COMP SCI 3004/7064 Operating Systems
Kernel Development Guide

Peter Kelly

Contents

1 Introduction

This guide is intended as a companion to the lecture slides used in the OS course, and as introductory material that will prepare you for doing the assignments. It describes the workings of a simple operating system kernel which illustrates some of the concepts covered in lectures. You'll be working with this kernel in your assignments, and in order to extend it you will need to understand how each part works.

These notes are not a replacement for the textbook, which you are advised to consult as well, as it goes into much more in-depth discussion and covers a wider range of topics than are addressed here. The textbook for this course is *Operating Systems Design and Implementation (3rd Edition)* by Tanenbaum and Woodhull.

The example code discussed in these notes is available at:

`http://adelaideos.sourceforge.net/`

Filename	Description
`Makefile`	Build script
`mkbootimage.sh`	Script to create boot disk image
`link.ld`	Linker script (used during compilation)
`constants.h`	Definitions of common constants used in the kernel
`kernel.h`	Header file for functions defined in the kernel
`segmentation.c`	Memory initialisation
`interrupts.c`	Interrupt handling
`start.s`	Assembly routines necessary for initialisation and interrupt handling
`main.c`	Initial startup code and timer/interrupt handling. This is where the main logic of the kernel goes.

Table 1: Files present in version 1

2 Getting started

This section describes version 1 of the sample kernel, which can be obtained from the URL mentioned in the introduction.

Up until now, all of the programs you have written run on top of an existing operating system such as Linux. This is the case for almost all software, in order to relieve the burden from the programmer of implementing functionality like file storage, networking, memory management etc. by putting this functionality into the OS. When you write a program, you mainly focus on what it needs to do, and simply make a call out to the OS whenever you need to access its functionality.

When writing an operating system however, you don't have any of the useful functionality that is normally available to application programmers. It is the operating system itself that must implement these features. When starting out, functions like `malloc` and `printf` aren't even present - since they're part of the C library, not the core language itself. As we go through this tutorial, we'll gradually add bits of functionality in each stage.

2.1 Hello World

Version 1 of the kernel contains the files listed in Table 1. The files `segmentation.c` and `interrupts.c` contain initialisation code that is necessary to set up the processor in the right state so that it can execute our code and handle events like key presses. `start.s` contains some low-level assembly code that is needed for the kernel to work, but can't be coded in C. You don't need to worry too much about these files for now; they're necessary, but rely on things we won't be covering in detail in the course. The only code of interest at this point is the code in `main.c`.

The most important function in `main.c` is `kernel_main`. This is the entry point for the kernel, i.e. the first thing that gets executed when the kernel boots. This first calls two initialisation routines, `setup_segmentation` and `setup_interrupts`. From here, we can basically do anything we want. For demonstration purposes, we'll print the string "Hello World" to the screen.

But how do we print? There is no `printf` function, because `printf` is part of the standard C library, and we don't have that available. On a typical operating system, `printf` does its formatting and invokes `write`, which is a system call to the kernel that outputs the string to the user's terminal. We don't have that either, because there is no concept of system calls or terminals yet. All we have is the hardware.

The answer to how we can get stuff on screen is to write directly to video memory. On PCs, the video card is initially placed in 80x25 text mode, and has its video memory located at address `0xB8000` (hexadecimal notation for 753664). Each character on screen is represented by two bytes, the first of which is the actual character to display, and the second is the foreground and background color. At the top of `main.c` there is a global variable `screen` defined, which is an array starting at address `VIDEO_MEMORY` (defined to `0xB8000` in `constants.h`).

The `screenchar` struct defined in `system.h` contains fields for the character, foreground color, and background color, so we can simply modify the elements of this array, and our text will appear on screen. The code in `kernel_main` shown below prints out a string by iterating over the characters defined in the string "Hello World", setting the appropriate element of the screen array to each character:

```
/* Output to screen by writing directly to video memory */
char *str = "Hello World";
for (x = 0; str[x]; x++)
  screen[x].c = str[x];
```

And that's all we need to print Hello World. There are some other functions defined in this file which we will come back to in a moment, but first let's look at how to compile the kernel and get it running.

2.2 Compiling and running

To compile the program, simply type "make"[1]. This compiles `start.s` using the assembler, and all of the others using gcc. This generates a set of `.o` files which the linker, ld, compiles into a single file called `kernel.bin`. Unlike normal compilation however, this is not a program that you can run directly under Linux - it is designed to be executed as a standalone program without any operating system underneath.

In order to run the kernel, it's necessary to use a program called a *boot loader*. This is the first thing that runs when a computer starts, and is responsible for loading the kernel file into memory and instructing the processor to start executing it. A boot loader is placed right at the very beginning of a disk, which is where the computer looks for an operating system as soon as it starts. For our testing we'll be using *GRUB*[2], an open source boot loader that is used by most modern Linux distributions.

The other thing you need of course is a computer to test on. It's a bad idea to use your main machine for testing, because doing so would require you to reboot each time you wan to test your kernel. Using a separate machine is better, though it still requires rebooting, which slows

[1] These instructions assume you are using one of the Linux machines in the CS labs. They should work on most Linux distributions, provided you have gcc and qemu installed. It is also possible to compile and run the kernel under Windows and OS X, but doing so requires some additional setup that we don't cover here.

[2] http://www.gnu.org/software/grub/

down your code-compile-test cycle. For these reasons, and because it's not practical for everyone to use two machines in the labs, we'll use a *virtual machine*. This is a program that runs on top of a regular OS and emulates the hardware. The virtual machine we'll be using is *qemu*[3].

GRUB can work with any type of disk - such as a floppy disk, hard disk, CD/DVD or USB memory stick. However, since we will be running under qemu, we will simply use an image file, and get qemu to pretend it is a floppy disk. Installing GRUB requires root access, which you don't have on the lab machines, so we've provided you with a disk image that is already set up, in the file `grub.img` in the source distribution. You will need to copy this file into the appropriate directory for the version of the kernel you are compiling.

All you then have to do is copy your `kernel.bin` file onto this disk image. This can be done with the following command:

```
mcopy -i grub.img kernel.bin ::
```

For convenience, the command "make boot" will do this step for you (as well as compiling), so you don't have to run separate commands each time. Using "make boot" requires the `grub.img` file to already be present in your source directory.

Once the kernel has been copied to the disk image, you can then run it under qemu using the following command[4]:

```
qemu -fda grub.img
```

2.3 Interrupt handling

Operating systems are based on an *event driven* programming model, where a program waits for external events to happen, and then takes some action based on what event has occurred. You may have used this general model before if you've done any GUI programming; keyboard presses, mouse clicks, and so forth are common events. In our kernel, we'll deal with two types of events: timer events, and keyboard events.

An *interrupt* is what happens when the kernel is notified of an event. Most of the time, the processor is executing application code, or sitting idle if there is nothing else to do. When an interrupt occurs, the processor stops what it was doing, and begins executing an *interrupt handler* function. When the function returns, the processor goes back to whatever code it was executing beforehand. In a sense, it's similar to a situation in which you are working on an assignment, and a friend interrupts you - you chat to them for a while, and then go back to what you were doing before.

On Intel x86 processors, each interrupt has a number in the range 0-255. The first 32 are reserved for *exceptions*, which are events relating to something that has gone wrong, such as a divide by zero exception. Another form of interrupt is an IRQ (interrupt request), which happens when a piece of hardware wants to let the kernel know that something has happened, such as a key being pressed. There are 16 different IRQs, which in our kernel are mapped to interrupts 32-47.

The `setup_interrupts` function called at the start of `kernel_main` is responsible for telling the processor what function to call when an interrupt occurs. `interrupts.c` contains a function called `interrupt_handler` defined for this purpose, which inspects the interrupt number

[3] http://www.qemu.org

[4] `-fda` tells qemu to use grub.img as the image for floppy drive A

and takes appropriate action. For interrupt 32, which occurs when the timer fires, it calls the `timer_handler` function in `main.c`. For interrupt 33, corresponding to a keyboard event, it calls `keyboard_handler`.

The remainder of the code in `main.c` demonstrates how to handle key press events and print whatever the user types in on screen. This implements basic terminal functionality, whereby the program keeps track of the cursor position on screen, and moves it whenever a character is printed. It also scrolls all of the text on the screen whenever the cursor goes beyond the last line.

When `keyboard_handler` is called, it is passed a parameter called `scancode`, which indicates which key the event relates to, and whether the event was caused by a key press or key release. The latter is determined by inspecting the last bit, which is extracted by masking `scancode` with `0x80`. The actual key code is obtained by masking the `scancode` variable with `0x7F`, which returns all but the most significant bit.

One complication is that the scan code does not correspond directly to a character, so we cannot print it as-is. Instead, we use it as an index into a lookup table, which we can use to find out what character it corresponds to. This is achieved by looking in the `kbdmap` array, defined in `interrupts.c`. Not all key codes correspond to characters, e.g. the alt/shift keys and arrow keys, so these will just be printed as spaces. In the case of shift, we treat this specially, keeping track of whether it is currently pressed or not. If it is, the character is taken from `kbdmap_shift` instead, which contains the uppercase versions of all keys.

```
void keyboard_handler(regs *r, unsigned char scancode)
{
  unsigned char key = scancode & 0x7F;

  if (scancode & 0x80) {
    /* key release */
    if (KEY_SHIFT == key)
      shift_pressed = 0;
  }
  else {
    /* key press */
    if (KEY_SHIFT == key) {
      shift_pressed = 1;
    }
    else {
      char c = shift_pressed ? kbdmap_shift[key] : kbdmap[key];
      write_to_screen(&c,1);
    }
  }
}
```

The code for `write_to_screen` is relatively straightforward; just have a look at the file for further details.

Another type of interrupt we can handle is the *timer interrupt*. This occurs at regular intervals, the frequency of which can be configured by the kernel. In the `setup_interrupts` function, we arrange for timer interrupts to occur 50 times per second. This means that once every 20ms, the `timer_handler` function will be called. As a simple demonstration of this, we just update

a variable which counts the number of ticks that have occurred, and print out a message every 5 seconds. The timer interrupt is crucial for implementing multitasking, which is the topic of Section 3.

2.4 C library functions

> This section describes version 2 of the sample kernel. Most of the new code is in `libc.c`.

When programming in C, you almost always use functions like `malloc`, `printf`, and `strlen` that are part of the C library. The C library is something that has to be available on a particular operating system for these functions to work - a program compiled by itself running directly on hardware doesn't automatically have them available. These libraries are present on Linux and other UNIX systems, but not on our kernel. Thus, we need to implement all of the library functions that we need to use.

