```
#include "console.h"
#include "io.h"
#include "string.h"
#include "stdarg.h"

typedef enum
{
    COLOUR_BLACK = 0,
    COLOUR_BLUE = 1,
    COLOUR_GREEN = 2,
    COLOUR_CYAN = 3,
    COLOUR_RED = 4,
    COLOUR_PINK = 5,
    COLOUR_BROWN = 6,
    COLOUR_LIGHT_GREY = 7,
    COLOUR_DARK_GREY = 8,
    COLOUR_LIGHT_BLUE = 9,
    COLOUR_LIGHT_GREEN = 10,
    COLOUR_LIGHT_CYAN = 11,
    COLOUR_LIGHT_RED = 12,
    COLOUR_LIGHT_PINK = 13,
    COLOUR_YELLOW = 14,
    COLOUR_WHITE = 15,
}
colour_t;

static uint8_t make_attr(colour_t background, colour_t foreground)
{
    return (background << 4) | foreground;
}

typedef struct
{
    uint8_t character;
    uint8_t attr;
}
vchar_t;

static int x, y;

static const int width = 80, height = 25;

static vchar_t *const vram = (void *)0xb8000;

static uint16_t base_vga_port;

void console_init()
{
    // read base vga port from bios data area
    base_vga_port = *(uint16_t *) 0x463;

    memset16(vram, make_attr(COLOUR_BLACK, COLOUR_LIGHT_GREY) << 8,
             width * height);
    memset16(vram + width * height,
             (make_attr(COLOUR_RED, COLOUR_WHITE) << 8) | 'X', width);
}

static void scroll_up()
{
    memcpy(vram, vram + width, width * (height - 1) * 2);

    uint16_t empty_attr = make_attr(COLOUR_BLACK, COLOUR_LIGHT_GREY) << 8;
    memset16(vram + width * (height - 1), empty_attr, width);
}

static void newline()
{
    x = 0;

    if (++y == height)
    {
        scroll_up();
```



```
        y = height - 1;
    }
}

static void update_cursor()
{
    uint16_t pos = y * width + x;

    outb(base_vga_port, 0x0e);
    outb(base_vga_port + 1, (pos >> 8) & 0xff);

    outb(base_vga_port, 0x0f);
    outb(base_vga_port + 1, pos & 0xff);
}

static void putc(char c)
{
    if (c == '\r')
    {
        x = 0;
        return;
    }

    if (c == '\n')
    {
        newline();
        return;
    }

    vram[y * width + x].attr = COLOUR_LIGHT_GREY;
    vram[y * width + x].character = c;

    if (++x == width)
    {
        newline();
    }
}

void console_puts(const char *str, uint32_t len)
{
    while (len--)
    {
        putc(*str++);
    }

    update_cursor();
}

static void itoa(int n, char *out)
{
    int len = 0, negative = 0;

    if (n == 0)
    {
        out[len++] = '0';
        out[len++] = 0;
        return;
    }

    if (n < 0)
    {
        negative = 1;
        n *= -1;
    }

    while (n)
    {
        out[len++] = '0' + (n % 10);
        n /= 10;
    }

    if (negative)
```

```
    {
        out[len++] = '-';
    }

    out[len] = 0;

    for (int i = 0; i < len / 2; i++)
    {
        int j = len - i - 1;
        char c = out[j];
        out[j] = out[i];
        out[i] = c;
    }
}

static char hex_dig(uint32_t dig)
{
    if (dig <= 9)
    {
        return '0' + dig;
    }
    else if (dig >= 10 && dig <= 15)
    {
        return 'a' + dig - 10;
    }
    return '?';
}

static void utox(uint32_t u, char *out)
{
    out[0] = hex_dig((u >> 28) & 0xf);
    out[1] = hex_dig((u >> 24) & 0xf);
    out[2] = hex_dig((u >> 20) & 0xf);
    out[3] = hex_dig((u >> 16) & 0xf);
    out[4] = hex_dig((u >> 12) & 0xf);
    out[5] = hex_dig((u >> 8) & 0xf);
    out[6] = hex_dig((u >> 4) & 0xf);
    out[7] = hex_dig((u >> 0) & 0xf);
    out[8] = 0;
}

void printf(const char *format, ...)
