POSIX specification. However this kernel has no interest in complying with this specification.

[6]This technique can be compared to the use of void pointers and function pointers to imitate lexical closures in C and C++, hence the name.

the processor, and a kernel function known as the IRQ handler is called. Similar to the SWI handler, this function saves the process's state before executing the yield function.

There were a number of unexpected issues implementing preemption, however. If an interrupt is received while a context switch is in progress, it is possible for registers to be corrupted. It's therefore necessary to disable interrupts when performing sensitive actions like this. A number of bugs related to this were uncovered in the context switching code that existed that needed to be fixed before preemption worked correctly.

3.7 Inter-Process Communication

While threads are able to communicate quite easily by sharing a memory address space, processes deliberately cannot. All communication must be done indirectly, via system calls. The initial mechanism chosen was a signalling mechanism based on POSIX signals[11], which interrupt a running process and call a handler function to respond to the interrupt before the previous operation is resumed.[7] However, this proved to be surprisingly difficult to implement. A signal handler can also issue signals, which means signal handlers can be nested arbitrarily deeply. This means an unbounded number of contexts need to be saved for each thread. By eschewing signals, only one context per thread plus one context for the kernel scheduler needs to be saved. The functionality that would have been implemented by signals was therefore split between three collectively simpler mechanisms.

1. Small, infrequent data is shared via message queues.
2. Large, frequent data is shared via shared memory.
3. Interrupting data is shared via spawning a new thread and blocking the old one until execution completes.

A message queue is a list of messages, which are arbitrary-sized sequences of bytes sent from one process to another. Any process can send a message to the message queue of any other process by using the msgSend system call, and a process can read the contents of a message in its queue by using the msgRcv system call. Messages are read in a First-In, First-Out (FIFO) order, hence the name of a queue.

There exists a singly linked list in the kernel's address space for each process which implements the message queue. When a message is sent, a message structure is allocated in the kernel's address space and the data is copied into the structure from the process's address space. The structure is then inserted at the tail of the linked list. When the message is read, the data is copied from the message structure into the address space of the receiving process and the message structure is deallocated.. In this way, data is shared indirectly and safely via copying it in and out of the address space of each process by mutual agreement. This can be quite inefficient and awkward to use if the data to be copied is large, or if it needs to be done often.

[7]This can be viewed as an analogue to hardware interrupts.

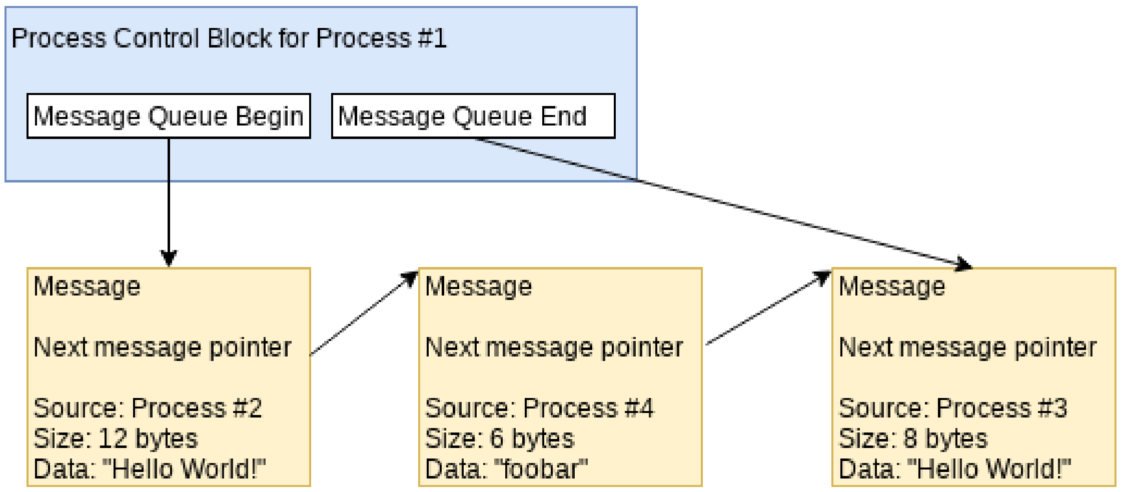

Figure 3.4: Implementation of the linked list data structure for the message queue.