Version 2 of the kernel contains a file called `libc.c` in which a few of the more important C library functions are provided. This includes `memset`, `memcpy`, `strlen`, `strcmp`, and `printf`. This is just a basic set in order to be able to perform common programming operations like printing formatted text to screen. Real C libraries are many times larger than this, and typically rely on operating system features that we haven't implemented yet. This will be enough for demonstration purposes however.

We will not go into detail about the implementation of these functions, as they are all reasonably straightforward C code that isn't directly related to OS programming.

One thing to note about this code however is that there are actually two versions of `printf` - the regular one, and another called `kprintf`. At this point, the two are identical. However, in later versions of the kernel that support the distinction between user mode and kernel mode, `kprintf` will be used for code within the kernel, and `printf` for code in user processes.

2.5 Comparing versions

If you want to see what has changed between two different versions of the code, you can use the `diff` command, e.g.:

```
diff -ur version1 version2
```

For each of the files that is present in both versions, this will show you a list of changes that have been made. For new files, `diff` will print out a message like the following:

```
Only in version2:  user.h
```

3 Multitasking

> This section describes version 3 of the sample kernel. Most of the new code is in `process.c` and `main.c`.
> See also: Tanenbaum, sections 1.3.1, 2.1, 2.4, and 2.5.

Modern operating systems support *multitasking*, which is the ability to have multiple processes running at the same time. Each process runs independently of the others, and is selectively given access to the CPU and memory in order to execute its instructions. It is common for many processes to be running at once, each of which does something different. For example, you may have a web browser, spreadsheet, and terminal program all running on your computer at the same time.

Although we often talk about multiple processes running "at the same time", this is in fact an illusion. A single processor is only capable of doing one thing at a time[5], so it is necessary for the kernel to provide an abstraction which makes it look as if several processes are running concurrently, but in reality they are actually taking turns in using the processor. This technique is called *time slicing*, and involves running each process exclusively for very short periods of time, and rapidly switching between them. These context switches happen many times per second, so as far as the user can see, it looks like all they are all executing simultaneously.

A process exists as an object within the kernel that has several pieces of state associated with it. This state includes a process identifier, the memory in use by the process, and the saved register values. When a process is running, it has exclusive access to the processor, and operates by accessing data in its own memory area. When it is not running, the register values it was using are saved in its state.

Each process may either be *ready*, meaning that it has work that it can carry out immediately if given access to the processor, or *suspended*, meaning that it is waiting for some event to occur before it can continue execution. The kernel maintains separate lists of ready and suspended processes. When a context switch occurs, the kernel will only switch to a process that is ready, since a suspended process cannot resume execution until the event it is waiting on happens. Examples of events that a process may wait for include the user entering some input, data becoming available over a network connection, or another process terminating.

Every process has a private area of memory associated with it, which it can use for storing data used during execution, such as variables. In most operating systems, there are generally several different memory regions, or *segments*, which comprise this private memory. The most important of these is the *stack segment*, which is used to store information about function calls and local variables. The *data segment* contains the process's heap, which used for dynamic memory allocation, and is managed by the `malloc` and `free` functions. The *text segment* contains the code that the program is executing, as well as any global variables.

Figure 1 shows a conceptual view of memory for a system that is running three processes. Each process has its own stack, data, and text segments, which are separate from the others. Additionally, there is an area of memory set aside solely for use by the kernel, and is used to store information such as the process objects themselves, and any other variables and data structures used by the kernel. With memory protection, discussed in Section 4, processes are explicitly prevented from accessing either the kernel memory or the private memory of other processes.

[5]Machines with multiple processors or cores *can* actually do multiple things at the same time, but time slicing is still used to cater for situations in which there are more running processes than cores. In our discussion here, we only consider the case of a single-core processor.

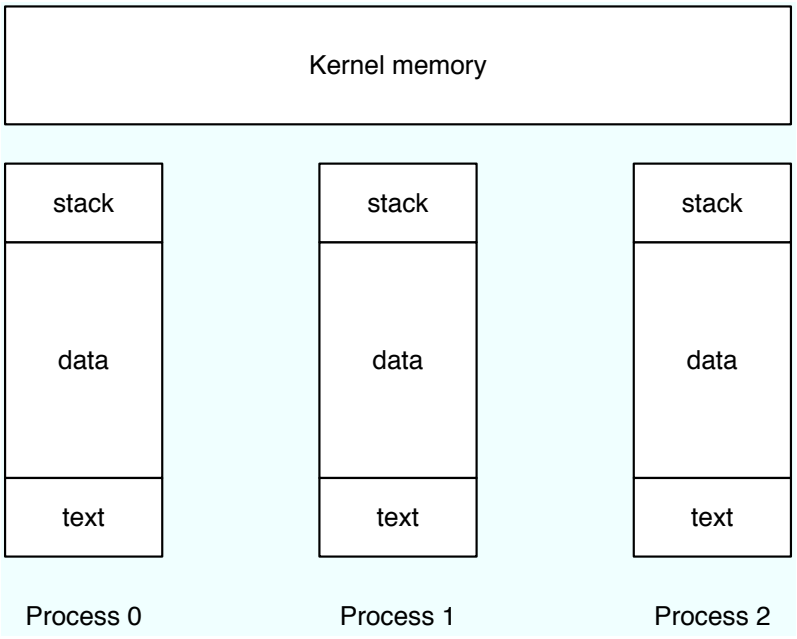

Figure 1: Memory areas used by the kernel and 3 processes

3.1 Implementation

For our initial implementation of processes, we only give each a stack segment. The data segment is only necessary if dynamic allocation is supported, which we shall look at in Section 6. A text segment is needed if the executable processes is loaded from a filesystem, but for now we shall only support processes that run code that is already part of the kernel, and thus does not require a separate private text segment.

Version 3 of the kernel contains the basic mechanisms to support multiple processes. The central data structure used for this is the `process` struct, which is shown below. An array of these called `processes` is defined in `process.c`; this array can store up to 32 elements.

```
typedef struct process {
  pid_t pid;
  int exists;
  regs saved_regs;
  int ready;
  struct process *prev;
  struct process *next;
  unsigned int stack_start;
  unsigned int stack_end;
} process;
```

The `pid` field is a unique number which identifies the process, and can be used to look up the process object in the `processes` array. The `exists` field is used to indicate whether or not this array element is in use, and is used when locating an empty slot in the array. `saved_regs` records the value of all CPU registers in the state that they were in when the process was last executing, and is used during a context switch to restore the CPU state so that it can continue executing the process where it last left off.

The `ready` field records whether or not the process is ready to execute code as soon as possible;

it is only a candidate for scheduling if this is set. The `ready` and `suspended` lists in `process.c` contain all of the processes in the respective state; the `prev` and `next` fields in this struct are used to store the forward and backward links in these doubly-linked lists.

Finally, the `stack_start` and `stack_end` fields store the bounds of the stack region in memory. Since each process is given a different area of memory for its stack, these will be different for each process. On x86 processors, the stack grows downwards, so the stack will begin at `stack_end`, and will increase downwards towards `stack_start` according to the depth of function calls made by the process.

3.2 Creating processes

The `start_process` function defined in `process.c` creates a new process and adds it to the list of ready processes. This takes care of finding an unused element in the `processes` array, determining the process id, allocating memory for the stack, and setting up the initial register values. The only parameter to the function is the start address of the process, which is the address in memory of the first instruction to be executed by the process. In C, we can get the address of a function's first instruction simply by using a function pointer, and so when calling `start_process` it is sufficient to simply specify the name of the function that the process should execute, e.g.:

```
/* Start two processes */
start_process(process_a);
start_process(process_b);
```

In `main.c`, there are two functions, `process_a` and `process_b`, which are used as test processes. Both of these simply execute in a loop, printing out a line of text at regular intervals. Between each print statement, they perform some computation by executing an empty loop a large number of times, just for the purpose of making some use of the processor. When you run this version of the kernel, you will see the interleaved output of both processes.

3.3 Context switches

The most important part of implementing multi-tasking is *context switching*, which is the act of changing the process that is currently being executed by the CPU. This must be done at regular intervals, and without each process having to do anything. We perform context switching within the `context_switch` function defined in `process.c`. This is called by the `timer_handler` function in `main.c`, which is executed 50 times per second as a result of interrupts generated by the system clock. Figure 2 shows an example of the way in context switching occurs between processes, and how suspended processes are ignored when making scheduling decisions.

The `context_switch` function takes a `regs` object as a parameter, containing the values of all CPU registers as they were when the timer interrupt occurred. The basic steps to perform a switch from this point are as follows:

1. Copy the CPU register values to the `saved_regs` field of the current process
2. Select another process to execute from the `ready` list, and update the `current_process` variable to point to this

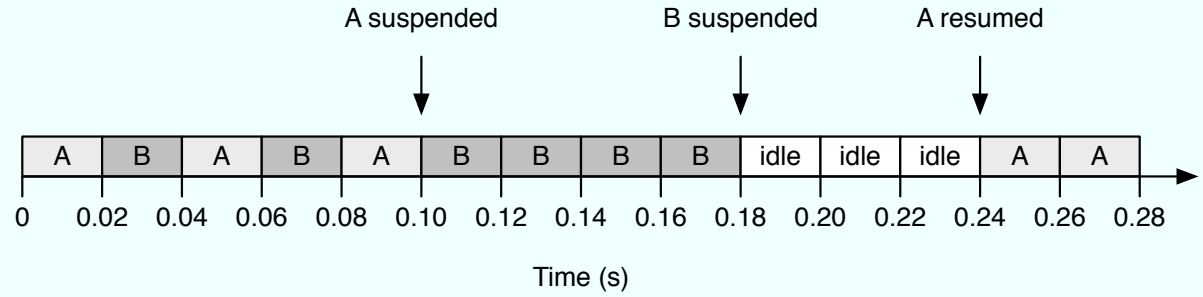

Figure 2: Time slicing

3. Copy the register values from the `saved_regs` field of the new current process to the `regs` object, which will be transferred to the actual CPU registers after the interrupt handler returns.

One complication with this is that the function must also handle the case where there is no current process, e.g. if the kernel has just started, or all processes are suspended. In this case, it sets up a temporary stack and causes the CPU to jump into an idle loop, using the `idle` function defined in `start.s`.

Although most of the logic to change to a different process is implemented in `context_switch`, there is a bit more to the story, which requires understanding what happens when an interrupt[6] occurs. The CPU maintains a table of interrupt handlers, each of which corresponds to a function that is called in response to a particular event occurring, such as a timer interrupt or key press. The assembly code in `start.s` contains a single handler function that is used for all interrupts. It saves all CPU registers to a `regs` structure on the stack, calls the `interrupt_handler` function in `interrupts.c`, and then subsequently restores the registers from the (potentially modified) `regs` structure back into the CPU. It is this `regs` structure that the `context_switch` operates on, allowing us to implement the higher-level logic in C.