{
    va_list va;
    va_start(va, format);
    vprintf(format, va);
    va_end(va);
}

void vprintf(const char *format, va_list va)
{
    while (1)
    {
        char c = *format++;

        if (c == 0)
        {
            break;
        }

        if (c != '%')
        {
            putc(c);
            continue;
        }

        switch (c = *format++)
        {
        case 'd':
            {
                char buff[16];
                itoa(va_arg(va, int), buff);
```



```
                console_puts(buff, strlen(buff));
                break;
            }
        case 'x':
            {
                char buff[16];
                utox(va_arg(va, uint32_t), buff);
                console_puts(buff, strlen(buff));
                break;
            }
        case 's':
            {
                const char *str = va_arg(va, const char *);
                console_puts(str, strlen(str));
                break;
            }
        case 'c':
            {
                putc(va_arg(va, char));
                break;
            }
        case 0:
            {
                goto ret;
            }
        default:
            {
                putc(c);
                break;
            }
        }
    }
  ret:
    update_cursor();
}
```

```
#ifndef CONSOLE_H
#define CONSOLE_H

#include "stdarg.h"

void console_init();

void console_puts(const char *str, uint32_t len);

void printf(const char *format, ...);

void vprintf(const char *format, va_list va);

#endif
```

```
use32

global __syscall0
global __syscall1
global __syscall2
global __syscall3

extern _main
extern _exit

section .crt0

%macro perform_syscall 0
    push ecx
    push edx
    mov ecx, esp
    mov edx, .ret
    sysenter
.ret:
    pop edx
    pop ecx
%endmacro

start:
    call _main
    push eax
    call _exit

__syscall0:
    mov eax, [esp+0+4]
    perform_syscall
    ret

__syscall1:
    push ebx
    mov ebx, [esp+4+8]
    mov eax, [esp+4+4]
    perform_syscall
    pop ebx
    ret

__syscall2:
    push ebx
    push edi
    mov edi, [esp+8+12]
    mov ebx, [esp+8+8]
    mov eax, [esp+8+4]
    perform_syscall
    pop edi
    pop ebx
    ret

__syscall3:
    push ebx
    push edi
    push esi
    mov esi, [esp+12+16]
    mov edi, [esp+12+12]
    mov ebx, [esp+12+8]
    mov eax, [esp+12+4]
    perform_syscall
    pop esi
    pop edi
    pop ebx
    ret
```

```c
#include <radium.h>
#include "crt.h"

void regdump()
{
    _syscall0(SYS_REGDUMP);
}

void exit(int status)
{
    _syscall1(SYS_EXIT, status);
}

void yield()
{
    _syscall0(SYS_YIELD);
}

uint32_t fork()
{
    return _syscall0(SYS_FORK);
}

uint32_t wait(int *stat_loc)
{
    return _syscall1(SYS_WAIT, (uint32_t) stat_loc);
}

void console_log(const char *str)
{
    uint32_t len = 0;

    for (const char *p = str; *p; p++)
    {
        len++;
    }

    _syscall2(SYS_CONSOLE_LOG, (uint32_t) str, len);
}
```

```
#ifndef CRT_H
#define CRT_H

typedef unsigned int uint32_t;

#define NULL ((void*)0)

uint32_t _syscall0(uint32_t number);

uint32_t _syscall1(uint32_t number, uint32_t arg1);

uint32_t _syscall2(uint32_t number, uint32_t arg1, uint32_t arg2);

uint32_t _syscall3(uint32_t number, uint32_t arg1, uint32_t arg2, uint32_t arg3);

void regdump();

void exit(int status);

void yield();

uint32_t fork();

uint32_t wait(int *stat_loc);

void console_log(const char *str);

int main();

#endif
```

```
#include "gdt.h"
#include "panic.h"
#include "types.h"

typedef struct
{
    uint16_t limit_0_15;
    uint16_t base_0_15;
    uint8_t base_16_23;
    uint8_t access;
    uint8_t limit_16_19_and_flags;
    uint8_t base_24_31;
}
gdt_entry_t;

static gdt_entry_t gdt[6];

volatile struct
{
    uint16_t size;
    gdt_entry_t *offset;
} __attribute__ ((packed)) gdtr;

static void gdt_set_entry_raw(gdt_selector_t sel, uint32_t base, uint32_t limit,
                  uint8_t access)
{
    if (sel >= sizeof(gdt))
    {
        panic("GDT overflow");
    }

    uint8_t flags = 1 << 6;     // 32 bit segment

    // 1 MiB is the maximum segment size that can be expressed with 1 byte
    // granularity. If we need to express a size bigger than this, we need
    // to divide the size by 4 KiB and use 4 KiB granularity.