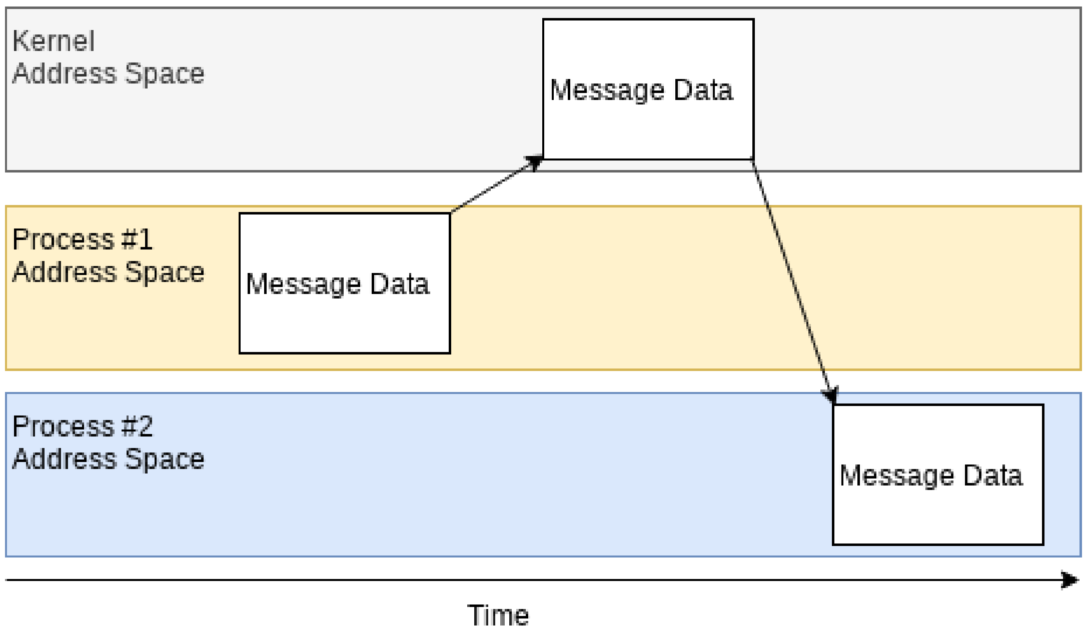

Figure 3.5: Illustration of how message data is copied between address spaces.

The design for shared memory was simple. A process which requests a dynamically allocated block of memory from the kernel is designated its owner. The owner of the memory block can then use a system call to change the permissions of another process to access this memory block - including read, write, execute and ownership permissions. This method has a significant initial overhead, and requires another IPC method like messages to inform the other process when and how shared memory is initialized. However it is extremely cheap and convenient for sharing large blocks of data frequently as they can simply be modified directly without the need to use system calls after set up. A special advantage of this method is that, due to the lack of virtual memory, the shared memory block is at the same address in the address spaces of both processes. This means complex data structures which use pointers will remain valid from the perspective of both processes. However, neither dynamic memory allocation nor permission modifying system calls were added as time ran out before any programs could be written which would use them.

Both messages and shared memory require are only polled during the course of a program's normal execution. In other circumstances, it is preferred for ordinary execution to be suspended for a handler to be executed before the previous activity resumes. An example for this would be a memory access violation. On a Linux system, the SIGSEGV signal would be sent to the process in this circumstance[11]. However, this kernel does not use signals. Instead, it was decided that a new thread should be spawned when an event like this occurs and that the previous thread should block as long as the new thread exists. The behaviour of this is very similar to that of a signal, as it suspends a process to execute a handler then resumes the process when the handler concludes. However, it is much simpler to implement as it reuses the existing process model and does not require execution contexts to be nested. A number of issues were identified with this method though. The overhead is very high, requiring a new process control block and call stack to be created every time interrupting data is received. This is expensive to do and may not always be possible if the device runs out of memory. This can be solved by using preallocation and caching to reserve these resources ahead of time. It is not memory efficient either, as the creation of a new call stack requires a minimum of 4KB of memory to be reserved. However as interrupting data is presumed to be infrequent this should not be an issue. The design and implementation of this mechanism were never completed due to time limitations, and in the final kernel any circumstance which would have used this mechanism simply terminates the process.