3.4 Suspending and resuming

In order to suspend or resume a process, it is sufficient to simply change the `ready` field of the appropriate `process` object, and then transfer it to the `ready` or `suspended` lists. These actions are performed by two functions defined in `process.c`, named `suspend_process` and `resume_process`.

The `timer_handler` function in version 3 of the kernel demonstrates suspension and resumption by performing these actions at pre-defined time intervals, so you can see the effects on the processes when running the kernel. This also demonstrates the `kill_process` function, which gets rid of a process altogether. Note that after suspending or killing a process, a context switch should be performed, so that if that process was the current one, then a new one will be switched to instead.

[6] An interrupt is essentially the hardware equivalent of a signal

Exercises (optional)

1. Add some more functions to `main.c`, and start a process running for each of them. Observe how this affects the speed at which the other processes run, since the CPU must be shared between a larger number of processes.
2. Within `keyboard_handler`, allow the user to suspend and resume processes by pressing the numeric key corresponding to their process identifier.
3. Implement a mechanism that allows one of the processes to read input from the user, and then print out responses. The process should be suspended while it is waiting for a key to be pressed, and resumed when a key press event occurs. See if you can correctly handle the case where multiple key presses occur before the process gets its next time slice.

4 Paging and virtual memory

This section describes version 4 of the sample kernel. Most of the new code is in `page.c`, with a few changes made in `start_process`, located in `process.c`.
See also: Tanenbaum, sections 4.1-4.6

In Section 3, we discussed giving each process its own area of private memory, and the possibility of implementing protection that prevents one process from accessing the private memory of another. Additionally, it is useful for the kernel to provide an additional abstraction which allows processes to reference memory as if they had direct access to it, but instead ensuring that all data a process uses is stored within its own private memory. The way in which we can achieve both of these goals is to use *virtual memory*.

By default, software running on a processor accesses memory according to a *physical address*, which is the actual location in memory at which a particular piece of data is stored. An example of where an address may be used is when an attempt is made to change the value of a global variable; the software executes a write instruction for the address of that variable, and the processor arranges for the specified data to be stored at that particular memory location. Another example is when a function call is made; a call instruction is executed that specifies the memory address of the function to be called.

If there is only one program ever running on a machine, with exclusive access to the hardware, then this is quite acceptable. When a program is loaded into memory, the addresses referenced by the instructions in the compiled code are used as the locations of functions and global variables. However, this becomes a problem when there are multiple programs running, because each program may expect different things to be at a particular address.

The solution to this problem is *virtual memory*, a technique which allows each process to be under the illusion that it has direct access to memory, when in fact the memory used is stored in a different physical location to that of all other processes. Virtual memory makes a distinction between *logical addresses*, which are used by the process to refer to particular pieces of data in memory, and *physical addresses*, which are the actual locations at which the data stored. A mapping exists between these two, such that when an instruction attempts to read or write data from a particular logical address, this is translated by the processor into the corresponding

physical address, to which the read or write is then performed. Each process has a separate mapping, such that a particular logical address in one process maps to a different physical location than the same logical address in a different process. Figure 3 shows an example of this, with two processes writing to the same logical address, which is translated into different physical addresses.

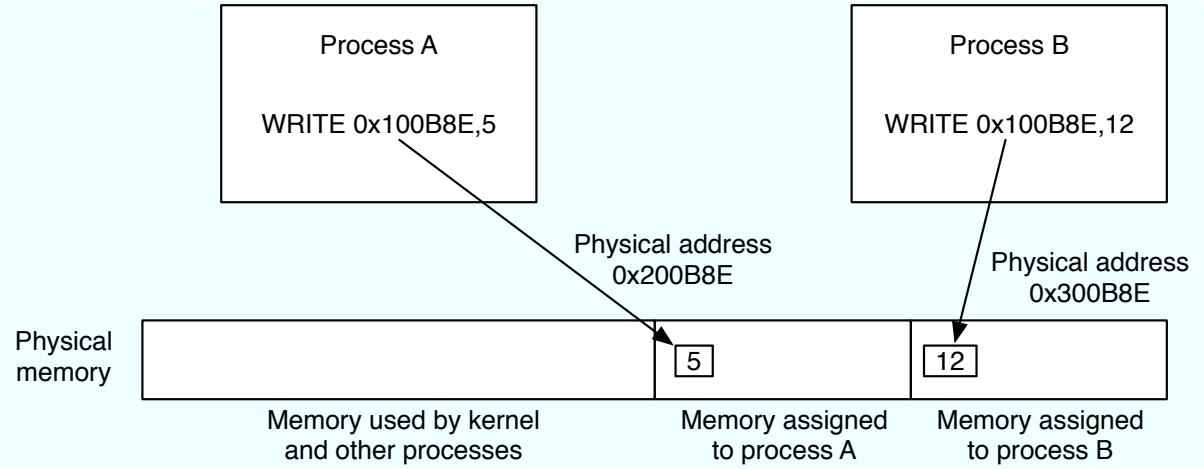

Figure 3: Logical vs. physical addressing

4.1 Page tables

The way in which the logical to physical address mappings are implemented is through the use of *page tables*. The address space is divided up into *pages*, which on x86 processors are 4kb in size. For each page in the logical address space, there is an entry in the page table that specifies which page in the physical address space the logical page corresponds to.

In Figure 4, a page table with mappings for the first 8 pages (32kb) of memory is shown. The entries in the right-hand column specify the start of a 4kb physical address range that the page corresponds to. In this example, the logical address 0x4782 corresponds to physical address 0x516782, because the logical address resides within page 4, which is mapped to a physical page located at 0x516000.

Page	Physical addr
0	0x100000
1	0x101000
2	0x102000
3	0x103000
4	0x516000
5	0x517000
6	0x518000
7	0x519000

Figure 4: Single level page table

Because the address space supported by most processors is very large, it is not generally feasible to use a single page table to cover all possible pages. For example, 32-bit x86 processors

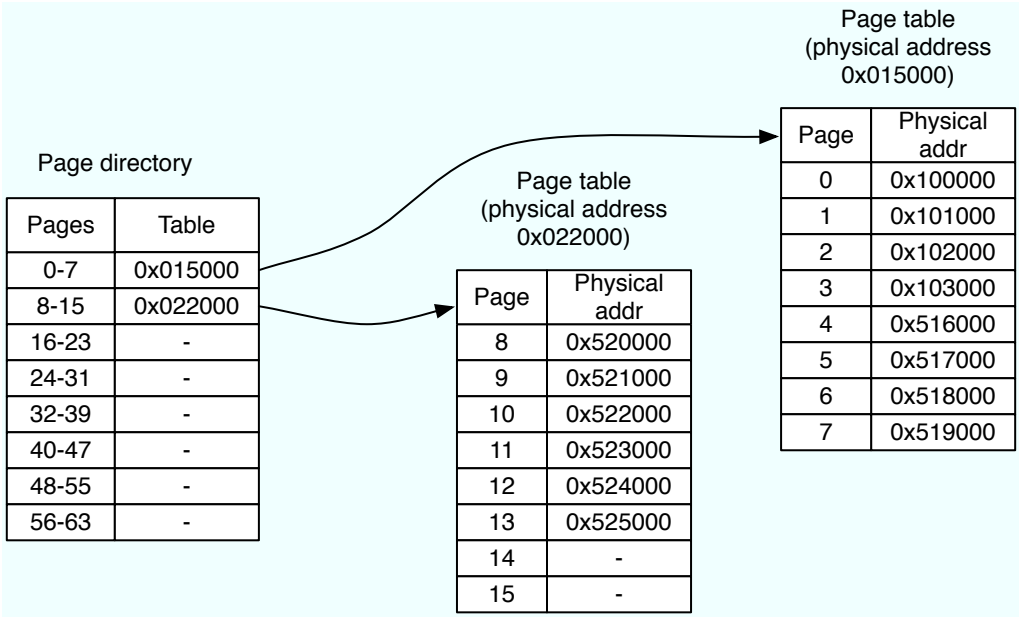

Figure 5: Multi-level page tables

support an address range of 4Gb, corresponding to a page table with 1 million entries. With each entry taking up 32 bits of memory, this would require 4mb per page table, which would consume a large amount of memory if many processes were active, since each requires a separate page table.

To reduce space requirements, it is more common to use a multi-level strategy, in which there are multiple page tables covering subsets of the address range, and a *page directory*, which contains the addresses of pages tables for each range. In Figure 5, we have a page directory in which each entry contains the address of a page table containing mappings for 8 pages. The first entry points to a page table at address 0x015000, containing the mappings for pages 0-7 as above. The second entry points to another page table, this time at address 0x022000, which contains mappings for pages 8-15. Because there are no mappings for pages 14 and onwards, there are no page tables allocated for later page ranges, which saves memory.

The x86 architecture uses this strategy for page mappings. Each page directory is an array of 1024 32-bit values, each containing the address of the page table for a particular page range, if it exists. Similarly, a page table is an array of 1024 values, each of which contains the physical address to which the corresponding page maps, if any. This scheme can support up to 1024×1024 4kb pages, i.e. 4gb of memory.

The entries within both the page directory and page table are not just addresses, but also contain several flags. Since each page is aligned on a 4096 byte boundary, the lower 12 bits of the entry can be used to store flags, which can be masked out in order to obtain the address. The three flags of most interest are:

- *present* - indicates whether this entry is present. For page directories, if this flag is not set, it means there is no table for the corresponding set of pages. For page tables, when this flag is not set, it indicates that the corresponding page is not mapped.
- *user* - indicates whether or not this page is accessible from user mode (discussed in the next section)

- *writable* - indicates whether or not the page can be written to by user mode code running within a process. If this is not set, but the page is user accessible, then user mode code may read from this page but not write to it.

The kernel can use this mechanism to selectively control which areas of memory a process is allowed to access. When a process is created, it is given a new page directory, which is initially empty. Mappings are established for common memory that is shared by all processes (e.g. the executable code within the kernel), and mappings to the private stack area for that process is also set up. Since the memory of other processes is not mapped within the page directory, it is not possible for the process to access the memory of others.

4.2 User mode and kernel mode

The notion of using memory protection raises an important issue: If we are to prevent a process from interfering with other parts of the system, then how do we do this while still enabling the kernel to take privileged actions? The answer to this problem lies in a hardware feature of the processor called *protection levels*.

At any given point in time, the processor runs in a particular protection level. Processor architectures differ in how many levels they provide, but there are generally at least two: *user mode* and *kernel mode*[7]. User mode permits only a limited set of operations to be executed; all normal computation facilities that applications need are allowed, but privileged operations such as accessing hardware devices, disabling paging, or shutting down the system cannot be carried out. Kernel mode allows full access to the whole system; code running in this mode can do anything it likes.

