



THE UNIVERSITY OF QUEENSLAND

Towards a POSIX userland for Ulysses

by
Sam Kingston
41196616

School of Information Technology and Electrical Engineering,
The University of Queensland

June 2009

June 12, 2009

Head
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia QLD 4072

Dear Professor Bailes,

I present the following thesis entitled:

Towards a POSIX userland for Ulysses.

This work was performed under the supervision of Dr John Williams for the course *COMP3001: Special Topics in Computer Science 3B*.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Sam Kingston
41196616

Acknowledgements

I would like to thank the assistance of my supervisor, Dr John Williams. His invaluable input and guidance into this project and the preparation of this document is greatly appreciated. Without the freedom and support he has given this project would never have been completed.

I would also like to acknowledge the assistance of David Belvedere and Dan Callaghan for their help in proofreading this document.

Abstract

Operating systems can be divided into two major sections: the kernel, and the userland. An operating system without a userland cannot run user applications—a vital component in making the system one that users can utilise. In this thesis, research is made into the various implementations of userland, and this research is used as a basis for the design and implementation of a working userland for Ulysses. Following this, the quantitative and qualitative results of testing the various subsystems of the userland are given. These results prove that the implementation is correct and works as expected. Finally, a rough roadmap for future work to overcome some of the limitations of the implementation is given at the end.

Contents

Acknowledgements	iii
Abstract	v
1 Introduction	1
1.1 The job of an operating system	1
1.2 Userland—why do we need it?	1
1.3 Ulysses	2
1.4 Outline of this report	2
2 Background and research	3
2.1 General operating system features	3
2.1.1 Task management	3
2.1.2 Context switching	4
2.1.3 Kernel threads	5
2.1.4 Interrupts/exceptions	5
2.1.5 Privilege separation	6
2.1.6 System calls	8
2.2 POSIX	8
2.3 Minix 3	9
2.3.1 Kernel threads	9
2.3.2 Software interrupts	9
2.3.3 System calls and message passing	9
2.3.4 Scheduler	10
2.3.5 POSIX-compatible	10
2.4 Linux	10
2.4.1 Kernel threads	11
2.4.2 Software context switching	11
2.4.3 System calls	11
2.4.4 Scheduler	12
2.4.5 POSIX-compatible	12
2.4.6 Dynamic timer ticks	12
3 Implementation and Methodology	13
3.1 Anatomy of a task	13

3.2	ps shell command	14
3.3	Task creation	14
3.3.1	Default stack	15
3.3.2	task_t structure	15
3.3.3	Creating the task	17
3.3.4	Cloning an existing task	17
3.4	Kernel threads	17
3.4.1	kthread_create	18
3.4.2	kthread_yield	18
3.4.3	kthread_exit	18
3.5	Context switching	19
3.5.1	Selecting a new task	19
3.5.2	Software context switch	19
3.6	Privilege switching/user threads	21
3.7	System calls	21
4	Results	23
4.1	Unit testing	23
4.1.1	Scheduler	23
4.1.2	Kernel threads	25
4.2	<i>In situ</i> testing	25
4.2.1	Round-robin scheduling	25
4.2.2	System calls	26
4.2.3	Context switching	27
4.2.4	Execution of Hello, World! task	29
4.3	Subsystem breakdown	29
5	Conclusion	31
6	Future work	33
6.1	Priority-based scheduler	33
6.2	Kernel thread mutexes	33
6.3	Smarter preemption model	34
6.4	ELF binary loader	34
	References	35

A	Code listings	37
A.1	test/run_tests.sh: Test script	37
A.2	diffstat output over project lifetime	39

List of Figures

1	Privilege rings for x86	7
2	Anatomy of a simple Hello, world! task in Ulysses	13
3	ps output	14
4	task_t structures in a singly linked list	16
5	Scheduler activity diagram	20
6	Timeline and process of a system call in Ulysses	22
7	A user task function with a valid system call	26
8	Code for context switching test	28
9	Example of giant kernel lock as a mutex	33

List of Tables

1	Default stack required for each task (from top to bottom)	15
2	Analysis of context switching kernel threads	29

1 Introduction

Currently, Ulysses, a tiny monolithic operating system, does not have the ability to manage or run user applications; a vital component for a usable operating system to have. This project aims to investigate and implement a working userland for Ulysses, giving it the ability to run user applications as other operating systems can.

1.1 The job of an operating system

An operating system exists to provide an interface between the hardware of a machine and the software running on the machine. The operating system controls the hardware directly, and is responsible for managing and controlling the running of user software (tasks). It provides an abstraction from the hardware, allowing the user to focus on performing the task at hand, rather than worrying about the low-level hardware details.

One key function of an operating system is the appearance of concurrency. Since a Central Processing Unit (CPU) can only execute one thing at a time, there is no way to have true concurrency unless the system has multiple CPUs. Instead, the appearance of multiple tasks running at once is achieved through multitasking—running multiple tasks in quick succession.

1.2 Userland—why do we need it?

If user programs were allowed full access to the hardware of a machine, it is possible that a program which is poorly coded or malicious in nature could bring down the entire machine by causing a fault to occur. This would render the machine unusable for other users. In this case, the operating system kernel would not be able to prevent the user program from crashing the machine.

To counter this, an operating system runs user programs in userland—a sandboxed environment that prevents each program from, amongst other things:

- Directly accessing hardware;
- Tampering with other programs in memory; and
- Using all of the available CPU time, therefore starving other tasks.

With assistance from various hardware protection measures, a kernel can “lock out” certain features and memory addresses from different programs, ensuring that a user program cannot damage other programs or bring the system down.

1.3 Ulysses

Ulysses is a Unix¹-like operating system targeting the Intel x86 architecture, written and maintained by the author. It is written primarily in C and Intel-style assembly language. Ulysses is released under the GPL version 3 license. A complete code listing of the Mercurial repository is available at <http://source.sjkwi.com.au/hg/ulysses/file/446833fc1392>.²

At the beginning of this project, Ulysses included support to initialise the low-level x86 hardware, and included a simple command interpreter.³ It did not have a task manager, and did not have the ability to even load a single task and run it.

1.4 Outline of this report

This report is divided into six chapters, each building on the last. Research was undertaken into the requirements for a userland, as well as the implementation of userland in other operating systems. The results of this research are documented in chapter 2 and form a guide to, and influence the design decisions described in, the rest of the document.

The design and implementation of a userland in Ulysses is documented in chapter 3. This chapter purely deals with the design decisions and methodology used with Ulysses, drawing from the research in the previous chapter. It is divided into several sections that describe each major piece of functionality which was necessary to achieve the aim of the project.

The results of testing the implementation from chapter 3 are included in chapter 4, as well as an analysis of these results. Conclusions are then given, and suggestions for future work to overcome some of the limitations of Ulysses' userland are discussed in chapter 6.

All timing and test results presented in this document were performed on Linux 2.6.29, running on an Intel Core2 Quad CPU clocked at 2.40 GHz with 4 GB of physical memory. *In situ* tests were run inside the QEMU 0.10.4 machine emulator on the above-stated system.

Partial code listings of relevant modules are given in appendix A.

¹Officially trademarked as UNIX.

²Mercurial is a distributed version control system, similar to Git, which is favoured by the Linux kernel developers.

³ The interpreter is known internally as the *kernel shell*—a misnomer since it actually runs *inside* the kernel, rather than providing access *to* it.

2 Background and research

In this chapter, the results of researching the components of userland are presented, as well as a brief overview of the design of each component in two existing Unix-based operating systems: Linux, and Minix 3. This research will help influence the design decisions and implementation presented in the following chapter.

2.1 General operating system features

The following sections detail the components and subsystems of an operating system kernel that are required to provide and support a userland.