3.8 Memory Protection

Memory protection is the control of access rights to different regions of memory by different processes. Memory protection ensures that one process cannot interfere with another process except by controlled inter-process communication. Memory protection also assists debuggers and security, as it can cause a process to terminate when reading invalid memory rather than continuing to execute erroneously.

Outside of software emulation, memory protection can only be accomplished with some form of hardware assistance. This is ordinarily a memory management unit (MMU), which also implements virtual memory. The Nintendo DS possesses a memory protection unit (MPU) which is much simpler and less capable, primarily being intended to assist debugging rather than implement process isolation[3]. However, it is sufficient.

The memory protection unit defines eight memory regions of descending priority. Each region has a size, a number of bytes which is a power of 2 between 4KB and 4GB. Each region also has a base address defined the same way, which is at least as large as its size. Each region defines permissions for reading instructions and reading or writing data whether in user or kernel mode. If the memory protection unit is turned on, then any memory which does not belong to a region is not accessible.

Firstly, the size and base address restrictions are restrictive. A 4KB minimum size and alignment restriction is common for virtual memory pages on personal computers, but they have much more memory available than the Nintendo DS which only has 4MB. Fortunately the default memory allocator makes it convenient to allocate memory of a certain size and with a certain alignment.

Secondly, there are only eight memory regions that can be defined. Processes should have no limitation on the number of memory regions they have access to. The solution is to only use the memory protection unit as a cache. While a process can have any number of logical memory pages in its page table[8], only seven MPU memory regions are used by the process. A number of memory pages are loaded into the MPU regions. If the process tries to access memory which is not protected by an MPU region, a CPU abort is triggered and an abort handler is executed. The abort handler is now able to traverse the page table and determine if the address which triggered the abort belongs to a page. If it did, then one page is evicted from an MPU region and the new page is loaded into it instead. Process execution can then be resumed as though it were never aborted. If it did not, then the process truly did commit an error and this error can be handled. The MPU is therefore used analogously to the translation lookaside buffer (TLB) of an MMU, as a cache to store frequently used entries of a much larger page table. However, walking the page table in the case of an MPU miss is not assisted by hardware as it is in many MMU implementations.

The permissions of a page are initialized when the page is added to the page table of a process. For example, the stack of a process is readable and writeable but not executable. On the other hand, the code-containing segment of a process is readable and executable but not writeable. This adds an extra degree of convenience for programmers as it will cause the memory protection error handlers to be called rather than for program execution to continue in error. However the ability for programs to define handlers for memory protection failures were not implemented in time. System calls to modify permissions, including permissions for a particular page granted to other processes, were planned but never implemented.

In accordance with the project's aims of simplicity and functionality above performance, a simple round robin algorithm was chosen to select which page should be evicted from a region for a new page to be loaded. However, the possibility of implementing more complex algorithms does not seem to exist. For example, a least recently used (LRU) algorithm cannot be implemented as there is no way to record when a page is accessed.

On the ARM CPU, there are two kinds of abort[13]. The first is a prefetch abort, which is triggered when the CPU tries to load an instruction from memory it does not

[8]These pages do not use virtual addresses as in a traditional page table, but the name is chosen here to distinguish them from memory regions in the MPU.

have permission to use. The second is a data abort, which is triggered when the CPU tries to load or store data from memory it does not have permission to use. In both cases, the handler function only has access to the address of the instruction that triggered the abort. In the case of a prefetch abort, this address can be used to find the page to load. In the case of a data abort, the instruction this address points to must be decoded using software to determine what data was actually attempted to be accessed. The ARM instruction set supports a variety of different instructions which access memory, all with different encodings. Only those instructions which appeared in programs that were run on the kernel were added to the decoding routine, with more being added as more programs were written.