Operating systems that support protected memory always run processes in user mode. This allows the kernel to maintain strict control over what each process is allowed to do, and to implement additional security facilities such as user accounts and filesystem permissions on top of these basic facilities. All operations such as writing to the screen, reading data from the disk etc. must be performed by the kernel, and these actions can be requested by processes. The mechanisms for making these requests are called *system calls*, which are discussed further in Section 5.

As far as paging is concerned, the distinction between kernel and user modes is relevant for understanding how virtual memory provides security. It is not sufficient to just set up page tables for a process that only map its private memory, since the process could simply disable paging or switch to a different page directory, which would enable it to access other areas of physical memory. Processes must be prevented from doing this by having them run in user mode, and only allowing the kernel to manipulate page mappings. This way, only the kernel can decide what memory a process is allowed to access.

The permission bits set in the page table entries determine the circumstances under which a page may be accessed. The *user* bit determines whether or not code running in user mode has access to the page - this is normally set, but for some parts of memory such as certain private data structures in the kernel, this may be disabled. Having page tables set up in this manner enables the kernel to still use the page mappings for a particular process when executing a system call, while still being able to access its own private memory. If the *user* bit is set,

[7] Sometimes known as *supervisor mode*

then the process can optionally be given read-only access to the page; this is controlled by the *writable* bit. The kernel can always write to all pages.

In version 4 of the kernel, the last action taken by `kernel_main` is not to simply enable interrupts, but also to enter user mode. When it does this, the processor will now be running in user mode by default, and only switch to kernel mode during interrupt handling. This causes the processor to enforce the above-mentioned security restrictions on all processes. Whenever a process violates these restrictions, an exception interrupt is raised, enabling the kernel to deal with the situation in whatever way it chooses. One example of these exceptions is a *page fault*.

4.3 Page faults

The kernel needs a way to handle the situation in which a process tries to access memory it does not have access to, either because there is no mapping in the page table, or because it violates the permissions of the relevant page. This situation is known as a *page fault*, and causes an interrupt to occur, which the kernel can respond to using the usual interrupt handling mechanisms. The normal action to take in such a situation is to simply kill the process. You will have undoubtedly have seen this happen sometimes when working in C, when your program tries to access an unmapped area of memory, e.g. by dereferencing a NULL pointer.

Page faults are also useful for implementing *swapping*, which is where the kernel is capable of providing processes with the illusion that they have access to more memory than is physically available, by temporarily storing some parts of memory on disk. This is sometimes (confusingly) what people refer to when they use the term "virtual memory". A page fault may occur because a process tries to access part of its memory that it does actually have permission to access, but is stored on disk. The kernel then reads the page from disk, stores it in physical memory, and updates the page table to point to its new physical location. Our demonstration kernel does not support swapping, though most mainstream operating systems do.

4.4 Implementation

Paging is a hardware feature - the processor itself implements all of the logic necessary to translate logical addresses to physical addresses when a memory access occurs, and to enforce the permissions set within page tables. Thus, most of the hard work is already done for us, and all the kernel needs to do is to manage the allocation of physical memory, and set up the page tables for each process. Version 4 of the kernel includes a file called `page.c` which contains the necessary functions to do this.

4.4.1 Page allocation

The first two functions in `page.c` are `alloc_page` and `dealloc_page`. These manage all of the physical memory with the exception of the first 6mb, which we have chosen to set aside for the kernel to use for its own purposes. Since these functions are actually memory allocators themselves, they do not call `malloc` in order to get hold of memory to use - in fact `malloc` doesn't exist in our kernel yet. Instead, they just access the memory directly, starting from the 6mb mark, going upwards until the end of physical memory. The `page_end` variable declared at the top of this file stores the end of the highest-numbered physical page that has been allocated so far; everything above this is considered free memory that is able to be used for new pages.

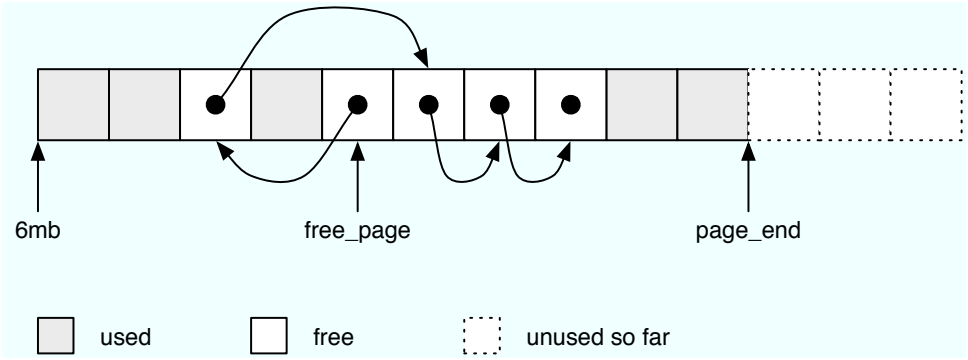

Figure 6: Free list allocation

In addition to the `page_end` variable, `alloc_page` and `dealloc_page` also maintain a list of free pages. The first time a particular physical page is allocated, it will be obtained from the value currently in `page_end`, which is subsequently incremented by 4kb. However, when a page is freed, it will be added to the free list. The reason for this is that there may be other pages above the freed page that are still in use, so we cannot simply decrement `page_end`. The free list is maintained as a linked list, and all new allocations check this first when looking for a free page. Figure 6 shows the state of memory after several allocations and deallocations have been performed. Note that the links are stored in the free pages themselves, which avoids the need for a separate data structure stored outside of the memory managed by these functions.

The reason why this allocation scheme is so simple is that all of the blocks are of the same size, so it is not necessary to cater for holes of different sizes when looking for space to allocate. More general allocation schemes discussed in lectures such as first fit, best fit, and buddy allocation involve more sophisticated logic, since they must attempt to minimise space wastage that comes about from having holes of different sizes that cannot always be completely filled.

4.4.2 Mapping pages

Setting up of page table entries is performed by the `map_page` function in `page.c`, which is relatively straightforward. For a given logical to physical address mapping, it first calculates the logical page number of that address, and then the positions within the page directory and appropriate page table that correspond to that address. If the page table does not exist, it created and added to the page directory. The entry within the page table corresponding to the logical address is then set to the specified physical address, combined with the permission flags and presence bit. The `map_new_pages` function automates the process of allocating multiple new pages and adding mappings for them.

Within `start_process`, a page table is created for each new process, and mappings are established for the stack memory, as well as certain memory within the kernel. This latter area of memory is only accessible to code running in kernel mode, with the exception of the memory between `kernel_code_start` and `kernel_code_end`. These values correspond to the region in memory containing the actual compiled code of the kernel; this needs to be readable by the process in order to execute its instructions, since at this point we only support processes that execute code that is already compiled into the kernel, as opposed to loaded from separate executable files.

In this version of the kernel, we need to make a special exception for two other areas of memory:

the video memory, and the kernel's global variables. The reason for this is that processes currently output data by calling the `write_to_screen` function. This writes directly to video memory, and uses the `xpos` and `ypos` global variables. All processes must be able to write to this memory in order to display their output. This is only a temporary measure until we have support for system calls, which we shall discuss in the next section.

The region of logical memory now used for the stack is the same for each process. This is made possible by the fact that we are able to map this logical address range to different sets of physical pages for each process. Because all processes have their stack start at the same logical address, we no longer have a need for the `allocate_memory` function that was previously used to give each a process a different area to store its stack.

5 System calls

> This section describes version 5 of the sample kernel. Most of the new code is in `syscall.c`.
> See also: Tanenbaum, sections 1.4, 2.6.8

Section 4.2 introduced the concepts of user mode and kernel mode. Processes run in user mode, and are not able to perform privileged operations or access areas of memory that belong to other processes or the kernel. However, privileged operations are necessary for all applications, and thus it is necessary for the kernel provide a safe, controlled way in which these can be carried out on behalf of processes. The mechanism for doing this is known as a *system call*.

A kernel must provide some set of system calls which are accessible to processes. Each call is simply a function within the kernel that does something which cannot be done by a normal process. Examples include creating a new process, reading the contents of a file, or writing text to the screen. Because these functions must run in kernel mode however, they cannot simply be called from a process using the standard function call mechanism. Instead, they must use a different execution path which allows the processor to temporarily switch into kernel mode while executing the function, and then go back into user mode when it returns.

Processes always run in user mode, but interrupt handlers run in kernel mode. For example, the timer handler, which calls `context_switch`, needs to run in kernel mode so that it can change to a different process and arrange for the CPU to use that process's page directory instead of that of the previously running process. Similarly, the keyboard handler must run in kernel mode, as it is necessary to interface with the keyboard controller in order to find out which key has been pressed. The interrupt handling mechanism is thus an ideal way to implement system calls, because we can use the logic to switch between privilege levels that is already there.

In our kernel, we have chosen to use interrupt no. 48 to represent system calls. This is the lowest unused interrupt after the IRQs set up within `interrupts.c`. Whenever a process wants to make a system call, it pushes the arguments to the call on to its stack, in the same way as for normal function calls. It then places a number identifying the system call to be made in a register, and executes an instruction which forces an interrupt to occur. From the programmer's perspective, all system calls look like regular functions; the difference is that they are implemented in the assembly file `calls.s`, which contains the instructions to set the call number and cause the interrupt.