2.1.1 Task management

A “task” is defined as either a *process* with its own address space, or a *thread* of execution running *inside* a process. The operating system needs a way of managing these tasks, since the CPU can only run one at any given time. This task management can be broken down into three broad categories:

Task creation Housekeeping needs to be performed to create a new task, or to clone an existing task. Either way a new task structure needs to be created for the operating system to keep track of the task, as well as a default stack and virtual memory mappings (on x86 this is achieved via a page directory) set up for the task. This is important since it allows the CPU to first start executing the loaded task.

When creating a new process, a new address space needs to be created. Unix-like systems typically do this by cloning the current address space. Threads do not need a new address space however, since they share their address space with the process they are executing in (their parent).

Since every task has a unique process identifier (pid) *for the duration of its execution*, a new pid must be generated upon creation. This cannot be reused until the task is destroyed.

Scheduling Once a task is created, the operating system needs some way of executing it. This is performed by the scheduler, which keeps track of all tasks on the system. It is responsible for deciding when the current task should be preempted and a new one picked, ensuring that all tasks are given an ample time slice to run on the available CPU(s). The operating system scheduler is a key component in providing multitasking support—giving the appearance of more than one task executing simultaneously.

There are many different scheduling algorithms that each have their own advantages and disadvantages. These will be discussed in detail under the specific operating system research in sections 2.4 and 2.3. Some common algorithms include:

Round-robin Assigns time slices to tasks equally in a circular algorithm with no concept of priority.

Fair-share Assigns time slices so all users on the system are given equal time, irrespective of how many tasks they own.

Task destruction Once a task has finished executing, either because it voluntarily exited or the operating system forced it to exit, there needs to be a procedure for cleaning up a task so it no longer executes or holds any resources. This is typically done by a handler that removes the task from the scheduling queue(s) and cleans up any resources it held while executing (e.g. stack, page directory, registers). The exit handler is often set at the bottom of the task's stack upon creation—this makes the task call this handler to clean itself up once it returns and its stack unwinds.

2.1.2 Context switching

A context is the state of a task at a given point in time. It describes the address of the next instruction to execute, as well as the memory addresses of the task's stack, page directory, and CPU registers. This context is saved when a task's execution is interrupted, and restored by the task manager when the operating system wants to resume execution of the task. By keeping a context of every task on the system, it is possible to interleave the execution of various tasks. Since the context is restored before re-execution, the task is unaware its execution on the CPU was interrupted and later restored, giving it the appearance that it is the only task running on the system.

On Intel x86, there are two supported methods of performing this context switch, both of which roughly perform what is described above:

Hardware switching The x86 architecture does provide a mechanism to switch context in hardware, via the (re)loading of multiple data structures that describe the state of a task, called a Task State Segment (TSS). Each task on the system has a single TSS attached to it. The TSS stores information such as the stack pointers, page directory pointer, and CPU registers.

This method is rarely used however, as it is not portable between different architectures⁴, inflexible, and only marginally faster than performing the context switch in software.

⁴Since different architectures have different registers that need to be saved and restored, any context switch is unavoidably non-portable to a point.

Software switching Since most of the overhead of switching contexts exists in hardware, software switching is typically chosen since [1]:

- It is only marginally slower;
- The operating system can manage the switch, offering greater flexibility; and
- It is much easier to debug.

In the case of software switching, only one TSS needs to be loaded on the CPU: this contains the details needed for a privilege switch to a *higher* privilege level. This greatly simplifies the routines to create and switch tasks. Since the operating system itself is keeping track of each task's stack pointers and registers, the single TSS on the system does not need to have this information stored in it.

2.1.3 Kernel threads

A kernel thread is a lightweight form of process that is implemented in the operating system's kernel, giving it multiple threads of execution. It allows the kernel to run background tasks, typically separating different subsystems. They are lightweight since they do not have their own address space—merely sharing the same address space as the kernel. Kernel threads have a distinct advantage over processes since they have a much faster creation time, and can be context switched more efficiently. Unlike context switching a process, only the stack pointers and instruction pointer of the thread need to be loaded onto the CPU. A new page directory does not need to be loaded when switching between threads, as their address space is shared by all other threads running in the kernel.

2.1.4 Interrupts/exceptions

Interrupts and exceptions are two methods that various hardware systems use to notify the operating system that something has happened.

Interrupts These are used to notify the kernel that it needs to service some event that has occurred in a hardware device. Common interrupts include the system timer, keyboard events and data arriving on a serial port. They are sent by one of the Programmable Interrupt Controllers (PIC) present on the system. Interrupts must be acknowledged by the kernel before the PIC will send any more.

Software interrupts The interrupt vectors used by the system can be remapped to allow user-defined interrupts to exist. They are triggered in software, rather than by a hardware event. These user-defined interrupts are known as software interrupts and are a key concept in the typical implementation of system calls, described in section 2.1.6.

Exceptions The other class of notification events are exceptions. These are generated by the CPU itself when the task executing on it performs some invalid or incorrect function. Exceptions cannot be ignored, although most can be recovered from by removing the faulting task from the system. Intel describes any possible recovery from each exception as a “program state change” in their x86 architecture manuals [2].

Some examples of CPU exceptions are:

- Divide by zero;
- Page fault, where the task attempted to access a memory location not mapped in the current page directory;
- General protection fault, where the task attempted to execute an instruction that is not allowed in the CPU’s current ring.

Unlike interrupts, exceptions do not need to be acknowledged, and the CPU may send multiple exception at the same time. If this happens, the interrupts are queued until the operating system can process them all. Once the kernel returns from the first interrupt, the second will be sent, and so on.

In both cases, the CPU will interrupt the task executing, and switch the previously executing task with the kernel, transitioning back into the kernel to allow it to handle the interrupt or exception. On the x86 architecture, this is done by the CPU inspecting one of its descriptor tables to determine where to jump execution to.

2.1.5 Privilege separation

Since a task executing on the CPU has direct access to the machine’s hardware, with no software protection, it would be very easy for a rogue task to bring down the entire operating system. An example of this is the ability to execute instructions that can render the machine unusable, such as disabling interrupts or halting the CPU.

To prevent this, each task is run inside a sandboxed environment that stops it from directly accessing resources that it should not (and does not need) access to. The x86 architecture provides a mechanism for enforcing this sandbox environment called **privilege rings** (see figure 1). These

rings are labelled from 0 (most privilege) to 3 (least privilege) and are enforced in hardware by the CPU itself.

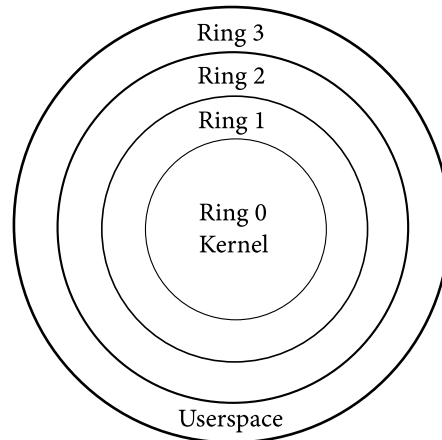


Figure 1: Privilege rings for x86

Anything running in ring 0 has full access to the CPU and all instructions on it. This includes access to privileged instructions, such as `sti` (set/enable interrupts), `cli` (clear/disable interrupts) and `hlt` (halt CPU).

Ring 3 is the ring of least privilege: privileged instructions are not allowed, and accessing the memory of other rings is prohibited. The CPU enforces this, generating a *General Protection Fault* exception if a process tries to perform an operation that is not allowed due to the lower privilege level.

Rings 1 and 2 are rarely used in the mainstream operating systems, however they can be used to maintain a higher level of control of some device drivers by running them in a higher ring than a userspace task, but less than the kernel itself.

On x86, switching from a ring of higher privilege to one of lower (e.g. ring 0 to ring 3) requires setting both the code and data segment selectors on the CPU to the new ring, and then returning to execution. The CPU will then switch privilege rings before continuing execution of the process. This is achieved through the exception return instruction, `IRET`.

There is no direct method of transitioning to a *higher* ring however, as the segment selectors on the CPU are locked out from all but ring 0. To overcome this, operating systems set the *Requested Privilege Level* (RPL) in the current task's TSS while still in ring 0. When the task is next interrupted, the CPU inspects the TSS' RPL and changes the privilege ring before executing an interrupt handler. This automatic transition back into ring 0 is seamless and requires no extra handling in the task or operating system.

