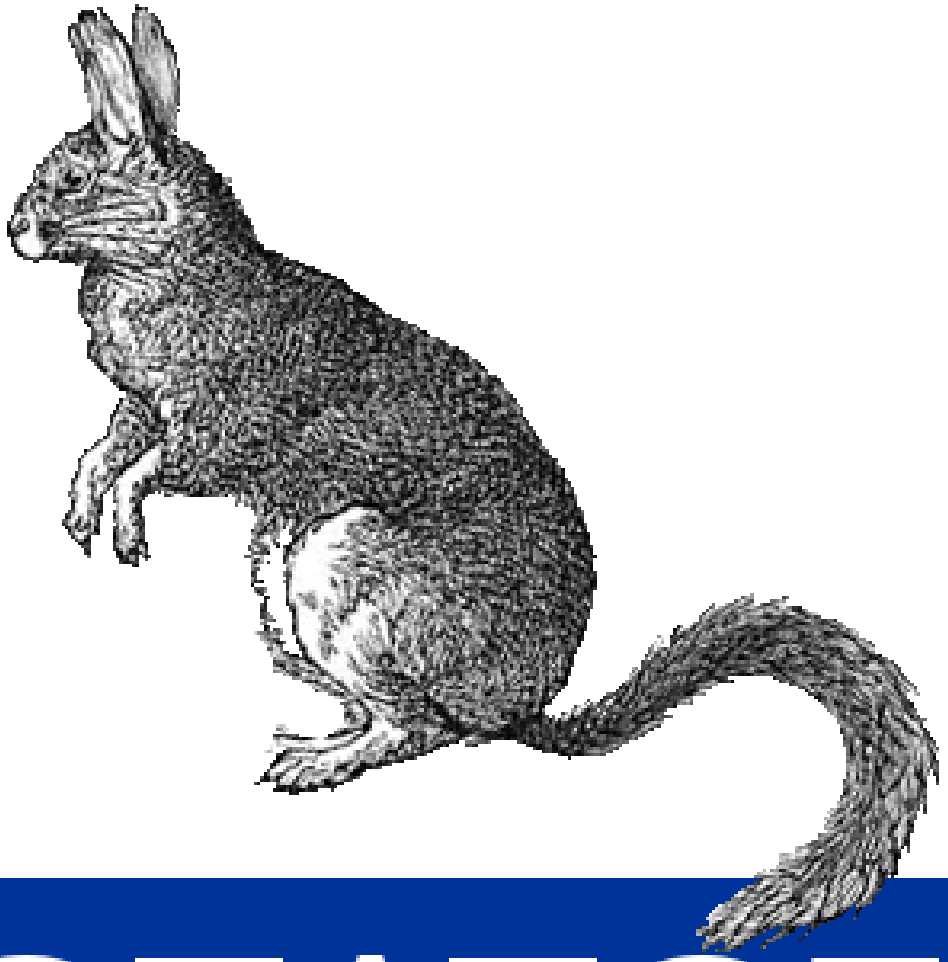
System / Development Documentation



POTATOES

In A Nutshell

D. Bader, V. Doelle, J. Schamburger, D. Traytel

Contents

1 Introduction

1.1 Motivation

POTATOES, a "Practical Oriented TeAching Tool" and "Operating (and) Educating System", aims to be a small but full fledged operating system based on the KISS [1] principle. POTATOES's goal is not to replace established and advanced operating systems. It's purpose is to provide an easy to understand and well documented way to the design and implementation of operating systems.

Moreover, *POTATOES*'s open source offers everyone the chance to use *POTATOES* as a basis for own experiments or feature implementations. Thus, one can gain valuable practical experiences without disproportional high efforts and unforeseeable problems. Especially students for instance are welcome to implement some basic concepts of operating systems in order to get a deeper understanding of their theoretical knowledge.

Finally, the *POTATOES* development team looks forward to working together with ambitious programmers and creative thinkers!

[1] "Keep It Short and Simple"

1.2 System

POTATOES is separated into four main kernel subsystems:

1. Process Management
2. Memory Management
3. Input/Output Interface
4. File System

Using a clear specification of the subsystems' interfaces and well defined communication directions, each subsystem can be maintained separately and independently from the remaining kernel components.

A schematic overview of the *POTATOES*'s kernel and user space looks as follows:

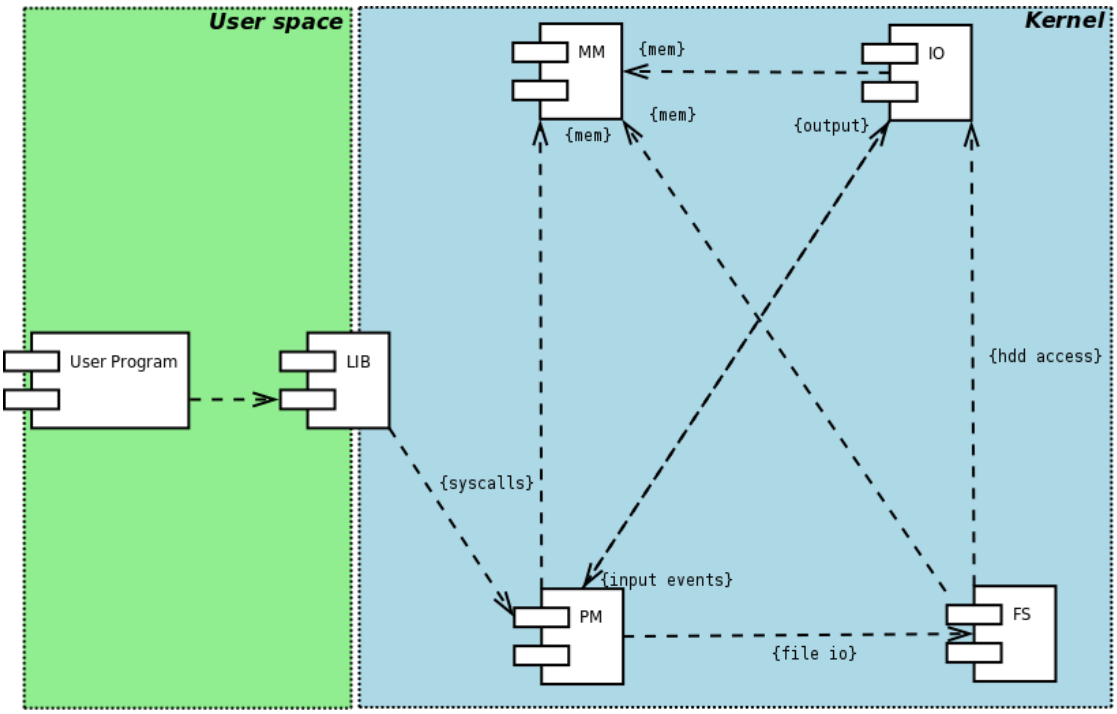


Figure 1: *POTATOES*'s system overview

2 Memory Management

An important task of the operating system is to manage the main memory. Therefore it needs a memory manager which provides possibilities to programs and the kernel to allocate and free space dynamically.

2.1 GDT

The Global Descriptor Table (GDT) is an array of GDT entries describing the segmentation system. Our boot loader *GRUB* already creates a GDT for our system, but we need to create our own one because we don't know where *GRUB* stores its GDT. Thus it could be accidentally overwritten.

Listing 1: kernel/mm/mm.h

```
1 struct gdt_entry {
2     uint16 limit_low;
3     uint16 base_low;
4     uint8  base_middle;
5     uint8  access;
6     uint8  granularity;
7     uint8  base_high;
8 } __attribute__((packed));
```

One GDT entry represents one certain segment (code or data) its start, end and the access levels. So it is possible to divide the main memory into one segment for data and another segment for code, but as we don't actually use segmentation in *POTATOES*, each segment has access to the whole address space (0x0 - 0xFFFFFFFF considering a 32bit processor). So we set the GDT entries to cover the whole address space.

2.2 Paging

Paging is used for two main goals: memory protection and virtual memory.

Virtual memory in general means that the address a program (or the kernel) gets by allocating some space is just a virtual address which is mapped on a physical one. For example an allocation call can return the address 0x00C00000 which is mapped on the physical address 0x00001200. So, each program can have an address space which starts at 0x00000000. The translation from virtual to physical address is done by the memory management unit (MMU)[2].

2.2.1 Organization of frames

In systems that use paging, the whole main memory is divided into fixed sized frames (in *POTATOES* one frame has 4 KB). To organize these frames, *POTATOES* uses a bitmap structure. With that bitmap it is quite simple to mark frames as occupied/free or to find the first free frame.

[2] `http://en.wikipedia.org/wiki/Memory_management_unit`

2.2.2 Organization of pages

To be able to write to frames, we need to have "pages" (see below), each mapped to a specific frame. For organizing these pages we need some simple structures:

The **page directory**, which mainly consists of two arrays of page tables (one for the virtual pointers and one for the physical addresses).

The **page tables** itself are just arrays of pages.

As the two structures above are very simple, the most interesting structure is the **page** itself:

Listing 2: kernel/mm/mm_paging.h

```
1  typedef struct page
2  {
3      uint32 present  : 1;    // 1: page is present in memory; 0: not
           present
4      uint32 rw       : 1;    // 1: readable and writeable; 0: read-only
5      uint32 user     : 1;    // 1: accessible from user mode; 0: only
           accessible from supervisor
6      uint32 res      : 2;    // reserved by the CPU
7      uint32 accessed : 1;    // 1: page has been accessed since last
           refresh
8      uint32 dirty    : 1;    // 1: page been written to since last refresh
9      uint32 res2     : 2;    // reserved by the CPU
10     uint32 avail    : 3;    // available for kernel use
11     uint32 frame    : 20;   // frame address (least significant 12 bits
           are 0 because each frame
12                 // is fixed at 4KB in size
13 } page_t;
```

To enable paging we simply need to set the paging bit in CR0[3] and copy the (physical!) address of our page directory to the x86 control register 3 (CR3)[4] so that the MMU knows where to find the page directory which is needed to translate the virtual addresses to physical ones.

2.2.3 Page faults

The second main ability of paging (along with virtual memory) is memory protection. So, if paging is enabled, several actions lead to a page fault:

> A **minor page fault** occurs when a program tries to access a page which is loaded in memory without the present bit is not set.
>
> A **major page fault** occurs when a program tries to access a page which is not loaded in memory (which means that the page must be loaded from the hard disc before it can be used).

[3] http://en.wikipedia.org/wiki/Control_register#CR0
[4] http://en.wikipedia.org/wiki/Control_register#CR3

An **invalid page fault** occurs when a program tries to read memory referenced by a null pointer.

A **protection fault** occurs when a program attempts invalid page accesses (for example accessing pages which do not exist, writing to read-only pages or trying to modify privileged pages).

As we don't use swapping (a technique to swap pages to the hard drive if all available frames are occupied), every existing page is loaded to the memory. So, the major page fault isn't relevant for our system.

2.2.4 Heap

Now that we have enabled paging, we need a heap to handle the memory requests from our kernel.

The kernel heap (currently the only heap in *POTATOES*) starts right after the kernel. The heap is initialized with a fixed size much smaller than the main memory size. Thus the rest of the memory can be used for heaps of kernel-external programs, for example.

The main goal of the heap is to provide the possibility to allocate and free fixed-size blocks of memory. Therefore, in *POTATOES*, the heap is represented by a doubly linked list of allocated blocks.

Each block consists of two parts: The header struct:

Listing 3: kernel/mm/mm.h

```
1 typedef struct mm_header {
2     struct mm_header *prev;
3     struct mm_header *next;
4     char name[32];
5     uint32 size;
6 } mm_header;
```

The header of a block saves pointers to the previous and next allocated block, the name of the block (mainly for debugging purposes) and, of course, the block's size. The other part is the block itself, i.e. a block of the given size, which is reserved for being written to by the one who allocated it.