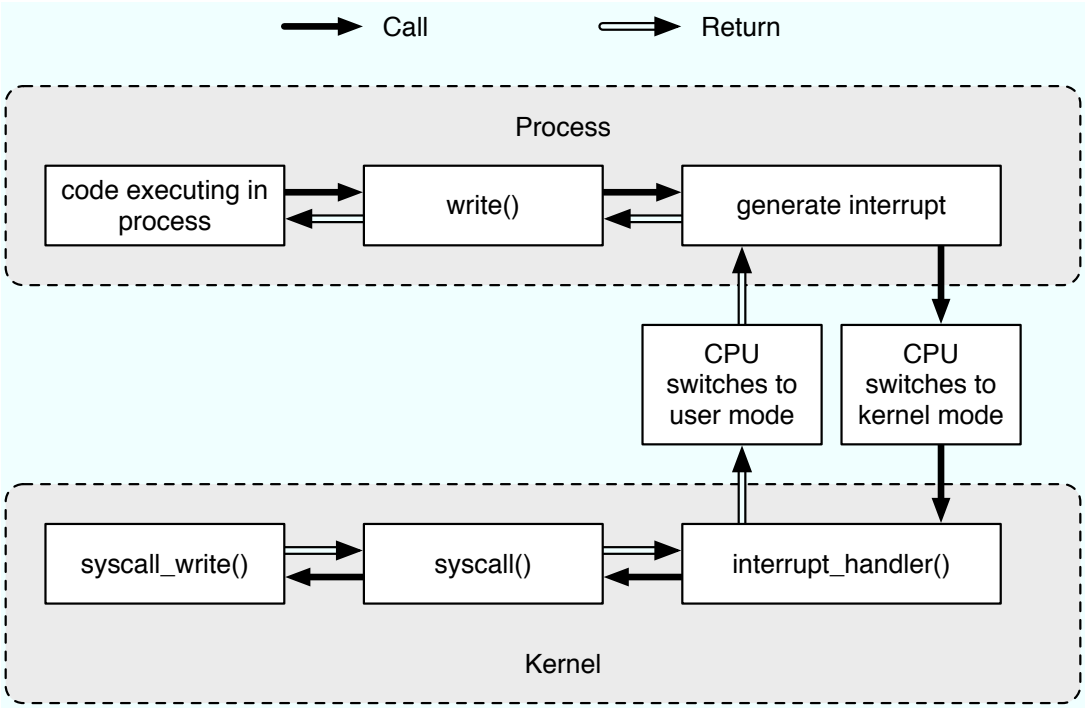

Figure 7: Control path for invoking a system call

Figure 7 shows the flow of control that occurs when a system call is made. We have just described the stages that happen within the process, shown in the top part of the diagram. When the interrupt occurs, the CPU switches to kernel mode, and executes the interrupt handler, which is actually implemented in assembler within `start.s`. This pushes the register values onto the stack, and then calls the `interrupt_handler` function, which is implemented in C. The registers that were put onto the stack are available as a parameter to this function, as a `regs` object. The interrupt handler then calls the `syscall` function, which inspects the registers to determine which system call was requested, and then dispatches to the appropriate handler function. In Figure 7, the system call being made is `write`, which is implemented by the handler function `syscall_write`.

The parameters to the system call may be accessed by looking at the process's stack. One of the saved registers is the *stack pointer*, which the `syscall` function uses to determine the location in memory of the parameters, which it then passes to the handler function. The handler for the system call then performs whatever actions are necessary, which can include privileged operations such as writing to any area of memory, since this code runs in kernel mode. Once the handler function returns, control is passed backwards along the same path, until it returns to the process that was previously executing. In some cases, such as the `read` or `exit` system calls, the process may have been killed or suspended. If this is the case, `syscall` performs a context switch, so that a different process will be executed when the interrupt handler returns.

5.1 The `write` system call

Up until this point, the test processes within our kernel have been using the `write_to_screen` function, called from `printf`, in order to output data to the screen. This has required the processes to have permission to write directly to video memory, and when setting up the page mappings in `start_process` we had to make an special exception for the video memory and

kernel global variables, to access `xpos` and `ypos`. However, processes should not really have permission to modify the video memory, since the kernel is supposed to control all access to hardware.

With system calls, we can modify the output logic used by processes so that everything that is printed to screen goes through the system call mechanism. This allows the kernel to do the actual updates to video memory on behalf of the processes. And since the screen is a shared resource, using a system call also allows the kernel to prevent race conditions by ensuring that only one update is occurring at a time. Race conditions like when two processes simultaneously call the `scroll` function, which could lead to incorrect results, are thus avoided.

In version 5 of the kernel, the `syscall_write` function in `syscalls.c` is invoked whenever a process invokes the `write` system call. All it does is simply call `write_to_screen`, and returns the number of characters written. Since this system call is now available, the `printf` function in `libc.c` has been updated to invoke `write`, instead of calling `write_to_screen` directly. The temporary code in `start_process`, which gave the process permission to write to video memory and kernel data, has also been removed.

In Section 2.4, we mentioned that there were two versions of the `printf` function: the regular one, and another called `kprintf`. The latter is intended for use within the kernel itself, such as when we need to print out debugging information from within interrupt handlers or during initial kernel startup. `kprintf` calls `write_to_screen` directly, just as `printf` used to, and thus it can only be used when running in kernel mode. The normal `printf` can no longer be used from within kernel mode, since it invokes a system call, and the way in which we have implemented system calls only permits one to be active at a time.

5.2 Other system calls

We have implemented a few other system calls to demonstrate common functionality that is provided by most kernels. The `getpid` function simply returns the process identifier of the current process. Since this is part of the `process` object stored within the kernel's private memory, it cannot be directly accessed from user mode. The system call provides controlled access to this field, in that processes can read it but not modify it. Similar system calls are often provided for other process fields such as the current working directory within the filesystem.

The `exit` system call is used by a process to terminate itself. When this call is made, the `kill_process` function is called, and the handler function returns the special value `-ESUSPEND`, indicating to the `syscall` dispatching function that the current process is no longer running, and it should perform a context switch. Processes should always call `exit` as their last action, to indicate the kernel that they have no more instructions to be executed.

6 Dynamic memory allocation

> This section describes version 6 of the sample kernel. Most of the new code is in `buddy.c`.

Virtually all programs need to dynamically allocate memory on a heap. In C, this is done using the `malloc` function, which sets aside an area of memory of the requested size, and returns

a pointer to the program. This can be used to store data structures. The lifetime of memory allocated in this manner is controlled by the program, which calls `free` when it no longer needs the memory. This scheme is more flexible than using memory on the stack, because stack frames disappear when their corresponding function call returns.

There are many different ways to implement dynamic memory allocation. In each case, a specific region of memory is designated as the heap, and the `malloc` and `free` functions manage this according to whatever technique is in use. The simplest possible implementation is to have `malloc` keep track of the lowest unused address, and increment this whenever new memory is allocated. This works well for allocation but does not permit memory to be freed, since there is no record kept of what blocks of memory have been given to the application, other than the end of the last one. We used this approach in the `allocate_memory` function in version 3 of the kernel, where we needed a simple way of obtaining a memory region to use as a stack when creating a process.

First fit and *best fit* are two schemes which keep track of the last allocated address, as well as all of the blocks of memory which have been allocated. When a block is freed, it is marked as unused, and new allocations will first look at the list of free blocks to see if any of the required size are available. First fit simply takes the first block that is at least as big as the requested size. Best fit searches all of the free blocks until it finds the smallest one that is large enough to store the requested block, in order to minimise wastage. Both of these schemes are discussed further in the lecture slides, and in Tanenbaum section 4.2.2.

A drawback of the first fit and best fit algorithms is that they can easily lead to fragmentation, in which there are lots of small holes that cannot be filled, even though they may collectively add up to enough memory to satisfy new allocation requests. In our kernel, we use a different scheme called *buddy allocation*, which is designed to minimise fragmentation.

6.1 Buddy allocation

Buddy allocation manages a region of memory by dividing it up into blocks whose size is always a power of two. Whenever an allocation request is made, the requested size is rounded up to the nearest power of two, and a block of that size is allocated. If the block size (after rounding) is less than that of the smallest free block, then the free block is split up repeatedly until a block of the appropriate size is obtained.

Initially, the memory region managed by the buddy allocator is treated as one large block of size 2^k, for some value of k. As allocation requests are made, this can be split into blocks of size 2^{k-1}, each of which can optionally be split into blocks of size 2^{k-2}, and so forth. Splitting is performed to minimise wasted space by only handing out blocks of the smallest size necessary to fulfill an allocation request.

Figure 8 shows the layout of a 2mb memory region which is managed by buddy allocation, and how the block divisions change as allocation requests come in. In step 0, the memory is one large free block of 2mb. In step 1, an allocation request is made for a size of 200kb. This is rounded up to the nearest power of of two, which is 256kb. Since 256kb is smaller than 2mb, the block is split in two, resulting in two free blocks of size 1mb each. The first of these is split again, resulting in two 512kb blocks. Finally, the first 512kb block is split in two, with the first half marked as allocated and returned to the program.

In steps 2-4, more allocation requests come in. Separate free lists for each block size are maintained, which allows the allocator to determine whether it can hand out an existing block,

Figure 8: Buddy allocation

or whether it needs to look for a free block of a larger size and split that. The second and third allocations both require 256kb blocks; for the second, an existing free 256kb block can be used, but for the third, a larger block must be split. The fourth allocation needs a block of 512kb; this is obtained by splitting the remaining 1mb block.

When freeing memory, attempts are made to coalesce blocks together. This is where the notion of *buddy* blocks becomes useful. The buddy of a block is the other half of the larger block from which both were obtained. When freeing a block, we inspect its buddy to see if it is also free; if so, it is possible to coalesce both into the larger block that they were originally both part of, and update the free lists accordingly. This is possible in steps 6 and 7 in Figure 8. In the last case, two merges occur, since the 256kb blocks can be coalesced into a 512kb block, and then this can be combined with the adjacent 512kb block to form a 1mb block. Larger blocks are more desirable, since they can be used to satisfy both large and small allocation requests.

6.2 Free list management

As mentioned in the previous section, buddy allocation requires that separate free lists be maintained for each block size. One way of doing this is to use linked lists, and to store the links in the free pages themselves. This is not the only way of maintaining the lists, but it simplifies implementation because it avoids the need for a separate data structure outside the region being managed that would need to grow and shrink dynamically. Instead, it is sufficient to just have a fixed size array which stores the pointer to the first block in each free list. The array can be indexed by the power of two of the block size. For example, the free list of 128kb blocks can be accessed by inspecting array element 17, since 2^{17} = 128kb. Figure 9 shows the free lists for a heap with several free blocks of size 128kb, and one free block of 512kb.

This is one of numerous cases in which the lack of type safety provided by C is actually an advantage. Although treating a given location in memory as different type of data depending on the context is a less “clean” way of programming than what higher-level languages tend

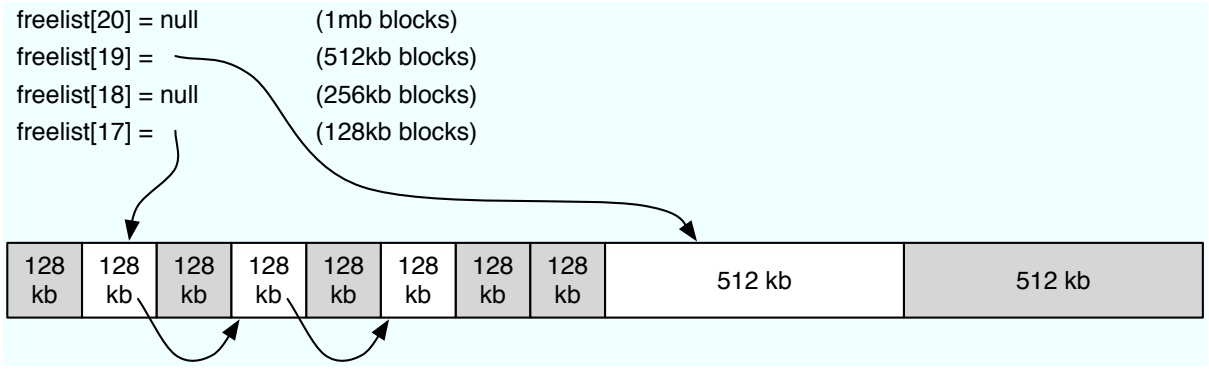

Figure 9: Free list management for buddy allocation

to enforce, it does allow optimisations that in certain cases are desirable. This style of programming can take a bit of getting used to, but once you start to think of types in C as just a convenience mechanism for accessing memory, it makes more sense. However, this approach can be more error-prone, and it is advisable to avoid using it unless there is a real need to, due to performance or other implementation concerns. The lack of a separate memory allocator, since we are implementing one, is an example of such a concern.