2.1.6 System calls

Although user tasks now have no direct access to hardware devices or the operating system itself (since they are running at a lower privilege level), they still may require access to specific devices to perform tasks. An example of this is a user task writing to a network device. The task itself does not have direct access to the network device due to privilege separation, but still legitimately needs access to read and write data on it.

This problem is solved by allowing the user task to *request* the kernel (which is operating at a higher privilege level) to perform the service for it. The act of requesting a service in this fashion is called *making a system call*. The user task sends all of the required information to the kernel, which in turn performs the operation on behalf of the application, and returns control to the task.

Software interrupts are typically used to make the CPU switch back to the kernel and perform the system call request. On x86, this trap into the kernel is usually implemented by registering an interrupt handler on software interrupt vector 80 (INT 0x80). A user task pushes the system call number, as well as any arguments needed on to the stack, and triggers an interrupt.

This is not the only way, however. Newer processors include two instructions known as SYSENTER and SYSEXIT that are optimised to provide a quick transition to privilege ring 0 [3].

Regardless of how the transition to the kernel is handled, once the kernel receives a system call request, it needs to look up the handler for the specific system call and dispatch it. This can be done in many ways, but typically involves the kernel keeping a table of system calls and their corresponding functions.

An important note here is that the kernel does not unconditionally service a system call without first checking to see if the user task should be allowed the access it requested. This conditional behaviour serves as a way to implement security, since the very idea of a system call is to get a task of *higher privilege* to perform the action, and could be easily exploited if no restrictions were put in place.

Once the kernel has performed the requested service (or rejected the request), the kernel triggers a context switch back to the requesting task—its execution resumed at the instruction after the system call. If the operating system uses a software interrupt to trap into the kernel, the return to a task is typically done by the kernel executing the IRET instruction.

2.2 POSIX

Compatibility between operating systems is achieved through the operating system and user applications complying with Portable Operating System Interface for Unix (POSIX)—a collection of standards published by the Institute of Electrical and Electronics Engineers (IEEE) that defines how programs interface with Unix-like operating systems. According to the specifications, any program

written to compile on one POSIX-compliant operating system shall compile and run on any other POSIX-compliant operating system [4]. This means that POSIX-compliant operating systems must have compatible system calls, program and shell interfaces and signals, as well as various other Application Programming Interfaces (APIs).

POSIX-compliance is typically a goal for the developers of an operating system since it reduces the amount of work application developers need to do to port their applications to the target operating system. This compatibility obviates the need to reinvent the wheel.⁵

2.3 Minix 3

Minix (“minimal UNIX”) is a microkernel-based operating system developed originally by Andrew Tanenbaum for use as a teaching resource [5]. It takes the design elements of a microkernel to the extreme, running nearly every core service and driver in userspace, outside of the kernel. The latest version, Minix 3, was co-written by Albert Woodhull and many of Tanenbaum’s research students.

2.3.1 Kernel threads

There is no support for kernel threads in Minix 3, since the kernel itself does very little. Core operating system functions such as drivers, process management and networking stack are implemented in *servers*, which run as user tasks, negating the need for a multi-threaded kernel.

2.3.2 Software interrupts

Minix treats software interrupts in the same respect as hardware interrupts: the kernel is loaded on to the CPU and the handler for the given interrupt vector is invoked from a lookup table. The kernel dispatches this handler and uses the IRET instruction to return from interrupt. This is an important concept for the next section, which deals with how system calls work.

2.3.3 System calls and message passing

Unlike monolithic operating systems, user tasks cannot directly request Minix’s microkernel to perform tasks for it. Instead, they must make system calls to the user-space servers, which in turn send a message to the kernel requesting a function be performed. There are only three of these kernel calls in Minix 3:

send Kernel tasks use this kernel call to send messages to the user-space servers.

⁵Take, for example, echo—there are three mainstream versions of this application: GNU echo, BSD echo, and a bash built-in. All of the versions perform the same basic task.

receive Most kernel tasks call this to wait for a message to come in from a server.

sendrec This is the only kernel call that the user-space servers are allowed to make. It is the microkernel equivalent of making a system call. This kernel call serves a dual purpose: a message is constructed by a server and sent to the kernel (on behalf of a user task), and the server waits for a response from the kernel.

The above kernel call messages trigger a software interrupt to occur to trap into the kernel. This is done on interrupt vector 80. After checking the validity of the call, the kernel dispatches it appropriately, by using a lookup table to map the call number to a handler function. The kernel executes this handler, and returns control to the user-space server, which in turn passes the kernel's reply back to the user task.

2.3.4 Scheduler

Minix 3 implements a priority-based, round robin scheduler with tasks stored on different priority queues. Each task is given a predefined scheduling quantum at creation; this is decremented on every timer tick. Once a task expired its quantum, it is shuffled to the bottom of a *lower* priority queue. If a task does not consume its quantum, it is assumed to be blocking for I/O to occur and is shuffled to the *head* of the current queue when ready. This is a difference from a typical implementation of a round robin scheduler, where a task that is blocking would be shuffled to the *tail* of a queue.

2.3.5 POSIX-compatible

Minix 3 is certified as having full POSIX-compliance to the 1990 version of the standards. This technically means any program written on another POSIX-compliant operating system will compile natively on Minix 3, without the need for porting or reprogramming [6].

2.4 Linux

The Linux kernel (hereafter referred to as Linux) is a monolithic kernel written by Linus Torvalds, who extensively studied the code of Andrew Tanenbaum's Minix while he was a student. Although based on Minix 1, its design differs greatly—with all drivers and core system processes running in kernel space.

2.4.1 Kernel threads

Linux has a kernel threading library that allows multiple threads of execution inside the kernel. Linux uses threads to run background tasks inside the kernel, as they are lightweight and efficient to create and switch. The kernel threads library supports resource locking and resource sharing between threads via the use of mutual exclusion methods [7].

As of kernel version 2.4.6, preemption support was added to the kernel, allowing kernel threads (and indeed the kernel itself) to be preempted. This is available only if the kernel is built with CONFIG_PREEMPT set. Early benchmarks of having preemption enabled in the kernel resulted in reduced performance as it wasted CPU cycles in preempting a task when it is more efficient for threads to simply yield the CPU when they have finished a branch of execution [8].

Kernel preemption is now shown to be useful for real-time, interactive events, since a lower-priority task can be preempted for a higher-priority task to execute. However the degraded performance issue first seen in version 2.4.6 cancels the usefulness of preemption if the system is running tasks that favour throughput over response time (such as a server).

2.4.2 Software context switching

Linux performs context switching in software, opting to manually manage the resources a task holds. In the case of the x86 architecture, the kernel uses a single TSS for the transition from user to kernel mode.

2.4.3 System calls

The trap into the kernel from a system call was previously implemented through the software interrupt vector 80, which is available on all x86-derivative architectures. However once Intel developed the SYSENTER and SYSEXIT instructions for accelerated access to the kernel on a system call, Linux changed to using these instructions as of kernel version 2.5 [9].⁶

Once the kernel has been invoked to respond to a system call, it validates the system call number passed to it and looks up the handler in the system call table. After dispatching the system call, the kernel restores the state of the previously executing task and switches back to it, either via IRET or SYSEXIT. The user task then resumes execution as before.

⁶These instructions were introduced as part of the instruction set provided by the Intel Pentium II CPUs, and are available on all following CPUs.

2.4.4 Scheduler

As of version 2.6.23, Linux uses a *completely fair scheduler* that is significantly different to other operating systems, as it uses a red-black tree instead of multiple run queues to organise tasks. The scheduler is quite complex in comparison with other scheduler implementations, as it eliminates the notion of time slices—instead using nanosecond granularity to allocate a task's time on the CPU. It is an example of a fair share scheduler.