    if (limit >= (1 << 20))
    {
        limit /= 4096;
        flags |= 1 << 7;    // 4 KiB granularity
    }

    gdt_entry_t ent;
    ent.limit_0_15 = limit & 0xffff;
    ent.base_0_15 = base & 0xffff;
    ent.base_16_23 = (base >> 16) & 0xff;
    ent.access = access;
    ent.limit_16_19_and_flags = ((limit >> 16) & 0x0f) | flags;
    ent.base_24_31 = (base >> 24) & 0xff;

    gdt[sel / sizeof(gdt_entry_t)] = ent;
}

void gdt_set_entry(gdt_selector_t sel, uint32_t base, uint32_t limit,
               gdt_privilege_t priv, gdt_type_t type)
{
    gdt_set_entry_raw(sel, base, limit, (1 << 7) |  // present
                      ((priv & 3) << 5) |   // privilege
                      (1 << 4) |// dunno lol
                      ((type & 1) << 3) |   // code/data
                      (1 << 1)  // data segments are always writable, code
                                // segments are always readable
        );
}

void gdt_set_tss(gdt_selector_t sel, uint32_t base, uint32_t limit)
{
    gdt_set_entry_raw(sel, base, limit, 0x89);
}

void gdt_reload();

```



```
void gdt_init()
{
    gdtr.size = sizeof(gdt) - 1;
    gdtr.offset = gdt;

    gdt_set_entry(GDT_KERNEL_CODE, 0, 0xffffffff, GDT_KERNEL, GDT_CODE);
    gdt_set_entry(GDT_KERNEL_DATA, 0, 0xffffffff, GDT_KERNEL, GDT_DATA);
    gdt_set_entry(GDT_USER_CODE, 0, 0xffffffff, GDT_USER, GDT_CODE);
    gdt_set_entry(GDT_USER_DATA, 0, 0xffffffff, GDT_USER, GDT_DATA);

    gdt_reload();
}
```

```
 1  extern gdtr
 2  global gdt_reload
 3
 4  section .text
 5  gdt_reload:
 6      lgdt [gdtr]
 7      jmp 0x08:.flush_cs
 8  .flush_cs:
 9      mov eax, 0x10
10      mov ds, eax
11      mov es, eax
12      mov fs, eax
13      mov gs, eax
14      mov ss, eax
15      ret
```

```
 1  #ifndef GDT_H
 2  #define GDT_H
 3
 4  #include "types.h"
 5
 6  typedef enum
 7  {
 8      GDT_KERNEL_CODE = 0x08,
 9      GDT_KERNEL_DATA = 0x10,
10      GDT_USER_CODE = 0x18,
11      GDT_USER_DATA = 0x20,
12      GDT_TSS = 0x28,
13  }
14  gdt_selector_t;
15
16  typedef enum
17  {
18      GDT_KERNEL = 0,
19      GDT_USER = 3,
20  }
21  gdt_privilege_t;
22
23  typedef enum
24  {
25      GDT_DATA = 0,
26      GDT_CODE = 1,
27  }
28  gdt_type_t;
29
30  void gdt_set_entry(gdt_selector_t sel, uint32_t base, uint32_t limit,
31                 gdt_privilege_t priv, gdt_type_t type);
32
33  void gdt_set_tss(gdt_selector_t sel, uint32_t base, uint32_t limit);
34
35  void gdt_reload();
36
37  void gdt_init();
38
39  #endif
```

```
#include "console.h"
#include "gdt.h"
#include "idt.h"
#include "io.h"
#include "types.h"

typedef struct
{
    uint16_t offset_0_15;
    uint16_t segment;
    uint8_t zero;
    uint8_t type_attr;
    uint16_t offset_16_31;
}
idt_entry_t;

static idt_entry_t idt[256];

volatile struct
{
    uint16_t size;
    idt_entry_t *offset;
} __attribute__ ((packed)) idtr;

void interrupts_register_isr(uint8_t interrupt_no, uint32_t handler)
{
    idt_entry_t ent;

    ent.offset_0_15 = handler & 0xffff;
    ent.segment = GDT_KERNEL_CODE;
    ent.zero = 0;
    ent.type_attr = (1 << 7)// present
        | 0xe                   // interrupt gate
        ;
    ent.offset_16_31 = (handler >> 16) & 0xffff;

    idt[interrupt_no] = ent;
}

static void remap_irqs()
{
    // remap IRQ table
    outb(0x20, 0x11);
    outb(0xa0, 0x11);
    outb(0x21, 0x20);
    outb(0xa1, 0x28);
    outb(0x21, 0x04);
    outb(0xa1, 0x02);
    outb(0x21, 0x01);
    outb(0xa1, 0x01);
    outb(0x21, 0x00);
    outb(0xa1, 0x00);
}

void idt_init_asm();

void idt_load();

void idt_init()
{
    remap_irqs();
    idt_init_asm();

    idtr.size = sizeof(idt) - 1;
    idtr.offset = idt;
    idt_load();
}
```

```
1   #ifndef IDT_H
2   #define IDT_H
3
4   void idt_init();
5
6   #endif
```

```c
#include "crt.h"

int main()
{
    console_log("Hello world from init!\n");

    if (fork())
    {
        wait(NULL);
        return 123;
    }
    else
    {
        regdump();
        yield();
        regdump();
        return 456;
    }
}
```

```
#ifndef IO_H