6.3 Memory regions

Memory allocators work with a particular region of memory, rather than all of the memory that is available. The reason is that parts of memory are used for things other than the heap - such as the stack and code segments of a process. When using virtual memory, we can pick any address range we like for the heap, and just make sure that this is mapped to physical memory upon process creation. `constants.h` in version 6 of the kernel contains a constant called `PROCESS_DATA_BASE`, specifying the base of the data segment, which we use as the start of the heap. `PROCESS_DATA_MAX` specifies the size of the heap given to each process (plus space for bookkeeping info), and thus determines the largest block size available to the buddy allocator.

It is also useful to have a heap for the kernel, so that private data structures used solely within the kernel can also be allocated dynamically. The region in memory that this is stored in should be mapped by each process, but using protection bits to make sure that it can only be accessed by the kernel. Thus, we can use this memory when inside a system call, in which the current process's page directory is still active, but not directly from within the process itself. In our kernel, we have chosen to reserve the address range 2-6mb for a 4mb kernel heap, which is also managed by buddy allocation.

The buddy allocator within our kernel, implemented in `buddy.c`, is parameterised by the address range it is managing. By this, we mean that all of the functions take a parameter specifying the range of memory, as well as references to the free lists and an additional array indicating what areas of memory are used, and what sized blocks they are part of. When calling `malloc` and `free`, these call the buddy allocation routines with a data structure that uses `PROCESS_DATA_BASE` and `PROCESS_DATA_MAX` to define the region of memory being managed. There are also kernel-specific versions of these functions, named `kmalloc` and `kfree`, which use the 2-6mb address range, and have separate free lists and memory usage maps. The latter are global to the whole system, while the process heap is specific to each process.

7 Pipes and file descriptors

> This section describes version 7 of the sample kernel. Most of the new code is in `filedesc.c` and `pipe.c`.

Any operating system needs to provide a way for processes to communicate with each other and transfer data to and from other entities, such as files and network connections. There are many different ways in which this can be implemented, but the one chosen by UNIX is the concept of *file descriptors*. These act like references to objects which can be created, manipulated, and destroyed by processes.

You'll likely be familiar with the `InputStream` and `OutputStream` classes from Java. These are abstract classes which provide a generic way for a program to read or write data from various types of streams, such as files, network connections, and the terminal. A program can be written that works in terms of these abstract classes, without needing to know what type of stream the data is being transferred through. All the program deals with are methods like `read`, `write`, and `close` - all of which work on different types of streams.

File descriptors are the UNIX equivalent of Java's input and output streams. Rather than being separated into two categories however, a file descriptor may support both reading *and* writing, although some types support only one of these operations. Just like Java's streams, the `read`, `write`, and `close` functions can be used with file descriptors. The way in which a file descriptor is created depends on its type; for example, file descriptors corresponding to files are created using the `open` system call, and those representing sockets are created using the `socket` system call.

If you've done much C programming, you'll likely be familiar with using these system calls. In this section, we'll look at how they can be implemented within a kernel. There are several different ways to implement them, but the technique we use is similar to Java's approach of using different classes which each provide their own implementations of the required methods. Since C is not an object-oriented language though, we cannot use classes as such, but we can get reasonably close to this approach by utilising structs and function pointers.

7.1 File descriptors

Under UNIX, each process has table of file descriptors associated with it. Each file descriptor is identified by a small integral value, and so this table is usually implemented as an array. File descriptors are actually *references* rather than objects, and it is possible for two or more of them to reference the same object. Thus, the file descriptor table does not contain the objects themselves, but simply pointers to them. In our sample implementation we refer to these objects as *file handles*.

When a process starts, it always has three file descriptors open. These are known as *standard input* (fd 0), *standard output* (fd 1), and *standard error* (fd 2). When a program is run from the terminal, standard input is connected to the user's keyboard, and standard output and error both refer to the screen. The UNIX shell supports I/O redirection, where these can be changed to refer to files or other processes using the standard `<`, `>`, and `|` constructs.

Figure 10 shows an example of a process which has four file descriptors open. Standard input is coming from `input.txt`, standard output and error are both going to `output.txt`, and the

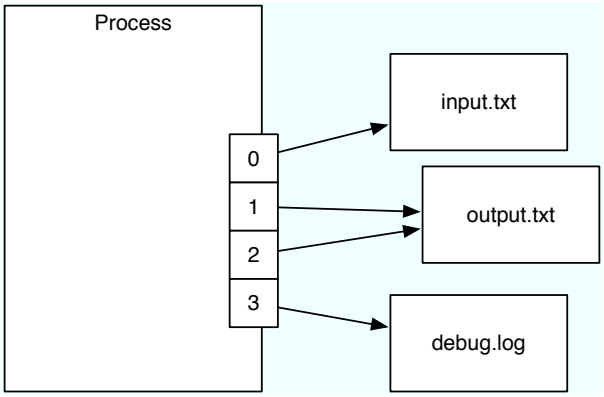

Figure 10: A process's file descriptor table

process has another file open called `debug.log` which is associated with file descriptor 3. This situation could have arisen from a command such as the following:

```
./program < input.txt >& output.txt
```

In this figure, the boxes refer not to the actual files on disk, but to *file handles*, which as mentioned above are similar to `InputStream` and `OutputStream` objects in Java. Each file handle keeps track of its position within the file, such that the next `read` or `write` call will start from that position in the file. Thus, when two file descriptors refer to a particular file handle, reading from or writing to either of them will advance the file position. For example, if the process prints one line of text to standard output, then another to standard error, then the first and second line will appear in `output.txt` in the same order that they were printed.

Version 7 of the kernel implements this mechanism, and supports three types of file handles. The first type is `FH_SCREEN`, corresponding to a write-only file handle which outputs to the screen. The other two types of handles are for pipes, which we shall discuss in Section 7.2. The `filehandle` struct defined in `kernel.h` is as follows:

```
struct filehandle {
  int type;
  int refcount;
  write_fun *write;
  read_fun *read;
  destroy_fun *destroy;
  pipe_buffer *p; /* for pipes */
};
```

The `write`, `read`, and `destroy` fields of this structure are function pointers. When a file handle is created, the constructor function assigns these fields to the functions that implement the logic for that particular type of file handle. This is where the similarity with Java's stream classes lie; these function pointers are like virtual functions that the user can override when implementing a subclass. The prototypes that these functions must match are given just above the `filehandle` definition in `kernel.h`.

An example of a constructor function is the `new_screen_handle` function in `filedesc.c`. This creates a file handle struct as above, and sets the function pointers to the `screen_*` functions defined in that same file. The main function of interest is `screen_write`; all this does is

call the `write_to_screen` function implemented in earlier versions of the kernel. This will be executed whenever a process invokes the `write` system call on a file descriptor which refers to a `filehandle` struct that was created by this function.

To see how these calls are dispatched, have a look at the implementations of the `read` and `write` system calls in `syscalls.c`. In this version of the kernel, they have been modified to work with file descriptors. They both first check that the file descriptor passed in is valid, and then call the appropriate function pointer on the corresponding `filehandle` object. This is what allows a process to use these system calls for different types of file handles, without having to care which implementation of the interface is actually in use.

In `filedesc.c`, there are some other functions for dealing with file handles. One of these is the implementation of the `close` system call, which a process invokes when it no longer needs a particular file descriptor. This removes the appropriate entry from the file descriptor table, and decrements the reference count associated with the file handle. The file handle will *only* be deleted if the reference count reaches 0, i.e. when there are no other file descriptors referencing it.

7.2 Pipes

Pipes are one of the most commonly used forms of inter-process communication under UNIX. You're probably familiar with using the UNIX shell to pipe the output of one process to the input of another, e.g.

```
cat mysongs.txt | sort
```

When you execute a command like this, the shell creates a pipe, and executes two processes. The file descriptors of the processes are set up by the shell such that standard output (fd 1) of the first process writes to the pipe, and standard input (fd 0) of the second process reads from it. Thus, instead of the output of `cat` going to the terminal, it is passed to `sort`, whose resulting output finally goes to the terminal.

Within the kernel, a pipe is a separate object that is referenced from two separate file handles. One file handle is used for write-only access to the pipe, and the other is used for read-only access. These two file handles are referenced from the relevant file descriptors of the processes. When process 1 invokes the `write` system call with fd 1, data will be written to the pipe. When process 2 invokes the `read` system call with fd 0, it will read data from the pipe. This situation is depicted in Figure 11.

Just like other types of file handles, the pipe writer and reader both have reference counts associated with them. A common use case of pipes is that a parent process creates a pipe, and then forks off a child process that uses one end of the pipe. The parent and child can thus exchange data through the pipe in a single direction. A similar case is what the shell does in the above example; a pipe is created, and two processes are forked off. In both cases, the child processes inherit the file descriptors of the parent, and thus the reference counts on the associated file handles are incremented.

In our implementation, we maintain reference counts for the file handles, but not for the pipes. Instead, pipes have two fields indicating whether they are open for reading and/or writing. When a pipe is initially created, both are set to true. When either end of the pipe is closed,

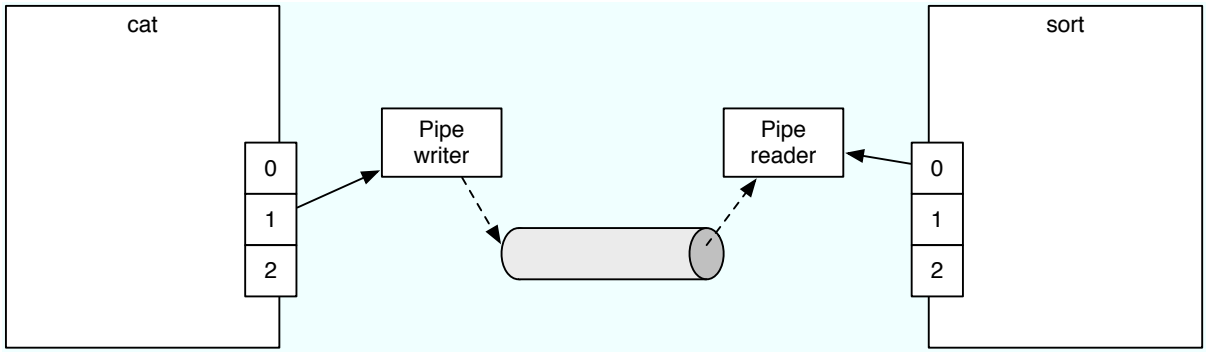

Figure 11: Two processes exchanging data via a pipe

which occurs when the writer or reader file handle is destroyed, the corresponding field is set to false. A pipe can thus be freed once both ends have been closed. Note that this scheme implies that there must only be a single pipe writer and pipe reader file handle for each pipe.