Allocating: When a block shall be allocated, the allocation function searches the list until a free block of the given size is found. Then, it creates the new block and inserts it into the linked list. If the heap is "full", i.e. there is no hole which is big enough, the heap is expanded by the given size to ensure there now is an adequate hole.

Freeing: Freeing a block is quite similar to allocating: When a block shall be freed, it is removed from the list. Although the block's content is still existent, it cannot be accessed anymore.

3 Input/Output

Most of the *POTATOES* I/O-Management is handled by usage of Port Mapped I/O (PMIO). PMIO is a method to communicate with peripheral devices using some special instructions. In our case, these are above all the IN- and OUT-instructions of x86-compatible processors. Concretely that means, that after reading the next section you will fall in love with **outb** and **inb** - our C wrapper functions for the mentioned x86-assembler instructions. Memory Mapped I/O (MMIO) is also used in some subsystems, e.g. the VGA display driver. MMIO is another method to communicate with peripheral devices. The difference to PMIO is that there is no need for special instructions. The peripheral ports are mapped into RAM and can be accessed in the same way as common memory.

3.1 Monitor

POTATOES uses the standard VGA text mode. In this mode every character is represented by 16 bits as follows:

bits	15 12	11 8	7 0
meaning	background color	foreground color	ascii character

Listing 4: kernel/io/io_monitor.c

```
/**
 *  Writes a colored character to the display.
 *
 * @param ch character to be written
 * @param fg foreground-color
 * @param bg background color
 */
void monitor_cputc(char ch, uint8 fg, uint8 bg)
{
    ...
    switch(ch) {
    ...
    default:
        *disp = bg * 0x1000 + fg * 0x100 + ch; //prints the character to
            the display pointer
        disp++;
    }
}
```

In text mode the display can show 25 lines with 80 characters per line. The kernel communicates with the display through MMIO, as mentioned before. The start address of the VGA-memory is 0xB8000.

Instead of writing characters directly to this memory *POTATOES* uses the concept of virtual monitors, which will be described later. For this reason the function **monitor_cputc(char ch, uint8 fg, uint8 bg)** is almost unused in our system. The only function that prints its message directly to the display is the kernel's panic function.

It is too dangerous to involve any other subsystem in the process of printing the error message. The message might be lost if the virtual monitor subsystem itself is buggy or stuck.

3.2 Interrupts

Interrupts are signals indicating that there is a special situation, which requires a change in the execution process. There are three types of interrupts:

- **hardware interrupts** are caused by a peripheral device.
- **software interrupts** are side effects caused by faulting instructions, e.g. *division by zero*.
- **interrupts by instruction** - every x86-CPU allows the programmer to "throw" an interrupt with the INT-instruction.

3.2.1 Interrupt Descriptor Table (IDT)

The IDT is used to tell the CPU how to handle an interrupt. Every known interrupt has an IDT entry containing the address of its handling function. An IDT entry looks like this:

Listing 5: kernel/io/io_idt.c

```
1  struct idt_entry{ //16+16+8+8+16=64 bit
2      /**
3       * Lower 16 bit of the interrupt handler's code address
4       */
5      uint16 low_offset;
6
7      /**
8       * Code segment selector in the GDT.
9       */
10     uint16 selector;
11
12     /**
13      * Unused
14      */
15     uint8 separator; //always 0
16
17     /*********************************************
18      * 0_____1___________3_____________________7 *
19      * |  P  |    DPL    |   01110   | *
20      * |_____|___________|___________________| *
21      * P=segment present?              *
22      * DPL=descriptor privilege level        *
23      *********************************************/
24     uint8 flags;
25
26     /**
```

```
27        * Upper 16 bit of the interrupt handler's code address
28        */
29      uint16 high_offset;
30  }__attribute__((packed)); //gcc-flag to use 64 connected bits of memory
        for the struct
```

After every instruction the CPU checks for pending interrupts. If there is an interrupt to be handled, the CPU saves some essential registers (especially the flag register) and picks a handler function appropriate to the interrupt number from the IDT. *POTATOES* has three different interrupt handlers for every type of interrupt. Software interrupts are handled by the so called **interrupt service routines**. Hardware interrupts are also called **interrupt requests** and have an own handler, too. Besides the interrupts of these two types, there is only one other interrupt entry in the IDT of *POTATOES*. It is used for the **syscalls**. The syscall-handling-system is described in the process management section.

3.2.2 Interrupt Service Routines (ISR)

To keep *POTATOES* as simple as possible, every software interrupt causes a kernel panic. That sounds radical, but as *POTATOES* does not have any memory protection for reasons of simplicity, we have to assume that only clean, error-free code is executed anyway. There are no reasonable software interrupts handlers, because there are no expected errors in computation.

Hence every ISR saves the CPU state, writes the name of the received interrupt to the screen (for debugging issues) and starts an infinite loop holding the system. Possible software interrupts are:

- division by zero
- debug
- not maskable interrupt
- breakpoint
- overflow
- out of bounds
- invalid opcode
- no coprocessor
- double fault
- coprocessor segment overrun
- bad tss
- segment not present

- stack fault
- general protection
- page fault
- unknown interrupt
- coprocessor fault

3.2.3 Interrupt Requests (IRQ)

The programmable interrupt controller (PIC - chip 8259A) informs the CPU about incoming hardware interrupts. This chip is, as its name says, programmable. That means it can be mapped to every entry of the IDT (in *POTATOES* these are the IDT-entries 32-47). Every PIC is responsible for eight hardware device types. That is not enough, so there are 2 PICs - the master PIC responsible for the IRQs 0-7 and the slave PIC routing the IRQs 8-15. The supported IRQs are:

IRQ0 timer

IRQ1 keyboard

IRQ2 mask for slave PIC

IRQ3 com 2,4,6,8

IRQ4 com 1,3,5,7

IRQ5 ltp 2

IRQ6 floppy

IRQ7 ltp 1

IRQ8 rtc

IRQ9 vga

IRQ10 pci

IRQ11 scsi

IRQ12 ps/2

IRQ13 coprocessor

IRQ14 primary ide

IRQ15 secondary ide

When an IRQ comes in, the handler function from the IDT entry to which the PIC is mapped is called. After saving the CPU state, the interrupt is forwarded to a device-specific handler.