#define IO_H

#include "types.h"

static void __attribute__ ((unused)) outb(uint16_t port, uint8_t value)
{
    __asm__ volatile ("outb %1, %0"::"r" (value), "Nd"(port));
}

static uint8_t __attribute__ ((unused)) inb(uint16_t port)
{
    volatile uint8_t value;
    __asm__ volatile ("inb %0, %1":"=r" (value):"Nd"(port));
    return value;
}

static void __attribute__ ((unused)) outl(uint16_t port, uint32_t value)
{
    __asm__ volatile ("outd %1, %0"::"r" (value), "Nd"(port));
}

static uint32_t __attribute__ ((unused)) inl(uint16_t port)
{
    volatile uint32_t value;
    __asm__ volatile ("ind %0, %1":"=r" (value):"Nd"(port));
    return value;
}

#endif
```

```
global idt_init_asm
global idt_load
extern interrupts_register_isr
extern idtr
extern panic
extern sched_switch

%macro register_isr 1
    push isr_%1
    push %1
    call interrupts_register_isr
    add esp, 8
%endmacro

%macro ack_irq 0
    push ax
    mov al, 0x20
    out 0xa0, al
    out 0x20, al
    pop ax
%endmacro

%macro begin_isr 1
    jmp isr_%1.end
    isr_%1:
%endmacro

%macro end_isr 1
    .end:
    register_isr %1
%endmacro

%macro generic_exception 2
    begin_isr %1
        push .msg
        call panic
        .msg db %2, 0
    end_isr %1
%endmacro

section .text
idt_load:
    lidt [idtr]
    ret

idt_init_asm:

generic_exception 0,  "divide by zero"
generic_exception 1,  "debug"
generic_exception 2,  "non-maskable interrupt"
generic_exception 3,  "breakpoint"
generic_exception 4,  "overflow"
generic_exception 5,  "bound range exceeded"
generic_exception 6,  "invalid opcode"
generic_exception 7,  "device not available"
generic_exception 8,  "double fault"
generic_exception 10, "invalid tss"
generic_exception 11, "segment not present"
generic_exception 12, "stack segment fault"
generic_exception 13, "general protection fault"

; page fault
begin_isr 14
    mov eax, cr2
    push eax
    push .msg
    call panic
    iret

    .msg db "page fault at 0x%x, error code: %x", 0
end_isr 14

```



```
; PIT irq
begin_isr 32
    ack_irq

    push ebp
    push dword 0
    push dword 0
    mov ebp, esp

    call sched_switch

    add esp, 8 ; fix up the fake stack frame we created
    pop ebp

    iret
end_isr 32

; keyboard irq
begin_isr 33
    ack_irq
    iret
end_isr 33

; spurious irq
begin_isr 39
    iret
end_isr 39

ret
```

```c
#include "kernel_page.h"
#include "paging.h"
#include "panic.h"
#include "string.h"

typedef union kernel_page
{
    union kernel_page *next;
    char mem[PAGE_SIZE];
}
kernel_page_t;

static kernel_page_t *allocated_to;

static kernel_page_t *end;

static kernel_page_t *next_free;

void kernel_page_init(virt_t begin_, virt_t end_)
{
    allocated_to = (void *)begin_;
    end = (void *)end_;
}

void *kernel_page_alloc()
{
    if (next_free)
    {
        kernel_page_t *page = next_free;
        next_free = next_free->next;
        return page;
    }

    if (allocated_to >= end)
    {
        return NULL;
    }

    page_map((virt_t) allocated_to, page_alloc(), PE_PRESENT | PE_READ_WRITE);
    void *retn = allocated_to++;

    return retn;
}

void *kernel_page_alloc_zeroed()
{
    void *page = kernel_page_alloc();
    memset32(page, 0, 1024);
    return page;
}

void kernel_page_free(void *ptr)
{
    kernel_page_t *page = ptr;

    page->next = next_free;
    next_free = page;
}
```

```
#ifndef KALLOC_H
#define KALLOC_H

#include "types.h"

void kernel_page_init(virt_t begin, virt_t end);

void *kernel_page_alloc();

void *kernel_page_alloc_zeroed();

void kernel_page_free(void *);

#endif
```

```
 1  global loader
 2  global end_of_image
 3
 4  extern kmain
 5
 6  section .multiboot_header
 7  align 4
 8  multiboot_header:
 9      dd 0x1badb002        ; magic
10      dd 3                 ; flags
11      dd -(0x1badb002 + 3) ; checksum = -(flags + magic)
12
13  section .text
14  align 4
15  loader:
16      mov esp, stack
17      push dword 0
18      push dword 0
19      mov ebp, esp
20
21      push eax ; multiboot magic number
22      push ebx ; pointer to multiboot struct