7.3 Blocking

When a process tries to read from a pipe, there may not be any data available. In this situation, there are two choices: to have `read` suspend the process until data becomes available, or to return immediately and indicate that there is no data at the present time. These two approaches are known as *blocking* and *non-blocking*, respectively.

The non-blocking approach is simpler to implement, but less convenient to use. All that the `read` function needs to do is check the amount of data in the pipe, and return immediately. However, a process that cannot continue until it is able to read some data must sit in a tight loop, continuously calling read. This eats up CPU time that could otherwise be used more productively by other processes.

Blocking is generally preferable, since the process only needs to call `read` once each time it needs some data, and can rely on the fact that the call will not return until it has a useful result. The process will be suspended until data becomes available in the pipe, or the pipe is closed by the writer. When either of these events occurs, the process is resumed, and the `read` system call will be invoked again, this time returning a result.

In the blocking case, the function that writes to the pipe must check if there is a reader blocked on it that needs to be woken up. To avoid duplicating this logic in every place where a pipe could be written to, it is implemented in the `wake_up_reader` function in `pipe.c`, which is called from both `write_to_pipe` and `pipe_writer_destroy`. As long as these two functions are the only ones used to add data to a pipe or close the writing handle, readers will always be unblocked correctly.

Under UNIX, most file descriptors support both blocking and non-blocking operation. The choice of which to use is a property of the file handle, and is set using the `fcntl` system call. For simplicity however, this is not implemented in our kernel.

7.4 Example

In version 7 of the kernel, two processes are started at boot time to demonstrate the use of a pipe to exchange data between processes. The functions `uppercase` and `number_lines` are

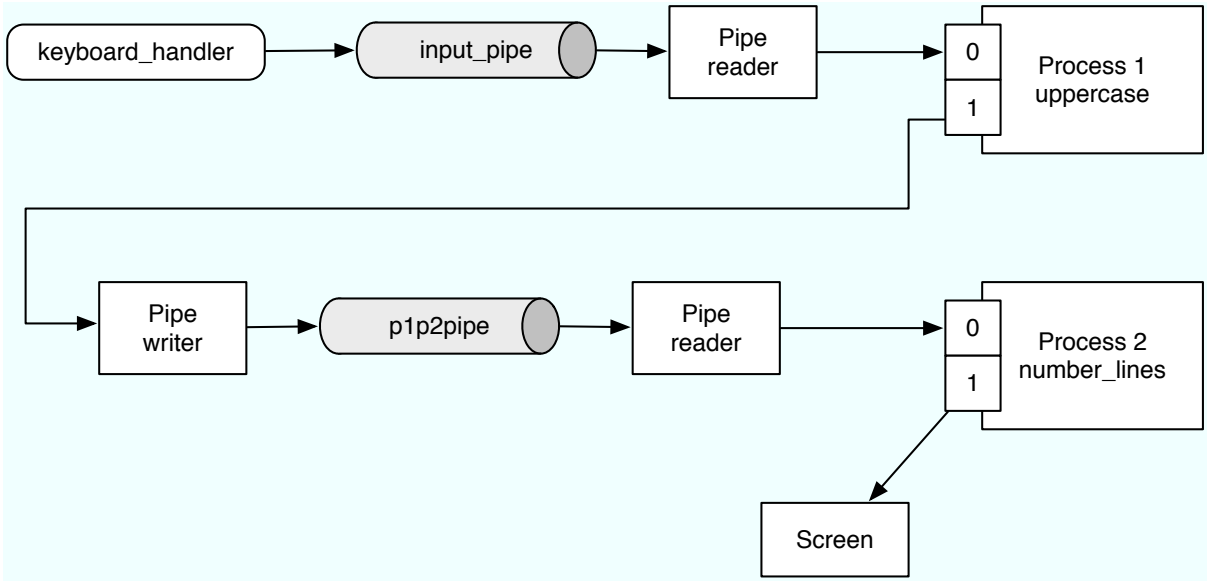

Figure 12: Process/pipe arrangement set up in `kernel_main`

both launched as processes within `kernel_main`, and a pipe is created to redirect the output of the first process to the input of the second. When these processes are run, `uppercase` reads characters from the keyboard, converts them to uppercase, and writes them to its standard output. `number_lines` places a line number at the beginning of each line. The resulting output will look like the following:

```
0 THIS IS THE FIRST LINE
1 AND HERE IS THE SECOND LINE
```

In our implementation, we handle keyboard input using a pipe. The `start_process` function has been modified in version 7 to create a pipe for each new process, and assign a reader file handle to standard input. Within `kernel_main`, the global variable `input_pipe` is assigned to the pipe created for the first process. The `keyboard_handler` function writes all characters typed by the user to this pipe, instead of outputting them to the screen as was done in previous versions. As a result, the process that reads from this pipe receives all data entered by the user. In Section 8, we shall see how this can be used to implement an interactive command line interface.

Figure 12 depicts the configuration of pipes and processes used in this example.

7.5 The `pipe` system call

Pipes can be created using the `pipe` system call. This takes a single parameter which is an array of two integers, each of which will be set by the system call to refer to the reading and writing file descriptor. Once the pipe has been created, the process can then read from the pipe using the first file descriptor, and write to it using the second.

Since the most typical use of pipes is to transfer data from one process to another, it is necessary to have a mechanism which allows a newly-created pipe to become accessible to the second process. This can be achieved using the `fork` system call, which duplicates a copy of a process. The child process created by `fork` has a copy of the parent's memory, as well as a copy of all

of its file descriptors. This means that the parent and child both reference the same set of file handles. The child thus has access to the pipe, as well as the parent.

After calling `fork`, it is usual for the parent and child processes to both close the file descriptors that they do not need. In the case of a pipe, one will close the reading end, and another will close the writing end. Which is closed by the parent and child will depend on the direction of data flow, e.g. if the parent is writing and the child is reading, the parent process will close the first file descriptor in the array, and the child will close the second. An example code snippet demonstrating the use of fork is as follows:

```
int fds[2];
pipe(fds);
if (0 == fork()) { /* child */
  close(fds[1]);
  char buf[1024];
  int r = read(fds[0],buf,1024);
  /* ... */
}
else { /* parent */
  close(fds[0]);
  write(fds[1],"Hello",5);
  /* ... */
}
```

In version 7 of the kernel, the `pipe` system call is implemented by the `syscall_pipe` function in `pipe.c`. The first thing this does is search through the file descriptor table of the calling process to locate two unused file descriptors. It then creates a pipe, as well as pipe reader and pipe writer file handles referring to it. The previously discovered free entries in the file descriptor table are then updated to point to these, and the corresponding file descriptors stored in the supplied array. After this, the pipe can be used just like any other file descriptor using the `read` and `write` system calls. The implementation of `fork` is discussed in the next section.

7.6 The `dup2` system call

Another system call that is useful in conjunction with pipes is `dup2`. This allows a process to modify its file descriptor table by setting or replacing a particular entry to contain a reference to another file handle. The two parameters to this call specify the source and destination file descriptor, and the destination entry in the file descriptor table is updated to point to the same file handle as the source. This is useful for changing the file handle associated with standard input, output, or error so that when the process writes to these it can actually read from or write to a file or pipe. The implementation of this system call is given in `filedesc.c`.

8 UNIX-style process management and filesystem access

This section describes version 8 of the sample kernel. Most of the new code is in `filesystem.c`, `filesystem.h`, `unixproc.c`. Several programs have also been added - `sh`, `ls`, `cat`, and `find` which run under the kernel, and `fstool` which runs under Linux.
See also: Tanenbaum, sections 1.3, 4.7, and 4.8

The way in which process creation has been implemented in our kernel so far via the `start_process` function, which creates an entirely new process that is independent of all others. This function takes a parameter specifying the function that the process should run. Once created, it has no way to interact with other processes, unless we add other code within the kernel to set this up. An example of such code was given in the `kernel_main` method of version 7 of the code, in which a pipe was created to link two processes together.

This technique is very limiting, for a couple of reasons. Firstly, it would be preferable to allow processes to set up their inter-process communication mechanisms like pipes with other processes directly. The use of `pipe` and `fork` discussed in Section 7.5 is an example of this, and is the most common way for processes to interact under UNIX. Thus, we should provide an implementation of the `fork` system call to enable this to happen, which will be a separate mechanism to the `start_process` function that has been used so far.

The second limitation of our current approach is that the code that a process runs must be compiled into the kernel. This is not normally practical, since most operating systems allow you to compile and run your own programs on them, without having to recompile the kernel itself. UNIX provides a system call named `execve` which instead allows you to specify the name of a program stored in the filesystem that should be run. This is how all processes are run under UNIX. It also implies that we need a filesystem.

Real filesystems can be quite complex to implement, and are outside the scope of what we can reasonably hope to cover in detail here. What we shall do instead is to provide a very minimal "filesystem" which is simply a data structure in memory that supports read-only access. This can be implemented as a normal tree data structure. The functions that we'll use to access these simply need to take a path name and use that to traverse through the tree looking for the file that matches the names within the path.

In this section, and in version 8 of the code, the only functions we'll use for accessing the filesystem are those that directly access the tree in memory. These can be used within the kernel, which has full access to memory, but cannot be used from processes due to memory protection. Your task for practical 3 is to implement a set of system calls that processes can use to access the filesystem.

8.1 Filesystem organisation

There are two problems that we need to solve in order to have something approximating a filesystem. The first is deciding how the data is to be organised, and the second is how to make that data accessible to the kernel. In this section we shall look at the first problem. For this, we'll use what is perhaps the simplest way of storing file data, which is to simply concatenate a collection of files together, with a simple indexing mechanism that allows us to find out the position of a file with a given name.

Two structures are defined in `filesystem.h` - `directory_entry` and `directory`. The first contains several fields that are used to describe a file or directory, such as its name, size, and

location. The latter refers to the offset within memory, relative to the start of the filesystem data, at which the file or directory begins. For directory entries that correspond to files, the location contains the actual contents of a file. For directory entries corresponding to directories, the location contains a `directory` structure specifying the number of entries, directly followed by an array of `directory_entry` structures. The type of an entry is given by its `type` field, which is always set to either `TYPE_DIR` or `TYPE_FILE`.