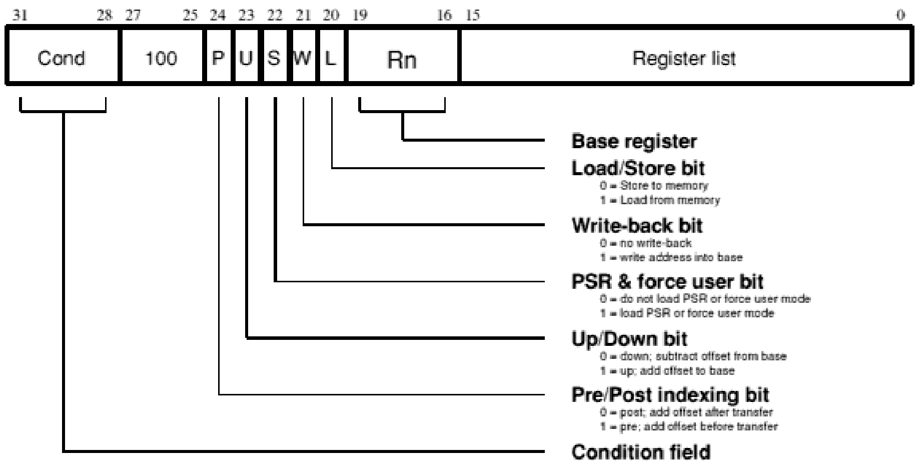
31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Condition field

Figure 3.6: A diagram of the binary representation of an ARM LDM/STM instruction.[14] This is one form of instruction which can be decoded by the data abort handler.

One may observe that the ability to run arbitrary abort handlers on memory accesses make it technically possible to implement a software implementation of virtual memory without resorting to full emulation. The MPU may be configured so the abort handlers trigger on every memory access, translating virtual addresses to physical ones with software routines. This technique can be compared to GBARunner2[15], another piece of software written for the Nintendo DS which executes software written for the closely related but incompatible Nintendo GBA platform. GBARunner2 uses abort handlers in this manner, translating addresses where the memory mapping differs between the two platforms. However, the abort handlers are so slow that they are the main bottleneck in the software despite being heavily optimised. Using abort handlers to perform software translation for all addresses, or any significant number of addresses, to implement virtual memory is technically possible but utterly infeasible due to the overwhelming overhead.

3.9 Input/Output

A number of additional input/output system calls needs to be available for programs to do anything useful. The most noteworthy are as follows.

- Read key inputs.
- Display framebuffer graphics.
- Read a frame counter.
- Print debug messages.

Most of these are rather thin wrappers over functions provided by libnds. In particular, the graphical output is much simpler than it could have been. The Nintendo DS has complex and fast hardware acceleration for graphics. However, exposing this hardware via a system call API would be much too complicated. Instead, only a simple software solution can be used - with processes being able to request access to a 96KB block of memory containing a framebuffer and writing 16 bit colours into it using the CPU to display on the screen. This is far less efficient than using hardware acceleration, but for the sake of illustrating a user program that can successfully display graphics it requires less support from the kernel and less code in the program.

The use of a frame counter is intended to be used to be used to time a program. A program can yield execution until the counter advances, which occurs 60 times per second. This ensures that a program which runs continuously like an ongoing simulation can run at a perceivably constant rate. An alternative which would cause a process to block until the frame counter advances was planned instead, but this approach was chosen because it was simpler and faster to implement.

4 Evaluation

The nature of a kernel is that it is very difficult to see its behaviour in and of itself. Its role is to provide a platform for other programs to run. The ideal way to test a kernel therefore is to create programs that rely on its functionality to work. As each feature was added to the kernel, a new program was written that utilizes it. A number of key programs are described ahead.

One of the first programs to be written tested position independence. This was accomplished by declaring global variables in a number of different source files and passing pointers to each through a number of different trivial functions. The program is written in such a way that all global variables must be used correctly or the program will crash. By declaring the variables in distinct source files, it is ensured that even with compiler optimizations enabled the global variables are not optimized away. By passing pointers between functions declared in different source files, predictable assembly code will be generated which will utilize pointers to those global variables in a manner that is easy to monitor using a debugger. Using a debugger to watch those pointers makes it easy to see whether or not they have been correctly relocated. This program was crucial for ensuring the correct function of the relocator in the program loader. After this test, there were no bugs.