The master PIC needs to be reactivated after every interrupt, the slave PIC only if the last interrupt was an IRQ 8-15. The reactivation happens through a special command sent to the PIC in PMIO[5] function. *POTATOES* uses only the IRQs 0, 1, 14 and 15. This will be changed in the future.

Now we take a look what can be achieved using the IRQs.

3.3 Timer

The timer controller 8254 "fires" in adjusted intervals using IRQ0. Without its help, it would be very hard (or even impossible) to implement concurrency. But you cannot have a (good) multitasking operating system without concurrency. To implement it, the only thing we have to do from the I/O-manager point of view (thanks to the timer chip), is to initialize this chip with a sane frequency[6] (the standard adjustment is 1193180Hz - *POTATOES* uses 100Hz) and to wake the process manager [7] (in this case the scheduler) after incrementing the global time variable **ticks**, updating the virtual monitor and the date.

3.4 Keyboard

The keyboard controller 8042 "fires" every time a key is pressed or released. At the same time it writes the keycode and the keystate to its 8 bit buffer, so it can be read via **inb**. The complex processing of the keycode takes place in the keyboard-handler[8] - there are several checks for shortcuts and particular keys. "Normal" keys are forwarded to the process manager, which writes them into the **stdin** of the active process.

The disadvantage of providing only "normal" keys to the process manager, is the fact that an application will never get to know about an incoming control key. That is why the handler represents the state of all keys in an array[9], through which the devices subsystem (i.e. **/dev/keyboard**) has raw keyboard access.

The dullest and most annoying thing in implementing the keyboard handler is writing the keycode-to-key map. A good advice is to use a ready-made one. We did not do it... and spent hours to find out which keycode belongs to which key.

3.5 Virtual monitors

The idea of the virtual monitors is to use a larger allocated piece of memory as a bigger display, but only a small movable pane (24-lines) is really visible.

A single virtual monitor is represented by the following structure:

[5] see kernel/io/int_irq.c the *void reactivate_pic(bool slave)*
[6] see kernel/io/io_timer.c function *void timer_init(sint32 freq)*
[7] see kernel/io/io_timer.c function *uint32 timer_handler(uint32 context)*
[8] kernel/io/io_keyboard.c
[9] bool keyboard_state[256];

Listing 6: kernel/io/io_virtual.h

```
typedef struct {
    /**
     * Pointer to the start of the allocated memory
     */
    uint16 *begin;

    /**
     * Size of the allocated memory
     */
    uint32 size;

    /**
     * Pointer to the start of the visible pane
     */
    uint16 *vis_begin;

    /**
     * Position of the cursor on the visible pane
     */
    uint32 offset;

    /**
     * Number of lines beneath the visible pane
     */
    uint32 scrolldown_limit;

    /**
     * Number of lines above the visible pane
     */
    uint32 scrollup_limit;

    /**
     * For framebuffer access. If this is false, the virtual monitor will
         not be painted.
     */
    bool disable_refresh;

    /**
     * The PID of the process that owns this virtual monitor
     */
    uint32 pid;
}virt_monitor;
```

The actual writing to the VGA-memory takes place in the function **update_virt_monitor(virt_monitor *vm)**[10]. The headline of the virtual monitor is also written to the VGA-memory in this function.

[10] kernel/io/io_virtual.c

3.6 Hard Disk Drive

The hard disk drive is probably the most interesting port-mapped device to communicate with. Hard disk drives are block devices - the smallest entity of data is a sector (often 512 bytes). *POTATOES* uses programmed-I/O (PIO) for this communication. That means, that in contrast to direct memory access (DMA), the CPU is used to get and send every single piece of data to the controller ports. This is probably not the most efficient method, but it goes with the *POTATOES* simplicity philosophy. Using PIO means that we do not have to deal with more and more complex (but also for the most part backwards compatible) hard disk drive communication protocols, but can use the first and simplest protocol: **AT Attachment Interface for Disk Drives**[11] (short ATA-1). To keep it simple we do not even need all parts of the protocol. We will now take a look at two types of ATA PIO commands used in *POTATOES*.

3.6.1 Data-in commands

Executing these commands is divided into five steps:

1. Write parameters to the ports.
2. Send the command to the controller.
3. Wait for the controller to fill its buffer with data (busy flag is set while the controller is not ready, some controllers also send an interrupt when they are ready).
4. When the controller is ready, read the status port, check for errors and read the data from the buffer (as the smallest amount of data is one sector (512 bytes) it would be kind of nasty to write loops for at least 512 inb-calls - fortunately, Intel x86-processors are CISC processors, so there is a nice assembler instruction for this purpose: **repinsw** - with it the read-buffer-procedure is only 6 assembler instructions long[12]).
5. When more data is expected than read, go back to step 3.

The data-in commands used in *POTATOES* are *IDENTIFY DRIVE* and *READ SECTOR(S) (W/RETRY)*

3.6.2 Data-out commands

Again five steps:

1. Write parameters to the ports.
2. Send the command to the controller.

[11] http://www.t13.org/Documents/UploadedDocuments/project/d0791r4c-ATA-1.pdf
[12] kernel/io/io_util.s

3. Write one sector of data to the controller's buffer (again there is an x86-instruction for this: **repoutsw**). When done the controller sets the busy flag and starts working.

4. Wait the controller to finish its job.

5. When more data is required than written, go back to step 3.

Here *POTATOES* uses only the *WRITE SECTOR(S) (W/RETRY)* command.

3.6.3 Implementation in POTATOES

POTATOES communicates with the hard disk drive only through single sector transfers. Our hard disk driver provides two simple functions to access the hard disk:

- **void hd_write_sector(uint32 dest, void *src)**
- **void hd_read_sector(void *dest, uint32 src)**

To provide an example, let us take a closer look at the implementation of the hd_read_sector-function[13].