A program called `fstool` is supplied with this version of the kernel. Its job is to build up a filesystem image in memory, based on a directory hierarchy on your (real) filesystem. The above data structures are used to represent files and directories within the image, each of which is placed directly one after the other. This data is then written to a file. To build a filesystem image, run the program as follows:

```
fstool -build <imagefile> <directory>
```

8.2 Getting filesystem data into memory

Once we have a file on disk with the contents of our filesystem, we need a way to make it accessible to the kernel. One way would be to copy this data to a hard disk, and then have the kernel use the appropriate hardware interfaces to read data from the disk. Under `qemu`, we could specify that the image file should be treated as a hard disk image, making this approach feasible for our test environment. However, this approach would be a bit more complicated to implement, since it would be necessary to add functionality for reading from disk, and the logic for traversing the tree would be somewhat complex because it would need to access the data block-by-block.

A simpler alternative is to simply load the whole filesystem into memory at kernel boot time. This is feasible for small filesystems, and is commonly referred to as a *RAM disk*. The reason that this is simpler is because the kernel can just treat the data as a normal data structure which can be accessed directly from memory. The traversal logic for locating a file and accessing its contents simply needs to work directly with data that is already in memory.

The GRUB boot loader includes a feature called *modules*, which are files that are loaded into memory along with the kernel at boot time. In order to get the filesystem data into memory, it is necessary to copy the image file onto the boot disk, and to set an option in GRUB's configuration file that asks it to load this at boot. The boot disk image mentioned in Section 2.2 already has the configuration file set appropriately, and in fact there is an empty file in that disk image called `filesystem.img`. GRUB has been loading this all along, it's just that it has not contained any data and we have not been making use of it.

In this version of the code, `mkbootimage.sh` has been modified to create a filesystem image from a directory adjacent to the source directory in the hierarchy. In order to build this successfully, you will also need the `testfiles` directory supplied as part of the sample code, located in source distribution at:

http://adelaideos.sourceforge.net/

The build script copies the compiled versions of the supplied programs (mentioned in Section 8.5) to the `testfiles/bin` directory, and then uses `fstool` to create a file called `filesystem.img` with the complete contents of this directory. This is then copied to `grub.img`, so that the boot loader can load this file into memory at startup time.

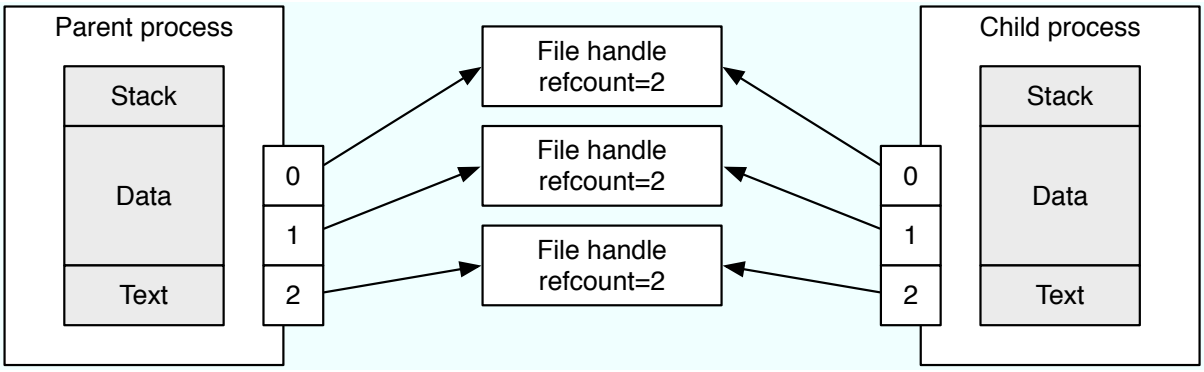

Figure 13: Parent and child process after `fork`

The `kernel_main` function in this version takes an additional parameter which contains several pieces of information passed to the kernel from GRUB. One of these is the location of the filesystem in memory, which gets loaded directly after the kernel[8]. During initialisation, this location is assigned to the global variable `filesystem_start`, which is used by other functions whenever they need to access data in the filesystem. All of the other logic necessary to locate files is implemented in `filesystem.c`. In particular, `get_directory_entry` can be used to locate a file or directory within the filesystem image, e.g. when `execve` needs to load a program into memory.

8.3 The `fork` system call

The job of `fork` is to create a copy of the calling process. The new process becomes a child of the current one, and contains an exact copy of all process state. After the call returns, both continue on from the same point, but can branch out into different code paths by inspecting the return value of `fork`. File descriptors are also copied, but the file handles aren't - they are shared. So the child process has a reference to all of the file handles that the parent process has. Figure 13 shows an example of a parent and child process after `fork` has been executed.

Version 8 of the kernel contains an implementation of `fork` in `process.c`. The first thing this does it create the child process with a unique process id, and then copies across several fields of the parent's process object. These fields include the start and end values of the text, data, and stack segments. The actual contents of these segments themselves must then be copied. This is done by creating a new page directory for the child process, which maps the same set of logical addresses, but to a distinct set of physical pages. The physical pages are an exact copy of those referenced by the parent's page directory. The only exception to this is the kernel memory, which stays the same and is identity mapped by the child process.

The file descriptor table is then copied from the parent to the child. For each of the file handles, the reference count is incremented, so that the kernel is able to keep track of the fact that there are now additional references to each of the file handles. When the parent or child process later exits, or closes any of these file descriptors, these reference counts will be decremented, and the file handles will only be freed when there are no more references to them.

[8]Since we're using a hard-coded value of 2Mb for the start of kernel private memory, problems will arise if the filesystem image goes beyond this limit, and thus only small filesystems (< 500k or so) should be used. It would be possible to get around this by having the kernel figure out where to start its primary memory region based on where the filesystem data ends.

These are the main steps that fork needs to execute. Once it returns, the parent will see a return value from the `fork` call that is equal to the child's process id. The child will see a return value of 0, which is arranged by having `fork` set the appropriate value in the child's stack segment just before exiting. After the call returns, both processes may continue execution independently, taking whatever actions they like. Often this will involve some form of interaction between the parent and child, such as transferring data via a pipe, or having the parent wait for the child to complete.

8.4 The `execve` system call

This system call causes a process to start running a different *program*, i.e. sequence of instructions. It takes a parameter specifying the filesystem path of the program, which is then used to locate the program on disk and load it into memory. The data contained in this file is mapped into the process's address space, so that it can access both the instructions and data stored in the executable file. This mechanism allows users to supply their own programs to run on an operating system, without being limited to code that is already compiled into the kernel, which is what we have been doing up until now.

In addition to loading a new program, `execve` also permits a small amount of information to be passed to the program to help with initialisation. Under most UNIX systems, this is in the form of command line arguments and environment variables, both of which are collections of strings. These usually contain configuration information and the names of files that the program should use. To simplify the implementation of `execve`, our kernel does not support environment variables, and only allows the passing of command line arguments.

The way in which these arguments are passed to the process is via the stack. Whenever a function call is made in C, the parameters are pushed onto the stack, and then the processor enters the code for the function. Thus, a function that wishes to access its parameters just needs to look at the portion of the stack that comes directly below its activation record. Since the `main` function is the first C function called in a program however, we need to set up the stack ourselves, so that it can access the parameters just like it had been called from other C code. For this reason, `execve` must build up the data for the first part of the program's stack in memory, containing all of the command line arguments passed to the call. The technique we use for doing this is very similar that of MINIX, as described in Tanenbaum section 4.7.5.

Once the arguments have been set up, `execve` completely replaces the process's text segment with the contents of the specified file. The data segment is resized to 0 bytes, because at startup the program does not use any heap memory; it can later expand this using the `brk` system call if needed. The stack segment can be re-used, except that the argument data prepared previously must be copied in to end of the stack segment so that the `main` function can access it. Finally, the saved registers for the process are re-initialised such that the instruction pointer refers to the first instruction in the program.

All programs that are launched via `exec` first begin execution in an assembly routine called `start`, defined in `crtso.s`. This performs any initialisation that needs to be done before `main` begins, such as setting up the initial heap data for `malloc`. It is this routine which calls `main`. However, it does not actually push any arguments on to the stack, because they have already been put there be `execve`. The `start` function also handles the case where the `main` function returns, by calling `exit` to terminate the process with the exit status returned from `main`.

All of the file descriptors that the process owns stay exactly the same, so the program that is run may use them to output to whatever file handles they reference, such as pipes to other processes. It is also worth noting that `exec` does not return to its caller unless it fails, since upon success the program that called `exec` no longer exists, and the new one is run instead.

8.5 Putting it all together

This last version of the kernel contains many of the essential elements of UNIX. While it is certainly a long way from doing everything that a normal UNIX system can do, the parts that we have demonstrated are perhaps the most important aspects. With a basic filesystem, the ability to launch processes via `fork` and `execve`, as well as support for pipes, you can experiment with numerous common things that can be done under UNIX.

To demonstrate how some of these features can be used, Version 8 of the source code contains a number of programs that can be run on the kernel. These include a basic shell, as well as a few basic commands such as `ls`, `cat`, and `find`. The shell is launched on startup, by having the `kernel_main` function create a process that uses `execve` to start running a shell. Once inside the shell, you can run other programs simply by typing their name. The shell looks in the `/bin` directory on the filesystem when it is asked to run a program.

You can add other programs by creating additional C source files, and adding them to the `PROGRAMS` variable in the Makefile. To have these copied on to the filesystem image with the other programs, you should also modify the `mkbootimage.sh` script in order to ensure they are copied into the bin directory. These other programs may then be launched simply by typing their name from the shell.

9 Conclusion

If you've read this far, you've hopefully gotten a reasonably good understanding of how a basic operating system kernel can be put together. What we've discussed here is just one way of going about it - there are many, many other ways, and indeed this area has been a subject of much research (and debate!) for a very long time. The point of this guide is not so much to show you the "best" way of doing things, but to demonstrate how to get started learning about the internals of operating systems, and perhaps convince you that writing or modifying one is something that is not necessarily any harder than with any other complex piece of software.

Whether or not you end up doing any operating system development in the future, it's very useful to have an understanding of how a kernel works internally. This knowledge can help you write better applications, by taking into account the interaction that occurs between user-level processes and the kernel, as well as how the kernel manages memory and other resources internally. The different layers that modern software environments are structured into each have an impact on overall system functionality and performance, and sometimes achieving particular goals requires you to modify, or at least understand, the layers below what you normally work with.

If you are interested in learning more about OS internals, the textbook for this course has a much more in-depth discussion of the issues we've covered here, as well as an implementation of a fully-fledged UNIX system, MINIX. It also contains a wide ranging discussion on a lot of

the higher-level design issues that operating system developers face, and the reasoning behind design choices in modern operating systems. You are encouraged to make use of this book, as it will help you further to understand the concepts we're teaching in this course.