23
24      fninit
25      mov eax, cr0
26      or eax, 1 << 5  ; FPU NE bit
27      mov cr0, eax
28
29      call kmain
30
31  section .bss
32  align 4
33  resb 65536
34  stack:
35
36  section .end_of_image
37  end_of_image:
```

```c
#include "console.h"
#include "gdt.h"
#include "idt.h"
#include "kernel_page.h"
#include "multiboot.h"
#include "paging.h"
#include "panic.h"
#include "pit.h"
#include "sched.h"
#include "string.h"
#include "syscall.h"
#include "task.h"
#include "types.h"

static multiboot_info_t *mb;

static multiboot_module_t *find_module(const char *name)
{
    multiboot_module_t *mods = (void *)mb->mods_addr;

    for (size_t i = 0; i < mb->mods_count; i++)
    {
        if (streq((const char *)mods[i].cmdline, name))
        {
            return &mods[i];
        }
    }

    return NULL;
}

void kmain(multiboot_info_t * mb_, uint32_t magic)
{
    (void)magic;
    mb = mb_;

    console_init();

    printf("Radium booting from %s.\n", (const char *)mb->boot_loader_name);

    for (size_t i = 0; i < mb->mods_count; i++)
    {
        multiboot_module_t *mods = (void *)mb->mods_addr;
        paging_set_allocatable_start(mods[i].mod_end);
    }

    gdt_init();
    idt_init();
    pit_set_frequency(100);
    paging_init(mb);
    task_init();
    syscall_init();

    multiboot_module_t *mod = find_module("/init.bin");

    task_boot_init((const char *)mod->mod_start,
                   mod->mod_end - mod->mod_start);
}
```

```
/* multiboot.h - Multiboot header file. */
/* Copyright (C) 1999,2003,2007,2008,2009 Free Software Foundation, Inc. * *
   Permission is hereby granted, free of charge, to any person obtaining a
   copy * of this software and associated documentation files (the
   "Software"), to * deal in the Software without restriction, including
   without limitation the * rights to use, copy, modify, merge, publish,
   distribute, sublicense, and/or * sell copies of the Software, and to permit
   persons to whom the Software is * furnished to do so, subject to the
   following conditions: * * The above copyright notice and this permission
   notice shall be included in * all copies or substantial portions of the
   Software. * * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
   KIND, EXPRESS OR * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
   MERCHANTABILITY, * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
   IN NO EVENT SHALL ANY * DEVELOPER OR DISTRIBUTOR BE LIABLE FOR ANY CLAIM,
   DAMAGES OR OTHER LIABILITY, * WHETHER IN AN ACTION OF CONTRACT, TORT OR
   OTHERWISE, ARISING FROM, OUT OF OR * IN CONNECTION WITH THE SOFTWARE OR THE
   USE OR OTHER DEALINGS IN THE SOFTWARE. */

#ifndef MULTIBOOT_HEADER
#define MULTIBOOT_HEADER 1

/* How many bytes from the start of the file we search for the header. */
#define MULTIBOOT_SEARCH                        8192

/* The magic field should contain this. */
#define MULTIBOOT_HEADER_MAGIC                  0x1BADB002

/* This should be in %eax. */
#define MULTIBOOT_BOOTLOADER_MAGIC              0x2BADB002

/* The bits in the required part of flags field we don't support. */
#define MULTIBOOT_UNSUPPORTED                   0x0000fffc

/* Alignment of multiboot modules. */
#define MULTIBOOT_MOD_ALIGN                     0x00001000

/* Alignment of the multiboot info structure. */
#define MULTIBOOT_INFO_ALIGN                    0x00000004

/* Flags set in the 'flags' member of the multiboot header. */

/* Align all boot modules on i386 page (4KB) boundaries. */
#define MULTIBOOT_PAGE_ALIGN                    0x00000001

/* Must pass memory information to OS. */
#define MULTIBOOT_MEMORY_INFO                   0x00000002

/* Must pass video information to OS. */
#define MULTIBOOT_VIDEO_MODE                    0x00000004

/* This flag indicates the use of the address fields in the header. */
#define MULTIBOOT_AOUT_KLUDGE                   0x00010000

/* Flags to be set in the 'flags' member of the multiboot info structure. */