2.4.5 POSIX-compatible

Linux is, for the most part, POSIX-compliant. The kernel developers favour the Linux Standard Base, which extends POSIX, as a standard to comply with [10]. Although not officially certified by the IEEE as having full POSIX-compliance, as of kernel version 2.6.29, Linux declares that it is compliant with an older POSIX standard, approved in 2001 (the latest amendment to the standards was approved in 2008).

2.4.6 Dynamic timer ticks

An interesting feature of Linux's interrupt architecture is that it has the ability to silence timer interrupts until absolutely necessary. These dynamic timer ticks allow the CPU to stay sleeping if there is no work for it to do, rather than constantly being interrupted to service a timer tick. This can only work if all CPUs in the system are idling—quantitative analysis of silencing the timer interrupt when one CPU still has a workload shows that there is an overall loss in performance as the overhead in reprogramming the clock timer proves too expensive [11].

3 Implementation and Methodology

In this chapter, the implementation of Ulysses’ userland is described, and its methodology discussed. Like the research in the previous chapter, this chapter is sectioned into different subsystems that each glue together to form the userland. The research into different operating system’s implementations gave insight and influenced the design decisions used in Ulysses’ userland.

3.1 Anatomy of a task

Since no binary loader is implemented in Ulysses, a “task” is simply a C function that terminates by either explicitly returning or being killed by the operating system kernel. The anatomy of an example task shown in figure 2, and annotated below.

```
1     #include <string.h>
2     #include <unistd.h>
3
4     void main_hello_world(void)
5     {
6         unsigned short i;
7         char *str = "Hello, world!\n";
8
9         for (i = 0; i < 5; i++) {
10            write(STDOUT_FILENO, str, strlen(str));
11        }
12
13        return;
14    }
```

Figure 2: Anatomy of a simple Hello, world! task in Ulysses

The preprocessor includes on lines 1 and 2 are part of Ulysses’ libc implementation. They provide prototypes and data structures needed by the operating system and tasks, following the POSIX standard as required. Since the task function is linked in to the operating system itself, the task is technically a library—this allows it direct access to the libc as provided by the kernel.

Line 4 is the entry point for the task. To create this task, “hello.world” is typed at the shell prompt. Through a lookup table, the shell translates this string name into the C function `main_hello_world`. The shell then forks and executes this function in its child.

Line 10 makes a system call to write the string “Hello, world!\n” to the standard output stream. Since Ulysses has no concept of an I/O model, the “standard output” of all tasks is simply printed to the kernel’s virtual terminal buffer. The `write` function makes a system call as described in section 3.7.

Although line 13 is strictly not required since the return type of the task is void, it helps to show

that when the return statement is executed, the task will exit, causing its stack to unwind and the task to call `task_exit`, as described in section 3.3.1.

3.2 ps shell command

To show the tasks executing on the system at a given point in time, a shell command was added to display details of each task. An example output from this `ps` command is shown in figure 3.

```
ulysses> ps
PID PPID  UID CPU    STATE  NAME
 4   1     0 6550    3     [ring3] (3)
 0   0     0 9000    -     kernel (0)
 1   0     0  0      5     [kthreadd] (0)
 2   1     0  1      5     [initrd] (0)
 3   1     0  70     3     [shell] (0)
```

Figure 3: ps output

The **PID** column shows the process ID of the task. This gives a timeline of the order of creation of the running tasks. **PPID** is the parent process ID of the given task: the task’s creator. The **CPU** column displays how much time a given task has spent running on the CPU before being preempted, in milliseconds.⁷ This figure is never reset to 0 once a task has been created, instead showing the CPU time across the task’s entire lifetime.

A kernel thread is shown in the output by having its name enclosed in square brackets, much in the same way as Linux’s `ps` and `top` do. Its current state is given in the **STATE** column, corresponding to either 3 (executing on the CPU), 5 (sleeping but ready to run again), or “-” if the task is not a kernel thread. These “magic” numbers are based on the design put forward by Plank and Wolski [12].

The number in brackets in the **NAME** column indicates the privilege level that the task is running in: 0 meaning the kernel, 3 meaning user space.

3.3 Task creation

There are no defined methods of creating a “task” in a kernel. It is therefore up to the operating system designer to decide what data structures to create and populate. According to the research in section 2.1.1, the bare minimum that the CPU needs to execute a task is a default stack and page directory.

⁷Tasks that yield before a timer tick do not have their CPU time updated for that running slice.

3.3.1 Default stack

The default stack required by an x86 CPU, as described by Intel, is shown in table 1. The very bottom of the stack contains a pointer to the task exit routine to destroy a task. This is called when the stack is unwound as a task falls out of scope or returns.

EFLAGS	status register
CS	code segment selector
EIP	current instruction pointer
EDI	
ESI	
EBP	stack base pointer
NULL	
EBX	contents of general purpose register
EDX	contents of general purpose register
ECX	contents of general purpose register
EAX	contents of general purpose register
DS	
ES	
FS	
GS	
task_exit	memory address of task exit routine

Table 1: Default stack required for each task (from top to bottom)

The relevant frames on the stack are annotated with their meaning. All other frames are used internally by the CPU and do not need to be updated by the operating system under normal circumstances.

3.3.2 task_t structure

It was decided that Ulysses will store information about a task in a new structure, which is given the type of `task_t`. Each structure will contain everything that the operating system needs to know about a given task. Task structures will be linked together in increasing order of creation as a singly linked list.

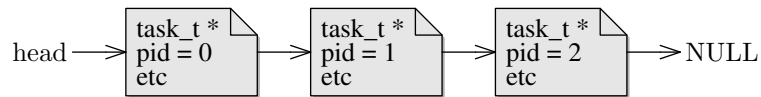


Figure 4: `task_t` structures in a singly linked list

The members of these structures are annotated below:

pid The process ID of the task. This is a uniquely generated value that is never reused. These IDs are declared as unsigned ints, and it is assumed the system will never be running enough tasks for this value to overflow.

ppid The process ID of the parent task.

uid The user ID of the task. root's tasks have a uid of 0.

name The ASCII-name of the task, terminated by a NUL byte ('`\0`').

esp The physical memory address of the task's current stack pointer. This is updated when a context switch occurs, and is used to restore the state of the task.

ebp The physical memory address of the task's base stack pointer. This is updated when a context switch occurs, and is used to restore the state of the task.

eip The physical memory address of the task's current instruction pointer. This is updated when a context switch occurs, and is used to restore the state of the task.

page_dir The current page directory for the task. Initially, this directory is set to the same as the kernel's current page directory. The directory is cloned and updated upon cloning an existing task.

s_ticks_left Number of scheduling ticks this task has left before preemption. Each timer tick decrements this value until it reaches 0, at which time the task is preempted and has its quantum reset.

s_quantum_size The size of this task's scheduling quantum. It is used as the maximum value of `s_ticks_left` when resetting a task's quantum.

kthread A pointer to the kernel thread struct for this task, or NULL if the task is not a kernel thread.

ring The CPU ring this task should execute in. It can only have a value of either 0 or 3. It can be represented as:

```
enum cpu_ring { RING0 = 0, RING3 = 3 };
```

next As the `task_t` structure is implemented as a singly linked list, this member points to the next task on the system, or NULL if it is the last.

3.3.3 Creating the task

The following procedure is followed to create a new task on the system:

1. A new `task_t` structure is created for the new task;
2. The task's default stack be set up as described in table 1;
3. A pointer to the current page directory is set in the task structure; and
4. The new task is added to the tail of the ready scheduling queue.

3.3.4 Cloning an existing task

Since the accepted method of creating new tasks on a Unix system is to clone an existing one, a fork function was implemented to clone the callee. A new task structure still needs to be created as described in section 3.3.3, however instead of using the calling task's page directory, a clone of it is taken and updated in the new task's structure. The current stack pointers are then read from the CPU registers and updated for when the task first executes.