The driver translates the given logical address (*uint32 src*) to a cylinder-head-sector(CHS)-address, which can be loaded directly to the hard disk drive controller ports:

Listing 7: kernel/io/io_harddisk.c

```
1     struct address addr = itoaddr(src);
2
3     outb(HDBASE + HDREG_COUNT, 1);
4     outb(HDBASE + HDREG_SEC, addr.sector);
5     outb(HDBASE + HDREG_CYL_LOW, addr.cyl);
6     outb(HDBASE + HDREG_CYL_HIGH, addr.cyl >> 16);
7     select_masterdrive(addr.head);
```

Now, according to the protocol, the command can be sent:

Listing 8: kernel/io/io_harddisk.c

```
1     outb(HDBASE + HDREG_CMD, HDCMD_READ); //read sector
```

The controller starts working, we have to wait:

Listing 9: kernel/io/io_harddisk.c

```
1     wait_on_hd_interrupt("read");
```

Because not every hard disk drive controller sends an interrupt, when its buffer is ready to be read from, the **void wait_on_hd_interrupt(char* str)** function is implemented as a mix of waiting for an IRQ14 or IRQ15 and polling the controller's status ports for the busy flag.

When the controller is ready, we can read data from its buffer:

[13] kernel/io/io_harddisk.c

Listing 10: kernel/io/io_harddisk.c

```
1       repinsw(HDBASE + HDREG_DATA, dest, 256); //read buffer
```

Because *POTATOES* only makes single sector transfers, we are done now - it was not that hard, was it? The write commands are handled in the same way. Even easier is the identification of the hard disk drive geometry and features. After sending the *IDENTIFY DRIVE* command, all you have to do is to wait for the controller to provide this information and to read the data (one sector) from the buffer. How to interpret this data is described in the ATA-1 protocol.

3.7 Real Time Clock (RTC)

The communication with the real time clock controller is easy: you send a request for what you want to read (weekday/day/month/year/hour/minute/second) to its command port with **outb** - then you can read the requested information via **inb** from its data port. The function **void rtc_update()**[14], which is called in the timer-handler, gets all these data from the RTC-controller and saves them in the global time struct:

Listing 11: kernel/io/io_rtc.h

```
1  /**
2   * Global time struct.
3   */
4  struct time {
5      uint8 sec;
6      uint8 min;
7      uint8 hour;
8      uint8 weekday;
9      uint8 day;
10     uint8 month;
11     uint8 year;
12 }time;
```

This struct is printed (converted to a string) on the virtual monitor's headline on updating, so *POTATOES* has a timepiece integrated into the virtual monitor.

[14] kernel/io/io_rtc.c

4 Process Management

A key concept in all operating systems is the process, which basically is a program in execution. A running program consists of its executable code, allocated resources (e.g. file descriptors and memory blocks) as well as its processor state or *context*.

4.1 Processes

POTATOES' process control block is defined in **pm_main.h**:

Listing 12: /kernel/pm/pm_main.h

```
typedef struct process_t {
    char *name; // a readable name string
    uint32 pid; // process id
    uint8 state; // process state: running, dead, ...
    uint32 context; // the memory area which constitutes the stack
    void *stack_start; // pointer to the beginning of the stack
    void *addr; // memory address
    ring_fifo *stdin; // STDIN queue
    proc_file pft[NUM_PROC_FILES]; // process file table
    virt_monitor *vmonitor; // the virtual monitor
    struct process_t *next; // linked list next ptr
} process_t;
```

New processes can be spawned by calling the **uint32 pm_create_thread(char *name, void (*entry)(), uint32 stacksize)** function.

The first process that gets ever created is always the kernel task and has a process id or *PID* of zero. With every new process that is spawned the *PID* is simply incremented by one.

4.2 Multitasking

POTATOES supports preemptive multitasking with a (theoretically) unlimited number of processes. The function **pm_sched()** implements a simple round-robin task scheduler. Every 10 milliseconds the timer chip raises an interrupt which causes the kernel to switch tasks. Let us have a look at first part of the main irq service routine **irq_common_stub**:

Listing 13: io/int_interrupt.s - Part 1

```
irq_common_stub:
pushad                  ; Push EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX
                        ; (32 bytes)

push ds                 ; Push the segment registers (another 16 bytes)
push es
push fs
push gs

mov eax, esp            ; Push the stack pointer (ESP) without changing it.
push eax                ; pm_schedule() needs this to save the context
```

```
12
13 push dword [esp+52]; Push the interrupt number our irqX function gave us
14
15 call irq_handler
```

The context information gets forwarded to **uint32 irq_handler(uint32 int_no, uint32 context)** and then to **uint32 timer_handler(uint32 context)** where the call to **pm_schedule** finally happens.

In this function all zombie processes up to the first valid process are removed from the scheduling queue. A *zombie process* is a process which has the *PSTATE* flag *PSTATE_DEAD*. After a valid (i.e. one that can be executed) process was found, we return its context back to the irq assembler routine:

Listing 14: io/int_interrupt.s - Part 2

```
1  mov esp, eax        ; irq_handler() returns the address of the new context
       ,
2                      ; ie stack pointer. Load ESP with that new value.
3
4  pop gs              ; Restore the segment registers (from the new context)
5  pop fs
6  pop es
7  pop ds
8
9  popad               ; Restore all general purpose registers
10
11 add esp, 8          ; Remove interrupt number and error code
12                     ; from the stack
13
14 iret                ; IRET will also reenable interrupts because EFLAGS
15                     ; will be restored to its state before the interrupt.
```

Now we have the context of the new process setup and can resume its execution.

4.3 System calls

In order to do anything remotely useful programs need access to the various resources the machine (or the operating system) provides.

A text editor, for example, needs read and write access to files, it has to display text on the screen and retrieve user input via the keyboard.

For security purposes and being able to run multiple programs concurrently, we cannot allow processes to directly access the machine's hardware. Also we want our program to run on a multitude of machines without having to make code changes.