/* is there basic lower/upper memory information? */
#define MULTIBOOT_INFO_MEMORY                   0x00000001
/* is there a boot device set? */
#define MULTIBOOT_INFO_BOOTDEV                  0x00000002
/* is the command-line defined? */
#define MULTIBOOT_INFO_CMDLINE                  0x00000004
/* are there modules to do something with? */
#define MULTIBOOT_INFO_MODS                     0x00000008

/* These next two are mutually exclusive */

/* is there a symbol table loaded? */
#define MULTIBOOT_INFO_AOUT_SYMS                0x00000010
/* is there an ELF section header table? */
#define MULTIBOOT_INFO_ELF_SHDR                 0X00000020

/* is there a full memory map? */
```



```
#define MULTIBOOT_INFO_MEM_MAP                  0x00000040

/* Is there drive info? */
#define MULTIBOOT_INFO_DRIVE_INFO               0x00000080

/* Is there a config table? */
#define MULTIBOOT_INFO_CONFIG_TABLE             0x00000100

/* Is there a boot loader name? */
#define MULTIBOOT_INFO_BOOT_LOADER_NAME         0x00000200

/* Is there a APM table? */
#define MULTIBOOT_INFO_APM_TABLE                0x00000400

/* Is there video information? */
#define MULTIBOOT_INFO_VIDEO_INFO               0x00000800

#ifndef ASM_FILE

typedef unsigned short multiboot_uint16_t;
typedef unsigned int multiboot_uint32_t;
typedef unsigned long long multiboot_uint64_t;

struct multiboot_header
{
    /* Must be MULTIBOOT_MAGIC - see above. */
    multiboot_uint32_t magic;

    /* Feature flags. */
    multiboot_uint32_t flags;

    /* The above fields plus this one must equal 0 mod 2^32. */
    multiboot_uint32_t checksum;

    /* These are only valid if MULTIBOOT_AOUT_KLUDGE is set. */
    multiboot_uint32_t header_addr;
    multiboot_uint32_t load_addr;
    multiboot_uint32_t load_end_addr;
    multiboot_uint32_t bss_end_addr;
    multiboot_uint32_t entry_addr;

    /* These are only valid if MULTIBOOT_VIDEO_MODE is set. */
    multiboot_uint32_t mode_type;
    multiboot_uint32_t width;
    multiboot_uint32_t height;
    multiboot_uint32_t depth;
};

/* The symbol table for a.out. */
struct multiboot_aout_symbol_table
{
    multiboot_uint32_t tabsize;
    multiboot_uint32_t strsize;
    multiboot_uint32_t addr;
    multiboot_uint32_t reserved;
};
typedef struct multiboot_aout_symbol_table multiboot_aout_symbol_table_t;

/* The section header table for ELF. */
struct multiboot_elf_section_header_table
{
    multiboot_uint32_t num;
    multiboot_uint32_t size;
    multiboot_uint32_t addr;
    multiboot_uint32_t shndx;
};
typedef struct multiboot_elf_section_header_table
    multiboot_elf_section_header_table_t;

struct multiboot_info
{
    /* Multiboot info version number */
```

```
145     multiboot_uint32_t flags;
146
147     /* Available memory from BIOS */
148     multiboot_uint32_t mem_lower;
149     multiboot_uint32_t mem_upper;
150
151     /* "root" partition */
152     multiboot_uint32_t boot_device;
153
154     /* Kernel command line */
155     multiboot_uint32_t cmdline;
156
157     /* Boot-Module list */
158     multiboot_uint32_t mods_count;
159     multiboot_uint32_t mods_addr;
160
161     union
162     {
163         multiboot_aout_symbol_table_t aout_sym;
164         multiboot_elf_section_header_table_t elf_sec;
165     } u;
166
167     /* Memory Mapping buffer */
168     multiboot_uint32_t mmap_length;
169     multiboot_uint32_t mmap_addr;
170
171     /* Drive Info buffer */
172     multiboot_uint32_t drives_length;
173     multiboot_uint32_t drives_addr;
174
175     /* ROM configuration table */
176     multiboot_uint32_t config_table;
177
178     /* Boot Loader Name */
179     multiboot_uint32_t boot_loader_name;
180
181     /* APM table */
182     multiboot_uint32_t apm_table;
183
184     /* Video */
185     multiboot_uint32_t vbe_control_info;
186     multiboot_uint32_t vbe_mode_info;
187     multiboot_uint16_t vbe_mode;
188     multiboot_uint16_t vbe_interface_seg;
189     multiboot_uint16_t vbe_interface_off;
190     multiboot_uint16_t vbe_interface_len;
191 };