3.4 Kernel threads

Implementing kernel threads in Ulysses would give the kernel the ability to run different subsystems as separate tasks, while allowing the threads to interact and cooperate to achieve different goals. They also provide a simple means of testing kernel features such as task creation and multitasking. Since they are lightweight, very little needs to be done to create one.

It was decided that a basic kernel threading module be created, based on Linux's kernel threads implementation and an outline proposed by Plank and Wolski [12]. These designs were chosen as they treat kernel threads as lightweight-processes, allowing the kernel scheduler to schedule these without it knowing the difference between a process and a thread.

Advanced features of threading such as joining and mutual exclusion that exist in Linux's kernel threads implementation were deemed unnecessary for Ulysses' kernel threads model, as no tasks would need this functionality for the purposes of this project.

A limitation with the chosen threading model is that there is no way to clone a kernel thread. Instead a new thread must be created and its entry point given as normal—making the code simpler. It was decided not to allow kernel threads to be cloned since there are no advantages in doing so:

unlike a process which requires its own address space and segments set up (which are much quicker to simply clone), a thread executes in the same address space as the kernel. The overhead in cloning a thread would be identical to simply creating a new one.

Three externally visible functions were created for controlling kernel threads, and are described in the following subsections.

3.4.1 `kthread_create`

The `kthread_create()` function creates a new kernel thread. It performs the following steps:

1. `create_task()` is called to initialise a new task structure.
2. A new kernel thread structure is created and a pointer to it is set in the task's structure. This identifies the task as a kernel thread.
3. The memory address of the thread's entry point is set as the task's default eip. The entry point is given by the following function prototype: `void (*func)(void);`
4. The scheduler's `add_to_queue()` is called to place the new task on the ready queue. This does not actually schedule the task, merely *allowing* it to be scheduled.

The thread can then be picked by the scheduler and executed as if it were any normal task.

3.4.2 `kthread_yield`

The `kthread_yield()` function makes the callee voluntarily give up the CPU and triggers a context switch, allowing another task to be scheduled and run. `switch_task()` saves the state of the callee so it can continue execution later.

3.4.3 `kthread_exit`

The `kthread_exit()` function terminates the callee. It makes a call to the `task_exit()` function which removes the calling thread and triggers a context switch. Since the thread has now been destroyed, `switch_task()` does *not* save the state of the callee. Calling this function is equivalent to simply letting the thread return on its own, since its stack would be unwound and `task_exit` called regardless.

3.5 Context switching

Of the two methods of performing a context switch, the software-based solution was taken since it is more flexible, and requires less lower-level x86 hardware support. Only one TSS is used for all tasks; this describes the transition from ring 3 back to ring 0. This is the same general concept that Linux uses to do context switching. This design decision was primarily based on the research that there is little overhead in performing the context switch in software, and that it is easier to set up and manage a single TSS as opposed to managing multiple ones.

3.5.1 Selecting a new task

Before the context can be switched, some “housekeeping” must be done to save the previously executing task, and a new task picked. This procedure described below.

1. If the current task should be saved, a copy of the current stack pointer (esp) and base stack pointer (ebp) are stored in the task’s structure.
2. The current instruction pointer (eip) is read and saved in the task’s structure.
3. The scheduler is called via `pick_next_task()` to pick the head of its ready queue. This becomes the next task to run.
4. The current page directory is updated to reflect the new task.
5. The kernel stack is saved, ready for a new task to execute.

The task’s state before being interrupted is saved so it can be restored later, continuing execution at the precise point it left off. This way, a task never knows that it was taken off the CPU. There are many other registers that could be saved, however since a context switch occurs so often, it was decided that as little time as possible be spent saving the task’s state.

3.5.2 Software context switch

Now that the system is ready to run the new task, the actual context switch can be done. This is described below.

1. Interrupts are disabled since a context switch is not re-entrant.
2. The instruction pointer is placed in a temporary CPU register (ecx). This is the (re)entry point for the new task.
3. The current stack and base pointers are loaded into their respective registers.

4. The new task's page directory is loaded on the CPU by moving a pointer to it into the `cr3` register.
5. Interrupts are enabled and a `nop` instruction is issued so the pipelined `sti` takes effect.
6. The kernel performs a long jump to the absolute address loaded in the `ecx` register. This starts (or resumes) the new task running on the CPU:

```
jmp *%%ecx
```

This method was chosen as it is quite minimal, again spending as little time restoring a task. It has the advantage of not having to restore the CPU's general purpose registers, as they are contained inside the task's stack, given by `esp` (they are pushed on to the stack when an interrupt handler is called).

Now that there is a method to switch a task on the CPU, there needs to exist a way for this to be triggered. The *trigger* for a context switch occurs inside `check_current_task()`, which is called on each timer tick. This function checks if the task which is running on the CPU has exhausted its time quantum. If it has, a context switch is triggered by calling `switch_task()`, preempting the task from the CPU. The control flow of this process is illustrated in Figure 5.

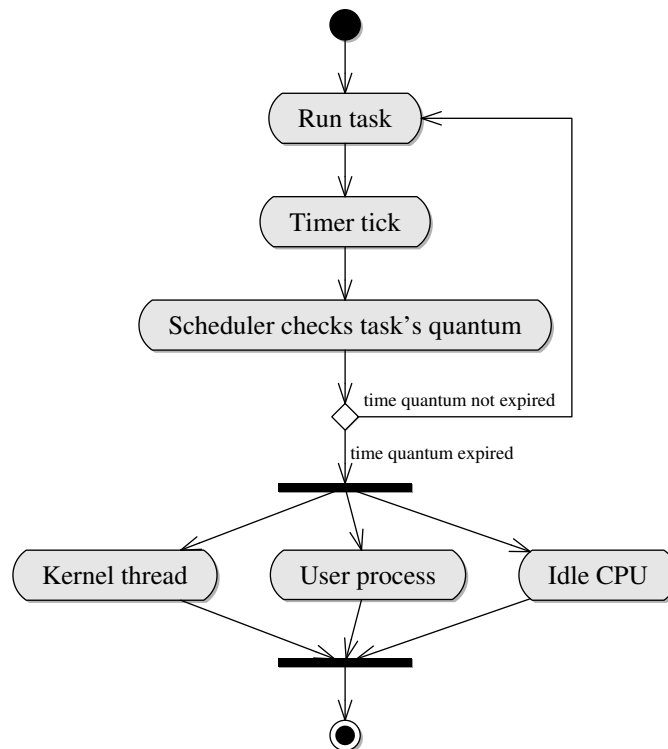


Figure 5: Scheduler activity diagram

A limitation of this method is that if the task executing *had not* exhausted its time quantum, a context switch still needs to occur to restore the task back on the CPU. This is a severe waste of CPU time, even if it simplifies the context switching code, and could distort the timing results given in chapter 4.

3.6 Privilege switching/user threads

Little needs to be done to perform an actual privilege switch. The only time the kernel needs to explicitly switch to a lower privilege ring is when the task is run for the first time. Since the current privilege ring is stored as a flag on the CPU itself, switching *back* to a lower privilege level is done automatically by the CPU when it inspects the requested privilege level (RPL) and current privilege level (CPL) flags in the currently loaded TSS.

The only way to switch privilege rings on x86 is to update the requested privilege level (represented by two code and data segment selectors) to reflect the ring of the new task, and trigger a return from interrupt (IRET). The trick here is that when the return from interrupt takes effect, the processor will be in ring 3. This avoids having to change the stack pointers over, saving some time when performing the switch.

3.7 System calls

The two parts to a system call—trapping into the kernel and dispatching the call handler—were chosen to closely resemble the methods used in earlier versions of the Linux kernel. The trap into the kernel is implemented using the software interrupt vector 80. Although this method is less efficient than using the accelerated SYSENTER and SYSEXIT instructions, it uses existing interrupt handling infrastructure in the kernel.

A user task makes a system call by pushing the call number onto the stack, as well as any arguments the call requires. It then executes an `INT 0x80` instruction to trap into the kernel. Once the CPU switches the kernel back on to the CPU, the kernel dispatches the system call by looking up and executing the respective handler, then returning control to the task via a return from interrupt. This procedure is represented in figure 6, showing the timeline of a system call as well as the process of it.