We solve this problem by providing an abstraction layer above the actual hardware: the *system calls* or *syscalls* for short. From a user's perspective a system call is nothing more than a simple function call. Let us take a look at the simplest syscall there is, the **_log()** syscall:

Listing 15: syscalls_cli.c

```
1 /**
```

```
 * Writes a string to the kernel debug monitor. Useful to dump
 * strings that should not be displayed in the process's own
 * vmonitor.
 *
 * @param msg the text to print
 */
void _log(char* msg)
{
    _syscall(SYS_LOG, msg);
}
```

The **_syscall()** function is an assembler routine defined in **int_interrupts.s** which takes two arguments: a *syscall id number* and a *data pointer*. **_syscall()** leaves these on the stack and then raises *interrupt* 0x42 to switch into kernel mode.

Incoming syscalls are handled by the kernel via the **incoming_syscall()** (in *int_interrupts.s*) and **pm_syscall()** (in *pm_syscalls.c*) functions.

Listing 16: pm_syscalls.c

```
/**
 * The syscall dispatch function. This gets called whenever a thread
 * request a syscall by raising the syscall interrupt (int 0x42).
 * It checks the syscall id for validity and calls the appropriate
 * syscall handler.
 * @see syscall_table
 * @see syscalls_shared.h
 *
 * @param id the syscall id number
 * @param data pointer to the syscall argument structure
 */
void pm_syscall(uint32 id, void* data)
{
    if (id > MAX_SYSCALL) {
        panic("pm_syscall: id > MAX_SYSCALL");
    }
    syscall_table[id](data);
}
```

After a brief validity check **pm_syscall()** dispatches the system call to its *handler function* by looking it up in the **syscall_table** jump table.

Now, **_log()**'s handler function **sys_log()** (all kernel side syscall handler functions have a *sys_* prefix) can finally do some real work:

Listing 17: pm_syscalls.c

```
/**
 * void log(char *msg)
 * Prints debug logging output to the console.
 * @param data arguments
 */
void sys_log(void *data)
{
    SYSCALL_TRACE("SYS_LOG('%s')\n", data);
```

```
9       puts((char*)data);
10  }
```

Because **_log()** only requires a single argument we do not have to define a *syscall argument structure*. Take a look at **syscalls_shared.h** for more complex examples.

Return values In order to return values to the calling program syscalls can designate fields for that purpose in their *data pointer structure*:

Listing 18: syscalls_shared.h

```
1  /** Arguments for the OPEN syscall. */
2  typedef struct sc_open_args_t {
3      int fd; // return value
4      char *path;
5      int oflag;
6      int mode;
7  } sc_open_args_t;
```

The client side syscall stub **_open()** allocates a data pointer structure, fills the required argument fields and then hands the structure pointer to **_syscall()**. The actual syscall implementation retrieves the arguments and fills in the return value(s):

Listing 19: syscalls_cli.h

```
1  /**
2   * Opens a file or a device.
3   *
4   * @param path the path of the file to open
5   * @param oflag the open flag. @see O_OPEN @see O_CREAT
6   * @param mode not used as of now. Set to 0.
7   * @return a valid handle on success or -1 if failed
8   */
9  int _open(char *path, int oflag, int mode)
10 {
11     sc_open_args_t args;
12     args.path = path;
13     args.oflag = oflag;
14     args.mode = mode;
15
16     _syscall(SYS_OPEN, &args);
17
18     return args.fd;
19 }
```

4.4 Devices

In order to provide a clean syscall interface *POTATOES* supports device files or *devices* for short. A device is a virtual resource that implements the following five operations:

1. **open(char *path, int oflag, int mode)**

2. **close(int fd)**
3. **read(int fd, void *buf, int size)**
4. **write(int fd, void *buf, int size)**
5. **seek(int fd, int offset, int whence)**

Most resources can be modeled this way. *POTATOES* currently provides the following devices by default:

4.4.1 null - The NULL device

The NULL device is simply a dummy device that can be written to or read from. Everything written to it is simply discarded and the *read* operation returns a buffer filled with zeros. Yet the NULL device has its uses. For example its code serves well as a template for new devices you want to implement.

4.4.2 stdin - The Standard Input device

The STDIN device. Provides a process with access to its character input queue. After a keypress an interrupt is generated which gets handled by the I/O code. In io_keyboard.c the keyboard scancode gets converted into an ASCII character which is then given to **pm_handle_input()**. **pm_handle_input()** then writes the new character to the focussed process' stdin queue. The respective process can then choose to read its stdin queue at any time via a call to **read(STDIN, &buf, len)**.

4.4.3 stdout - The Standard Output device

The STDOUT device. Allows a process to write text to the vmonitor assigned to it.

4.4.4 framebuffer - The Framebuffer device

The framebuffer device provides a *80x25 pixels, 16 color* video screen. Video output through the framebuffer device is much faster as it writes directly to the video memory and does not use the conversion functions of the vmonitors. Used primarily for the various demo games.

4.4.5 keyboard - The Keyboard device

The keyboard device provides direct access to the state of all keys. It can be used to retrieve the state of keys which do not produce ASCII characters. For example, this mechanism is used by the Pong and Snake games to read the state of the cursor keys. This information would otherwise not be available because hitting these keys does not modify the *STDIN* queue. Obviously, you cannot write to the device.

4.4.6 clock - The Clock device

Reading from the clock device returns a 24 character long string containing the current time and date.

4.4.7 brainfuck - The Brainfuck Interpreter device

Interprets code written in the brainfuck language. Try executing a brainfuck program by typing **cp [program.bf] /dev/brainfuck** in the shell.

5 File System

The *POTATOES* file system is a logical, self-contained entity that can be applied to any block device, such as hard disks or floppy disks. It can be modified, experimented with, and tested almost completely independently of the rest of the operating system.

5.1 Layout

The file system's layout consists of four components on disk and four components in memory.