192 typedef struct multiboot_info multiboot_info_t;
193
194 struct multiboot_mmap_entry
195 {
196     multiboot_uint32_t size;
197     multiboot_uint64_t addr;
198     multiboot_uint64_t len;
199 #define MULTIBOOT_MEMORY_AVAILABLE              1
200 #define MULTIBOOT_MEMORY_RESERVED               2
201     multiboot_uint32_t type;
202 } __attribute__ ((packed));
203 typedef struct multiboot_mmap_entry multiboot_memory_map_t;
204
205 struct multiboot_mod_list
206 {
207     /* the memory used goes from bytes 'mod_start' to 'mod_end-1' inclusive */
208     multiboot_uint32_t mod_start;
209     multiboot_uint32_t mod_end;
210
211     /* Module command line */
212     multiboot_uint32_t cmdline;
213
214     /* padding to take it to 16 bytes (must be zero) */
215     multiboot_uint32_t pad;
216 };
```



```
217 typedef struct multiboot_mod_list multiboot_module_t;
218
219 #endif /* ! ASM_FILE */
220
221 #endif /* ! MULTIBOOT_HEADER */
```

```
#include "console.h"
#include "paging.h"
#include "panic.h"
#include "string.h"

static phys_t next_free_page;

#define FL_PAGING_ENABLED (1 << 31)

static bool paging_enabled()
{
    uint32_t cr0;
  __asm__("mov %0, cr0":"=r"(cr0));
    return ! !(cr0 & FL_PAGING_ENABLED);
}

void set_page_directory(phys_t page_directory)
{
    uint32_t cr0;

    __asm__("mov cr3, %0"::"r"(page_directory));

  __asm__("mov %0, cr0":"=r"(cr0));
    cr0 |= FL_PAGING_ENABLED;
  __asm__("mov cr0, %0"::"r"(cr0):"memory");
}

static void invlpg(virt_t virt)
{
    __asm__ volatile ("invlpg [%0]"::"r" (virt):"memory");
}

phys_t page_alloc()
{
    phys_t page = next_free_page;

    if (paging_enabled())
    {
        phys_t *temp_mapping = page_temp_map(page);
        next_free_page = *temp_mapping;
        page_temp_unmap();
    }
    else
    {
        next_free_page = *(phys_t *) page;
    }

    return page;
}

void page_free(phys_t addr)
{
    if (paging_enabled())
    {
        phys_t *temp_mapping = page_temp_map(addr);
        *temp_mapping = next_free_page;
        page_temp_unmap();
    }
    else
    {
        *(phys_t *) addr = next_free_page;
    }

    next_free_page = addr;
}

void page_map(virt_t virt_page, phys_t phys_page, page_flags_t flags)
{
    // page directories are always recursively mapped into themselves:
    uint32_t *current_page_directory = (uint32_t *) CURRENT_PAGE_DIRECTORY;

    size_t page_dir_i = (virt_page / 4096) / 1024;
```



```
    size_t page_tab_i = (virt_page / 4096) % 1024;

    uint32_t *page_table =
        (uint32_t *) (CURRENT_PAGE_TABLE_BASE + page_dir_i * PAGE_SIZE);

    uint32_t pd_entry = current_page_directory[page_dir_i];
    if (!(pd_entry & PE_PRESENT))
    {
        current_page_directory[page_dir_i] =
            page_alloc() | (flags & PE_FLAG_MASK);
        invlpg((virt_t) page_table);
        memset(page_table, 0, 4096);
    }

    page_table[page_tab_i] =
        (phys_page & PE_ADDR_MASK) | (flags & PE_FLAG_MASK);
    invlpg((virt_t) virt_page);
}

void page_unmap(virt_t virt_page)
{
    page_map(virt_page, 0, 0);
}

phys_t virt_to_phys(virt_t virt)
{
    uint32_t *current_page_directory = (uint32_t *) CURRENT_PAGE_DIRECTORY;

    size_t page_dir_i = (virt / 4096) / 1024;
    size_t page_tab_i = (virt / 4096) % 1024;
    size_t page_offset = virt % 4096;

    if (!(current_page_directory[page_dir_i] & PE_PRESENT))
    {
        return 0;
    }

    uint32_t *page_table =
        (uint32_t *) (CURRENT_PAGE_TABLE_BASE + page_dir_i * PAGE_SIZE);

    uint32_t page_tab_entry = page_table[page_tab_i];

    if (!(page_tab_entry & PE_PRESENT))
    {
        return 0;
    }

    return (page_tab_entry & PE_ADDR_MASK) + page_offset;
}

static uint32_t *const temp_page_entry = (void *)CURRENT_PAGE_TABLE_BASE;

static uint32_t old_null_page;

void *page_temp_map(phys_t phys_page)
{
    old_null_page = *temp_page_entry;
    *temp_page_entry = phys_page | PE_PRESENT | PE_READ_WRITE;
    invlpg(0);
    return NULL;            // we map in the temp page at NULL - todo
                                // fix...