Four system calls were chosen to be implemented (in order): `exit`, `write`, `read` and `fork`. These were chosen since they are the most critical (they also happen to be the first few system calls in the Linux kernel), required for every task to perform their given function.

Creating a task requires forking from an existing task—this is done via the `fork` system call. Once a task is running on the CPU, it typically needs to get data from somewhere and put data to an output stream or similar. This is accomplished with the `read` and `write` system calls, respectively.

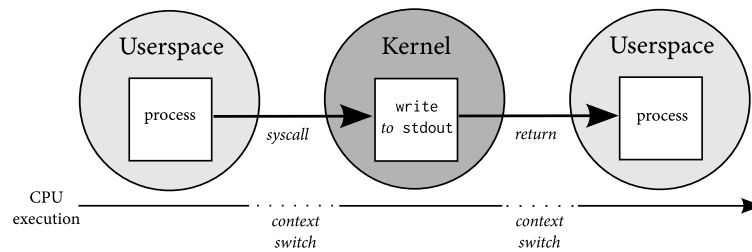


Figure 6: Timeline and process of a system call in Ulysses

Once a task has finished executing, it needs to `exit` so the operating system can clean it up and remove it from the scheduling queue(s).

Due to the limitation of Ulysses not having a real I/O model, the `read` system call simply returns that 0 bytes were read from the given source. Provided either `STDOUT_FILENO` or `STDERR_FILENO` are given to a `write` system call, the string passed to the kernel to write is output to the kernel's virtual terminal buffer, up to the first NUL-terminator or given size, whichever comes first.

`exit` is vital for task destruction—once dispatched, the kernel calls the same function that is set to the bottom of a task's stack to destroy it. Therefore calling `exit` is equivalent to explicitly returning from the task's main function.

4 Results

This chapter details the testing methods and results gained from placing the implementation from the previous chapter under test. The tests shown here are designed to prove that the subsystems implemented in Ulysses are functioning correctly, both inside and outside of the kernel.

Two testing scenarios were devised to complete the testing: unit testing, and *in situ* testing. Fortunately, many of the architecture-independent modules in Ulysses are sufficiently independent of other modules that they can successfully be isolated and compiled outside of the kernel. This allows unit testing of many of the kernel's subsystems, without interference from hardware or bugs in other modules.

However certain features of the kernel cannot be run in isolation. This typically includes any module that depends directly on hardware support to perform its given task. The only way to test these (without externally simulating the hardware itself) is to run the tests inside the kernel and observe the outputs. The tests are run with interaction from other subsystems, and possibly with interruptions that would taint the results of some tests. Therefore it is prudent to always take the observed results from an *in situ* test with *grano salis*⁸, and repeat the test if necessary.

4.1 Unit testing

A black box unit testing system was designed to isolate certain kernel subsystems and apply a unit test suite to them. To do this, the module at test must be compiled outside of the kernel, and linked with a test runner that can feed inputs to the module. The outputs can then be captured and compared with known correct values. This proves that the behaviour of a module's implementation is correct.

Since kernel subsystems typically call functions from other modules, it is necessary to simulate these function calls, by replacing them with test stubs. An example of this is the kernel's print function, `kprintf`. The test stub for this is a wrapper around the libc `vsprintf`, to ensure normal kernel output from the test goes to the test machine's standard output stream.

The test script can be invoked to run all test cases, or to run a specific test case. All tests in this chapter will use the latter. Full listing of the test script can be found in appendix [A.1](#).

4.1.1 Scheduler

The kernel scheduler can be tested to ensure that it can manipulate tasks on its singly linked list of `task_t` structures properly. A typical test scenario is shown below:

1. Create x handcrafted tasks and add each to the scheduler;

⁸“a grain of salt”

2. Iterate over x tasks and assert that asking the scheduler to pick a task yields each task in order; and
3. When the list has been expended, assert that picking another task will yield the first task.

This scenario shows if the scheduler is successfully shuffling tasks on its linked list. Results of running this test on the kernel scheduler with five tasks is shown in test output 1.

Test output 1 Result of running kernel scheduler test

```
sam@mia ~/Code/ulysses/test $ ./run_tests.sh -v sched
=====
Test #1: sched
=====
gcc -o sched/main.o -c -g -Wall -Wextra sched/main.c
gcc -o sched/sched.o -c -g -Wall -Wextra -isystem ../include
    /home/sam/Code/ulysses/kernel/sched.c
gcc -o stubs.o -c stubs.c
gcc -o sched/main sched/main.o sched/sched.o stubs.o
-----
sched: new task 0
sched: new task 1
sched: new task 2
sched: new task 3
sched: new task 4
sched: picked new task 0
sched: picked new task 1
sched: picked new task 2
sched: picked new task 3
sched: picked new task 4
sched: picked new task 0
SUCCESS: sched/main in 0m0.001s

1/1 tests passed
```

4.1.2 Kernel threads

The kernel threads implementation could be isolated to prove that its behaviour in creating new threads and yielding existing ones is correct, however this behavior can be seen in the context switching test in section 4.2.3. Therefore no unit tests were created for the kernel threading module.

4.2 *In situ* testing

Although many subsystems can be tested in isolation outside of the kernel, there are some subsystems that cannot be isolated. Instead these subsystems can be tested *in situ*, by running the test inside the kernel's normal operating environment. These tests can be created as shell commands and linked into the kernel. The behaviour and output can then be observed on the kernel's virtual terminal to determine if the module is working correctly.

4.2.1 Round-robin scheduling

Unit testing the scheduler shows that it is manipulating tasks correctly, however it cannot show if the scheduler is adhering to a round-robin scheduling policy. This can be done in place, and the scheduler's behaviour observed by use of the `ps` shell command. According to a round-robin scheduling policy, each task on the system should be given an equal amount of CPU time, over a given time period.

Therefore three tasks executing in a 30 second time period should each have approximately 10 seconds of CPU time each (discounting factors outside of the operating system's control, such as overhead in context switching and inaccuracies of the system timer). The output from Ulysses' round-robin scheduler is shown in figures 2 and 3.

Test output 2 `ps` output at start of test (0 seconds)

```
ulysses> ps
PID  PPID   UID  CPU    STATE  NAME
 4    1     0    0      3      [kt_test] (0)
 5    1     0    0      3      [kt_test] (0)
 6    1     0    0      3      [kt_test] (0)
[snip]
```

Test output 3 ps output at completion of test (30 seconds)

```
ulysses> ps
PID PPID    UID CPU    STATE  NAME
 4   1      0  9450    3     [kt_test] (0)
 5   1      0  9450    3     [kt_test] (0)
 6   1      0  9450    3     [kt_test] (0)
[snip]
```

By examining the CPU column in the output, it is clear that the three `kt_test` tasks each spent 9,450 milliseconds executing on the CPU. This equates to roughly 9.5 seconds each, matching the previously stated time of approximately 10 seconds of CPU time each. Since each task was given exactly equal time on the CPU, the requirements of a round-robin scheduling policy are satisfied, proving that the scheduler works as expected.

4.2.2 System calls

Testing a system call first requires creating a task that is operating outside of the kernel, in userland. This task should make a system call whose output can be observed on the kernel's virtual terminal. For this test, the `write` system call is applicable, although making any system call would be sufficient to test if the trap and kernel dispatcher are functioning correctly.

```
1 void main_syscall_valid(void)
2 {
3     write(STDOUT_FILENO, "Hello, world!\n", 14);
4 }
```

Figure 7: A user task function with a valid system call

Creating a task that runs the code in figure 7 in userland causes the following to be output on the kernel's virtual terminal:

Test output 4 Output from executing the `main_syscall_valid` test from userland

```
ulysses> syscall_valid
Hello, world!
```

Bypassing a system call and attempting to call `kprintf("Hello, world!\n");` directly from userland causes a page fault to occur and the task be killed by the kernel (shown in test output 5). This occurs since the kernel's virtual terminal buffers are not mapped in the user task's address space.