5.1.1 Disk

boot block	super block	block bitmap	data blocks ... (512 bytes per block)

5.1.2 Memory

- some block caches with specific characteristics
- the inode table
- the file table
- the process file tables

5.2 Components

5.2.1 Boot block

The disk's structure starts with a boot block containing necessary information to start the operating system with the help of an appropriate boot loader. After loading the operating system, the file system does not need the boot block for further operations.

5.2.2 Super block

The superblock[15] contains information regarding the layout, the 8 components and the file system itself:

[15] kernel/fs/fs_super.h

SUPER BLOCK
HD size (number of blocks)
number of blocks used by the block bitmap
block number of first data block
max. file size
*** in memory only ***
pointer to block bitmap
pointer to root inode
timestamp of last modification
read-only flag
dirty flag
magic number

Listing 20: kernel/fs/fs_super.h

```
1  struct super_block {
2          block_nr s_HD_size;
3          uint16  s_bmap_blocks;
4          block_nr s_first_data_block;
5          uint32 s_max_file_size;
6
7          uint8 *s_bmap;
8          m_inode *s_iroot;
9          time_t s_modify_ts;
10         bool s_read_only;
11         uint16 s_dirt;
12         uint32 s_magic_number;
13 };
```

5.2.3 Block bitmap

The block bitmap's[16] purpose is to tell the file system which block on disk is used and which not. Therefore, the whole disk is "mapped" on the block bitmap, each block corresponding to one bit, whereas TRUE equals "used" and FALSE equals "free". When the system is booted, the block bitmap needs to be loaded to RAM to ensure an adequate search performance.

5.2.4 Inodes

According to the philosophy "everything is a file", the *POTATOES* file system treats files and directories equally. Each file consists of an index node (inode) - describing the file's structure and meta information (e.g. mode, modify_ts, ...) - and data blocks arbitrarily distributed over the disk. The structure is defined by direct, single indirect and double indirect pointers "spanning up" the actual data tree.

[16] kernel/fs/fs_bmap.h

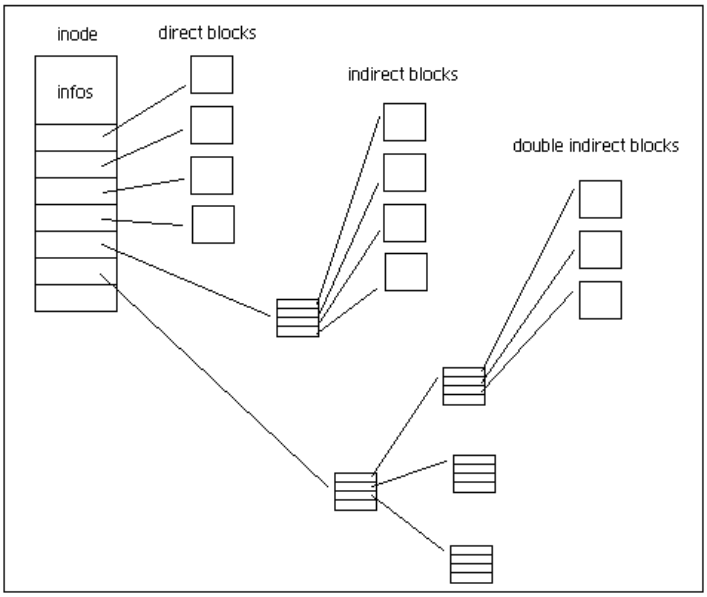


Figure 2: The inode concept

The file system's inode (on disk) looks as follows:

INODE
mode (DATA_FILE or DIRECTORY)
size (DATA_FILE: #bytes; DIRECTORY: #entries)
timestamp of creation
timestamp of last modification
pointer to 1st data block ... pointer to 30th data block
single indirect pointer
double indirect pointer

Listing 21: kernel/fs/fs_types.h

```
1 typedef struct d_inode {
2         uint16 i_mode;
3         uint32 i_size;
4         time_t i_create_ts;
5         time_t i_modify_ts;
6         block_nr i_direct_pointer[NUM_DIRECT_POINTER];
7         block_nr i_single_indirect_pointer;
8         block_nr i_double_indirect_pointer;
9 };
```

5.2.5 Inode table

When a file is opened, its inode is located and brought into the inode table [17] in memory, where it remains until the file is closed. The inodes in memory have two additional fields, "number" (the inode's ID) and "block address" (the inode's location on disk), being necessary to handle the inode in the inode table or to effectively write it back to disk.

Listing 22: kernel/fs/fs_types.h

```
typedef struct m_inode {
        inode_nr i_num;
        block_nr i_adr;
        uint16 i_mode;
        uint32 i_size;
        time_t i_create_ts;
        time_t i_modify_ts;
        block_nr i_direct_pointer[NUM_DIRECT_POINTER];
        block_nr i_single_indirect_pointer;
        block_nr i_double_indirect_pointer;
};
```

5.2.6 Block caches

POTATOES' file system uses some single block caches [18] to make the handling of read-/write processes easier. Mostly, they are used as temporary storages providing the last handled block with a specific format (e.g. address list, memory inode/disk inode, plain data, ...) and offering some meta information.

Listing 23: kernel/fs/fs_types.h

```
typedef struct block_buffer {
        block_nr block_nr;
        uint8 cache[BLOCK_SIZE];
};
```

Finally, there are 5 different types of block caches:

1. read_cache
2. write_cache
3. addr_cache (containing 32 bit block addresses)
4. d_inode_cache (containing a disk inode)
5. m_inode_cache (containing a memory inode)

[17] kernel/fs/inode_table.h
[18] kernel/fs/buf.h

5.2.7 Files

After a requested inode was successfully fetched from disk, a new "file" is created corresponding to the inode and brought into the file table[19]. The file then builds a bridge between the file handles offered to processes (= process files) and the inodes stored in memory. A file's structure looks like that:

FILE
global file descriptor
pointer to corresponding inode
file name (absolute path)
file mode (see "i_node")
number of opened links

Listing 24: kernel/fs/fs_types.h

```
1 typedef struct file {
2         file_nr f_desc;
3         m_inode *f_inode;
4         char *f_name;
5         uint8 f_mode;
6         uint16 f_count;
7 };
```

The purpose of a "process file" is to store meta information, e.g. current position within a file, as well as to be an interface between the operating system and the file system. Each process has to create and hold an own process file table managed by the file system. Once there are all process files closed, the file and the inode will be dropped in order to clear up and save memory.