The next program tested system calls. This program used handwritten assembly to initialize the CPU registers with a predictable pattern before issuing a system call. Using a debugger, the state of the CPU registers after returning from the system call is inspected. This makes it possible to monitor the program's state to ensure the context switch completed successfully. Other mechanisms which induce context switches like interrupts and aborts are tested in a similar manner. The context switching code was very fragile and there were a number of regressions when adding features such as memory protection, so these programs were used frequently to ensure the behaviour was still correct.

Another key program tested both co-operative multitasking, threading, messaging, and memory protection. This program creates 12 new threads which send messages to the main thread, while the main thread waits until all messages are received. Yielding a number of times ensures that the process scheduler is working correctly, and a number of bugs were uncovered using this method. Messaging is tested as the program only makes progress if the correct messages are sent. Memory protection is tested as using new threads means new memory pages corresponding to the program stack must be loaded into the memory protection unit, and using 12 threads requires that pages are correctly being loaded and unloaded into the memory protection unit. Because this program is testing memory protection, it must be executed on real hardware rather than an emulator with a debugger. The only reliable way to get debug output on real hardware is to print debug messages to a console on the screen. Due to the low resolution of the screen, the debug messages were carefully selected depending on what particularly was being tested at the moment, as otherwise critical information may have been scrolled off the screen at the time of the crash. This program was also used to test preemptive multitasking. By utilizing a compile time flag to turn the yield system call into a no-op, all programs which test co-operative multitasking will instead test preemptive multitasking.

The final program to be written was by far the most complex. One of the aims for the major aims for this program was for it to be interactive, so it can be used by individuals who are not familiar with the project. The behaviour of the program should demonstrate the functionality provided by the operating system. Development of the program should also take up little enough time that it does not interfere with the development of the kernel. It was decided that this program should be a simulation of Conway's Game of Life with multiple concurrent instances. This is a cellular automation that periodically updates. This allows an easy to comprehend demonstration of button inputs, graphical output, concurrency, interprocess communication and real time interaction. As a long-lived and interactive program whose inputs change from run to run, this can also be used to gauge the stability of the kernel.

5 Reflection

A great deal has been learned from the development of this project. Certain features were deceptively simple to implement, while others were beset by unexpected bugs. Some features were implemented precisely as planned, some were revised, and others were scrapped entirely. Most features that made it into the kernel were implemented as expected. Examples of this include

- System calls
- Co-operative multitasking
- Preemptive multitasking
- Memory protection

An example of a small feature which was changed during the development process was the implementation of position independence. It was presumed that all that needed to be done to accomplish this was to set a compilation flag when building user programs. This turned out not to be true, and it required an unexpected degree of additional work to implement. The decision was made to use load time relocation instead as it was simpler, and this change had very few consequent effects on the development of the kernel.

An example of a larger change was the move away from signals for IPC. The implementation complexities of signals were not considered when the plan was originally devised. This was a lapse in forethought, as all the information necessary to understand why signals are so difficult to implement was available at the planning time. However, the decision to use an incremental development process showed itself to be judicious as it enabled the switch to a different approach without having to revert any existing work. The revised approach turned out to be significantly easier to implement. However even that approach was only partially implemented, with shared memory and thread-spawning interrupts not being completely implemented. While this demonstrates the design was overly ambitious, it also shows how important it was that an incremental development model was used, as it demonstrated how a partial but functional solution could be made.

One of the features of particular interest was memory protection, as this is a situation where the target hardware differs substantially from other platforms where memory protection is implemented. Because of its centrality to the aims of the project, the viability of this feature was established early by analysing the capabilities of the available hardware. It remained surprising how much of the desired functionality was retained - with complete process isolation possible, but also providing support for relatively modern features like stack protection. The implementation of arbitrarily sized page tables with the MPU functioning as a cache was surprisingly similar to that of a TLB in a more conventional computer. One may assume then that the more complex hardware available in other computers is pointless. However the use of a basic MPU still makes the implementation of other desirable features impossible. Virtual memory permits memory paging, which enables a process to use much more memory than is physically available by swapping memory pages between RAM and storage on demand. Simple memory protection is not capable of supporting this feature, as it does not dissociate virtual memory addresses from the physical memory which backs it.[1]

The lack of any special graphics API is disappointing. The Nintendo DS has very fast and relatively simple graphics acceleration hardware, and a system call API which allows multiple processes to utilize it simultaneously would be an interesting technical accomplishment due to the hardware being intended to be used by only a single process. However the interface is still much more complicated than the chosen framebuffer API. Design and development of a hardware accelerated API was much too ambitious for this project considering the concessions that were already made during development and such an API would never have been practical to implement within the given time constraints. It was prudent not to include this in the project plan.