Test output 5 Output from attempting to call `kprintf` directly from userland

```
ulysses> syscall_invalid
Page fault at 0xC008FFB8 with error 7:
    write operation
    !ring 0
Page fault in pid 4: killing it.
```

This behaviour is to be expected and shows that a task running in userland cannot directly access the kernel or its data structures, instead having to make a system call to request this be done on their behalf. It shows that the kernel is properly receiving a system call with arguments and dispatching it accordingly.

4.2.3 Context switching

Context switching can be observed by running multiple tasks that each output the kernel's current uptime a number of times. Correct context switching will yield each task outputting in turn, with an ever-growing kernel uptime. To test this, three kernel threads were spawned that each loop three times and write the kernel's current uptime, then sleep for 50 milliseconds.⁹ The code for this is given in figure 8.

⁹Each thread is created with the same time quantum—each should be preempted and hence switched at a consistent rate.

```

1   void main_context_switch_thread(pid_t pid)
2   {
3       unsigned short i;
4       for (i = 0; i < 3; i++) {
5           kprintf("Hello from pid %d: kernel uptime %dms\n", pid, uptime());
6           sleep(50);
7       }
8   }

```

Figure 8: Code for context switching test

The sleep function call on line 6 is present to ensure that the task is preempted before it finishes executing a single loop iteration. This will allow preemption to be observed as the output from different threads should be interleaved. The output from spawning three threads with the above function is shown in test output 6.

Test output 6 Context switching during a kernel threads test

```

ulysses> kt_test
Hello from pid 5: kernel uptime 1803ms
Hello from pid 6: kernel uptime 1853ms
Hello from pid 7: kernel uptime 1903ms
Hello from pid 5: kernel uptime 2804ms
Hello from pid 6: kernel uptime 2854ms
Hello from pid 7: kernel uptime 2903ms
Hello from pid 5: kernel uptime 3805ms
Hello from pid 6: kernel uptime 3855ms
Hello from pid 7: kernel uptime 3904ms

```

The interleaved output of each kernel thread shows that the kernel is switching and preempting each of the threads. This output can be represented in a tabulated form for analysis:

Loop iteration	Kernel uptime		
	PID 5	PID 6	PID 7
Iteration 1	1,803 ms	1,853 ms	1,903 ms
Iteration 2	2,804 ms	2,854 ms	2,903 ms
Iteration 3	3,805 ms	3,855 ms	3,904 ms

Table 2: Analysis of context switching kernel threads

These results show that each loop iteration takes exactly 50 milliseconds to execute: the bulk of which is spent sleeping. The kernel is preempting the task before it enters another loop iteration, and it takes approximately 1,000 milliseconds for the task to resume executing. This delay can be accounted for as there were 5 other tasks running on the system during the test. Analysis of these timing results reveals that context switching is occurring at a consistent rate across the three tasks, and that the kernel is successfully preempting each task when its quantum expires.

The outcome of this test is that three subsystems are working as expected: the kernel threading module, scheduling, and context switching.

4.2.4 Execution of Hello, World! task

In section 3.1, the anatomy of a task in Ulysses was described with reference to a sample “Hello, World!” task (see figure 2). The result of running this task is shown in test output 7.

Test output 7 Hello, world! output

```
ulysses> hello_world
Hello, world!
```

4.3 Subsystem breakdown

Analysing the output from `diffstat` (see appendix A.2) shows which kernel subsystems were changed in this project. It is clear that by far the biggest amount of changes (on a line-by-line basis) is with the tasking code (`kernel/task.c`) and kernel scheduler (`kernel/sched.c`). This represents a fair proportion of the project’s focus, and was partially created from the now defunct `kernel/proc.c` module.

The next subsystem that has the most changes is the shell. This corresponds with the extra shell commands added to do *in situ* testing of various modules. It is interesting to note that the kernel threading code (`kthread.c`) has been severely reduced in lines of code over the course of this project. This is due to the bulk of the kernel threading code being made redundant with the additions of the multitasking code (since the scheduler does not know the difference between a thread and a process). At the end of the project, the kernel threading library contains only code relating to the setup and destruction of kernel threads, leaving the task management to the `kernel/task.c` module.

162 lines of code were added to the paging module (`arch/x86/paging.c`): this deals with the management and manipulation of page directories when tasks are cloned. This code did not previously exist in the kernel, as the kernel was the only task on the system (and hence only one page directory was used).

5 Conclusion

Operating systems exist to provide an abstracted, sandboxed environment for user applications to run and perform tasks. They ease the burden on application developers, allowing applications to be run on other compliant operating systems with little or no modification of the code. Without support for a POSIX-compatible userland, an operating system cannot achieve this goal.

Often in computer science there are many ways to reach a certain goal—this holds true with operating system development as well. The design decisions made when deciding *how* to implement a feature are not set in stone, which allowed a great degree of flexibility in the details of this project. The methods that other operating systems use, although tried and true, have often evolved out of the need for a certain feature set. Ulysses did not always need the same level of detail from these feature sets, so these methods merely helped to *influence* the designs made in the project, rather than control them.

Ulysses was brought to this project without the resemblance of a userland. It could not run user applications, and was mostly useless as an operating system. Implementing a POSIX-userland in Ulysses has given it the ability to run user applications, providing the mechanisms and framework for it to be used as a viable Unix-like operating system.

“UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.”

—Dennis Ritchie

6 Future work

As described in previous sections, there are limitations to the userland support implemented in Ulysses. A number of projects could be undertaken to overcome these limitations, described in the following sections.

6.1 Priority-based scheduler

Currently, Ulysses' scheduler has no concept of priorities. The algorithm could be rewritten to take priority into account when picking a new task. This would allow certain tasks to be given more CPU time than others, depending on their priority setting. The new scheduling model could be based in part on the priority queues that Minix 3 uses. Tasks would be shuffled amongst different queues depending on their priority setting, and the amount of time they have already spent executing on the CPU.

6.2 Kernel thread mutexes

Kernel threads typically need to interact and share data structures with other threads. Mutual exclusion techniques (mutexes) are typically used to avoid the simultaneous use of a shared resource. Two simple methods of implementing mutex support are disabling interrupts, or using a spinlock—either of which could be implemented in Ulysses.

By disabling interrupts around critical sections of code, the operations contained within are guaranteed to have exclusive access to all resources, as nothing can interrupt the execution. This could be implemented as a giant kernel lock, expressed in figure 9.

```
...
disable_interrupts(); /* lock */
/* this critical section now has exclusive access */
enable_interrupts(); /* unlock */
...
```

Figure 9: Example of giant kernel lock as a mutex

Although effective, this method is not efficient if the lock is held for a long time, or there are multiple CPUs attached to the system, as no other events can be processed while the lock is held. A more efficient method to implementing mutexes could be using a spinlock. A thread wanting exclusive access to a resource would simply spin in a loop waiting for the lock to become available. However this is only efficient if one thread holds a lock for a short period: if the kernel has to

preempt the thread and reschedule it while it is still holding a lock, the efficiency of other threads using a spinlock is negated by the overhead in performing a context switch.

6.3 Smarter preemption model

At the completion of this project, Ulysses' kernel supports preemption in both user and kernel tasks. It does not, however, have the ability to request a task be interrupted by the timer *only when it needs to be preempted*. Currently, a task is interrupted on every timer tick, only to be switched back to the CPU almost straight away (as the kernel decides that the task does not need to be preempted yet). This “dumb” method of determining when to preempt wastes CPU time with unnecessary context switches caused by timer interrupts.

Although the method is simpler, it is quite inefficient, and could be changed so the task will only be interrupted by the timer when its time quantum has expired. This time period can be calculated by the kernel before it switches to the task, ensuring that the next timer interrupt coincides with when the kernel should preempt it. This “smarter” preemption model is similar to Linux's dynamic ticks support and would reduce the amount of time the CPU spends switching contexts when it does not need to, increasing the efficiency of the operating system.