PROCESS FILE
process file descriptor
pointer to global file descriptor
file position

Listing 25: kernel/fs/fs_types.h

```
1 typedef struct proc_file {
2         file_nr pf_desc;
3         file_nr pf_f_desc;
4         uint32 pf_pos;
5 };
```

[19] kernel/fs/file_table.h

5.2.8 Directories

Last but not least, the following figure depicts a single directory entry. Although being a "simple" file, a directory differs in formating its data/entries as follows:

DIR ENTRY
block number of file's inode
file name

Listing 26: kernel/fs/fs_types.h

```
1 typedef struct dir_entry {
2         block_nr inode;
3         char name[NAME_SIZE];
4 }
```

6 Applications

For now every process in *POTATOES* has an own virtual monitor. The focused process is the process with the presently active virtual monitor. You can switch the focus by using the shotrcuts **Ctrl + '+'** and **Ctrl + '-'**. These shortcuts are only setting the next/previous virtual monitor as the new active monitor.

6.1 Shell

The *POTATOES* shell is the main point of interaction between the user and the system. After boot up the kernel launches one shell instance as a separate process. You can start additional instances in separate vmonitors by pressing **Ctrl + TAB**.

The shell's user interface consists of a prompt that displays the *current working directory* (which can be changed using the **cd** command). Commands can take (among other things) *relative* as well as *absolute* paths:

```
1 cat /dev/clock
```

is the same as:

```
1 cd /dev
2 cat clock
```

To get a list of available commands enter **cmdlist** and hit return. Another nice feature is *tab-completion* of commands. Simply type the first few letters of a longer command and hit the **TAB** key to have the shell fill in the rest of the command.

Behind the curtain the shell uses a simple parser to split command strings into arguments (**void shell_handle_command(char *cmd)** in shell_main.c). Commands are implemented in the source file shell_cmds.c. Take note that any new commands you implement become available only if you add their function pointer to **struct shell_cmd_t shell_cmds**[].

6.2 Editor

The embedded *POTATOES* editor is called **speed**. That is an acronym for **S**imple **P**OTATOES/**E**tiOS **ED**itor (etiOS is the former project name of *POTATOES*). The editor is not only called simple - it is simple! Speed starts (as every process) an own virtual monitor. The string you see on the editor's monitor while execution will be saved in the opened file on **ESCAPE**. For now speed only supports the control input of **ENTER**, **TAB** and **BACKSPACE**. In future other features as movement through the data-string using arrow-keys will be added.

6.3 Snapshot

Snapshot is a small tool for taking and viewing screenshots in *POTATOES*. The functioning of this tool is pretty easy - it just copies the current content of the video memory

into a data file. The capture is initialized on the shortcut **Ctrl + s**. You will be instantly prompted to enter a filename for the new screenshot. To view a screenshot file there is a shell command called **view**. This command takes control on the active virtual monitor showing only the content of the screenshot file until **ESCAPE** is pressed.

6.4 Games

No operating system is complete without any game to play. Also, interactive games require a lot more resources in comparison to most applications, which makes them an ideal stability and performance test. They are nice to play, as well.

6.4.1 Pong

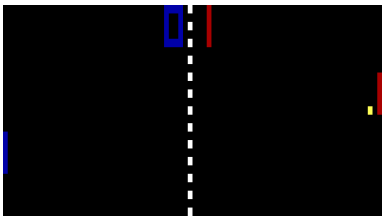


Figure 3: Pong game

Pong is a simplistic game from the old age of computing, popularized by the American game console manufacturer Atari.

POTATOES contains an implementation of Pong that is also multiplayer capable. You can launch the game by running the **pong** or **pong -2p** command in the shell. Pong's need for fast access to user input as well as high video refresh rate led to the addition of the *keyboard* and *framebuffer* device to *POTATOES*. Also, it is cool and popular with little girls, i.e. my cousin.

6.4.2 Snake

An other game implemented in *POTATOES* is the classic Snake game. You can launch this game by running **snake** for single- and **snake -2p** for multiplayer mode.

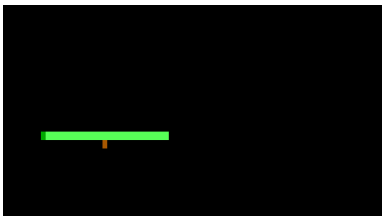


Figure 4: Snake game

Snake uses almost the same resources as Pong, in particular the keyboard and the framebuffer device. The most interesting characteristic in the implementation is the reuse of the FIFO-queue developed for the virtual monitors and the devices subsystem. The idea is to represent the snake itself as a FIFO-queue. That makes it really easy to move it on the screen just taking the endmost pixel from the queue and adding a new pixel as the new head of the snake.

6.5 Brainfuck interpreter

As an special delicacy *POTATOES* has an own **brainfuck interpreter**. Brainfuck is an esoteric turing-complete programming language, consisting of only eight commands[20]. The corresponding shell command is **bf** [**filename**]. It executes the given file which should contain a brainfuck program.

[20] `http://en.wikipedia.org/wiki/Brainfuck`

That is the functionality every interpreter will provide. The brainfuck interpreter in *POTATOES* can do even more. It can interpret single brainfuck commands on the fly. That made it possible to create an interactive brainfuck mode for our shell. It can be launched with the command **bf -i** [**sequence**] where [sequence] is an arbitrary sequence of brainfuck command e.g. the simple sequence **,.[,.]** is a short echo program in brainfuck. Because of the interactivity we can split this example sequence into single commands:

```
1 bf -i ,
2 bf -i .
3 bf -i [
4 bf -i ,
5 bf -i .
6 bf -i ]
```

Executing these commands one after another results in the same echo program as interpreting the whole sequence at once.