The decision to make the project structure flexible was wisely made. The time allocated for some tasks like getting a basic executable running on the system was overestimated, while time allocated to implement memory protection was heavily underestimated. Development of this feature was inhibited by debugging difficulty. This may have been alleviated if specialized development hardware were available, but this would have been too expensive. A number of features were cut, which is a disappointment, but these features were largely elaborations on existing features rather than being entirely new technical features. Cutting these features is not so harmful because their absence does not contradict the goals of this project, which is to demonstrate the viability of features rather than provide them in a way that is polished enough for regular use.

The notion of an operating system in general use encompasses more than just a kernel. Operating systems usually provide a large number of system software that exists outside of the kernel to implement useful services, such as a user interface. Nothing of the sort was written for this project, as it is not in question whether such software would be possible to write. The goal was to demonstrate the possibility of kernel features, not the operating system features that can be built upon them.

In spite of all the changes made to the project plan, the core aims of the project were always respected and ultimately fulfilled. A minimal operating system kernel demonstrating the principles of an operating system including multitasking, process isolation and hardware abstraction were successfully developed. The kernel succeeds as a proof of concept for the viability of these concepts on a hardware platform which does not traditionally use them.

[1]If the hardware uses memory mirroring, it may in fact be possible to give processes access to address a greater pool of memory than physically exists. However the Nintendo DS does not mirror memory in a way that would make this technique viable and performance is limited even in cases where it is viable.

Bibliography

[1] Andrew S. Tanenbaum, Albert S. Woodhull.
Operating Systems, Design and Implementation
Prentice Hall, 1997.

[2] DSLinux
http://www.dslinux.org/
11th April 2019

[3] GBATek - Gameboy Advance / Nintendo DS / DSi - Technical Info
https://problemkaputt.de/gbatek.htm
24th March 2015, visited 15th April 2019

[4] devkitPro - Portal
https://devkitpro.org/
16th April 2019, visitied 16th April 2019

[5] No$gba Gameboy Advance / Nintendo DS / DSi Emulator Homepage
https://problemkaputt.de/gba.htm
23rd Februrary 2019, visited 15th April 2019

[6] DeSmuMe: Nintendo DS emulator
http://desmume.org/
11th March 2019, visited 17th April 2019

[7] Relocatable vs. Position-Independent Code (or, Virtual Memory isn't Just For Swap)
http://davidad.github.io/blog/2014/02/19/relocatable-vs-position-independent-code-or/
19th Februrary 2014, visited 20th April 2019

[8] ELF for the ARM Architecture
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf
24th November 2015, visited 20th April 2019

[9] PTHREAD_CREATE(3) - Linux Programmer's Manual
http://man7.org/linux/man-pages/man3/pthread_create.3.html
23rd April 2019, visited 24th April 2019

[10] SYSCALL(2) - Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/syscall.2.html
23rd April 2019, visited 24th April 2019

[11] SIGNAL(7) - Linux Programmer's Manual
http://man7.org/linux/man-pages/man7/signal.7.html
23rd April 2019, visited 24th April 2019

[12] SETJMP(3) Linux Programmer's Manual
http://man7.org/linux/man-pages/man3/setjmp.3.html
23rd April 2019, visited 24th April 2019

[13] ARM946E-S Technical Reference Manual
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0201d/DDI0201D_arm946es_r1p1_trm.pdf
26th April 2019, visited 20th April 2019

[14] ARM Architecture Reference Manual
https://www.scss.tcd.ie/ waldroj/3d1/arm_arm.pdf
26th April 2019, visited 28th April 2019

[15] GBARunner2 repository
https://github.com/Gericom/GBARunner2
14th April 2019, visited 19th April 2019