6.4 ELF binary loader

The major limitation with Ulysses' userland implementation is the lacking of a binary loader. A binary loader is used to load a pre-compiled and statically linked binary application into memory and execute it. The most common format of a binary executable on Unix-derived systems is ELF: the Executable and Linkable Format. An ELF loader could be implemented in Ulysses that would give the kernel the ability to run binary applications. At a bare minimum, the ELF loader would need to:

1. Parse the ELF header to determine what needs to be done before executing the application;
2. Map enough pages in the new task's page directory and copy the executable to the required virtual memory addresses; and
3. Set the task's instruction pointer to the given ELF entry point.

This would allow Ulysses to run applications in the same way other Unix-like operating systems can.

References

- [1] Brendan, “Re: Software-based or TSS-based Multitasking?” post on OSDev.org forum, 2008. [Online]. Available: <http://forum.osdev.org/viewtopic.php?p=117933#p117933>
- [2] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3A: System Programming Guide, Part 1*, Intel Corporation, 2008. [Online]. Available: <http://www.intel.com/Assets/PDF/manual/253668.pdf>
- [3] C. Jiang, “SYSENTER—fast transition to system call entry point.” [Online]. Available: <http://www.ews.uiuc.edu/~cjiang/reference/vc311.htm>
- [4] *Standard for information technology — portable operating system interface (POSIX). System interfaces*, IEEE Std. 1003.1-2004, 2004. [Online]. Available: <http://ieeexplore.ieee.org/servlet/opac?punumber=9157>
- [5] A. Tanenbaum, “Some Notes on the ”Who wrote Linux” Kerfuffle,” 2004. [Online]. Available: <http://www.cs.vu.nl/~ast/brown/>
- [6] A. Woodhull, “MiniFAQ about MINIX 3 and POSIX,” 2005. [Online]. Available: <http://www.minix3.org/doc/posix.html>
- [7] M. C. Daniel Pierre Bovet, *Understanding the Linux Kernel*, 3rd ed. O’Reilly, 2005.
- [8] M. J. Yannikos, A. Morton, and A. Arcangeli, “Linux: Kernel preemption, to enable or not to enable,” 2004. [Online]. Available: <http://kerneltrap.org/node/2702>
- [9] M. Garg, “Sysenter based system call mechanism in linux 2.6,” 2006. [Online]. Available: http://manugarg.googlepages.com/systemcallinlinux2_6.html
- [10] *Linux Standard Base Core Specification*, Linux Foundation Std. 3.2, 2007. [Online]. Available: http://refspecs.linux-foundation.org/LSB_3.2.0/LSB-Core-generic/LSB-Core-generic/book1.html
- [11] “The dynamic tick patch,” 2005. [Online]. Available: <http://lwn.net/Articles/138969/>
- [12] J. Plank and R. Wolski, “CS560 Lecture notes—KThreads Lecture 2.” [Online]. Available: <http://www.cs.utk.edu/~plank/plank/classes/cs560/560/notes/KThreads2/lecture.html>

A Code listings

A.1 test/run_tests.sh: Test script

```
1 #!/bin/bash
2
3 # test/run_test.sh - test runner
4 # part of Ulysses, a tiny operating system
5 #
6 # Copyright (C) 2008, 2009 Sam Kingston <sam@sjkwi.com.au>
7 #
8 # This program is free software: you can redistribute it and/or modify it
9 # under the terms of the GNU General Public License as published by the Free
10 # Software Foundation, either version 3 of the License, or (at your option)
11 # any later version.
12 #
13 # This program is distributed in the hope that it will be useful, but WITHOUT
14 # ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
15 # FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
16 # more details.
17 #
18 # You should have received a copy of the GNU General Public License along
19 # with this program. If not, see <http://www.gnu.org/licenses/>.
20
21 # Exit statuses:
22 # 0 = all tests passed
23 # 1 = scones failed (so compiler or linker error)
24 # 2 = could not cd into test module directory
25 # 3 = one or more tests failed
26
27 # -v is verbosity
28 if [ "$1" == "-v" ] ; then
29     verbose=1
30     shift
31 else
32     verbose=0
33 fi
34
35 if [ $# -eq 1 ] ; then
36     test="$1"
37 else
38     test="\ls -l | grep ^d | awk '{ print $8 }'"
39 fi
40
```

```

41 total=o
42 failed=o
43
44 for dir in $test ; do
45     total=$((total+1))
46     echo "=====
47     echo "Test #${total}: $dir"
48     echo "=====
49     scons -Q $dir || exit 1
50     echo "-----"
51
52     pushd $dir >/dev/null || exit 2
53
54     # determine where to send test case output - we do this here since it's
55     # inside the test case's subdir
56     if [ $verbose -eq 1 ] ; then
57         exec 3>.output
58     else
59         exec 3>/dev/null
60     fi
61
62     # run the test case, putting its output in a file (or null if no -v), and
63     # capture time's output
64     time_out="`(time ./main 1>&3 2>&3) 2>&1`"
65     status=$?
66
67     # if there is any output, print it now
68     if [ -f "./.output" ] ; then
69         cat ./output
70         rm -f ./output
71     fi
72
73     # extract the actual amount of time the command took (real)
74     time="`echo $time_out | grep real | awk '{ print $2 }`"
75
76     # output if the test failed or not
77     if [ $status -ne 0 ] ; then
78         echo "FAILED: $dir/main exited with status $status in $time"
79         failed=$((failed+1))
80     else
81         echo "SUCCESS: $dir/main in $time"
82     fi
83
84     popd >/dev/null

```

```

85     echo
86 done
87
88 echo "$((total-failed))/total tests passed"
89 scons -c >/dev/null
90
91 if [ $failed -ne 0 ] ; then
92     exit 3
93 fi
94 exit 0

```

A.2 diffstat output over project lifetime

```

sam@mia ~/Code/ulysses $ hg diff -r247:504 -X ideas | diffstat | \
    egrep '\.c|\.asm'

```

a/arch/x86/cpu.c		101	—
a/arch/x86/util.c		16	
a/kernel/proc.c		149	—
arch/x86/cmos.c		46	
arch/x86/gdt.c		92	+
arch/x86/halt.c		50	-
arch/x86/idt.c		29	
arch/x86/isr.c		135	++
arch/x86/keyboard.c		117	+—
arch/x86/kheap.c		30	
arch/x86/paging.c		162	+++
arch/x86/screen.c		47	-
arch/x86/startup.c		127	+—
arch/x86/timer.c		64	+
b/arch/x86/a20.asm		66	+
b/arch/x86/cpuid.c		129	++
b/arch/x86/cputest.c		43	
b/arch/x86/flush.asm		65	+
b/arch/x86/interrupt.asm		166	+++
b/arch/x86/loader.asm		56	+
b/arch/x86/serial.c		57	+
b/arch/x86/syscall.c		73	+
b/arch/x86/syscall_trap.asm		22	
b/arch/x86/task.asm		58	+
b/init/make_initrd.c		113	++
b/kernel/heap.c		226	++++
b/kernel/sched.c		153	+++
b/kernel/shell_cmds.c		324	+++++++

b/kernel/syscall.c		154	+++
b/kernel/task.c		406	+++++++
b/kernel/trace.c		98	++
b/lib/ctype.c		63	+
b/lib/errno.c		22	
b/lib/sleep.c		14	
b/lib/strtol.c		73	+
b/test/heap/main.c		59	+
b/test/sched/main.c		106	++
b/test/stubs.c		49	+
b/test/test_run/main.c		26	
kernel/initrd.c		92	+
kernel/kprintf.c		80	+
kernel/kthread.c		241	+——
kernel/main.c		136	++
kernel/oarray.c		28	
kernel/shell.c		171	++-
kernel/shutdown.c		60	+
kernel/util.c		160	++-
kernel/vfs.c		25	
kernel/vt.c		78	-
lib/itoa.c		7	
lib/stdio.c		18	
lib/string.c		47	+