}

void page_temp_unmap()
{
    *temp_page_entry = old_null_page;
    invlpg(0);
}

bool page_is_user_mapped(virt_t virt)
{
    uint32_t dir_i = virt / (1024 * PAGE_SIZE);
```

```
145       uint32_t base_i = (virt / PAGE_SIZE);
146
147       uint32_t *page_directory = (uint32_t *) CURRENT_PAGE_DIRECTORY;
148
149       if (!(page_directory[dir_i] & (PE_PRESENT | PE_USER)))
150       {
151           return false;
152       }
153
154       uint32_t *page_table_base = (uint32_t *) CURRENT_PAGE_TABLE_BASE;
155
156       if (!(page_table_base[base_i] & (PE_PRESENT | PE_USER)))
157       {
158           return false;
159       }
160
161       return true;
162   }
```

```
#ifndef PAGING_H
#define PAGING_H

#include "types.h"
#include "multiboot.h"

#define PAGE_SIZE 4096

#define PE_FLAG_MASK  (PAGE_SIZE - 1)
#define PE_ADDR_MASK  (~PE_FLAG_MASK)

#define KERNEL_STACK_BEGIN       0x0fc00000ul
#define KERNEL_STACK_END         0x0ffffffcul
#define USER_BEGIN               0x10000000ul
#define USER_STACK_END           0xffc00000ul
#define CURRENT_PAGE_DIRECTORY   0xfffff000ul
#define CURRENT_PAGE_TABLE_BASE  0xffc00000ul

typedef enum
{
    PE_PRESENT = 1 << 0,
    PE_READ_WRITE = 1 << 1,
    PE_USER = 1 << 2,
}
page_flags_t;

void paging_set_allocatable_start(phys_t addr);

void paging_init(struct multiboot_info *mb);

void set_page_directory(phys_t page_directory);

phys_t page_alloc();

void page_free(phys_t addr);

void page_map(virt_t virt_page, phys_t phys_page, page_flags_t flags);

void page_unmap(virt_t virt_page);

phys_t virt_to_phys(virt_t virt);

void *page_temp_map(phys_t phys_page);

void page_temp_unmap();

bool page_is_user_mapped(virt_t virt);

#endif
```

```
#include "console.h"
#include "kernel_page.h"
#include "multiboot.h"
#include "paging.h"
#include "string.h"
#include "util.h"

static phys_t kernel_end;

static phys_t alloc_zeroed_page()
{
    phys_t page = page_alloc();
    memset32((void *)page, 0, 1024);
    return page;
}

static size_t register_available_memory_region(multiboot_memory_map_t * region)
{
    size_t pages_registered = 0;

    for (size_t offset = 0; offset + PAGE_SIZE <= region->len;
         offset += PAGE_SIZE)
    {
        phys_t addr = region->addr + offset;

        if (addr < kernel_end)
        {
            // don't put memory before the end of the kernel in the free list
            continue;
        }

        page_free(addr);
        pages_registered++;
    }

    return pages_registered;
}

static size_t register_available_memory(multiboot_info_t * mb)
{
    multiboot_memory_map_t *mmap = (void *)mb->mmap_addr;
    size_t pages_registered = 0;

    for (size_t i = 0; i < mb->mmap_length / sizeof(multiboot_memory_map_t);
         i++)
    {
        if (mmap[i].type == MULTIBOOT_MEMORY_AVAILABLE)
        {
            pages_registered += register_available_memory_region(mmap + i);
        }
    }

    return pages_registered;
}

static void create_page_tables_for_kernel_space(uint32_t * page_directory)
{
    for (size_t i = 0; i < KERNEL_STACK_END / (4 * 1024 * 1024); i++)
    {
        page_directory[i] = alloc_zeroed_page() | PE_PRESENT | PE_READ_WRITE;
    }
}

static void identity_map_kernel(uint32_t * page_directory)
{
    // we start looping from PAGE_SIZE in order to leave the null page
    // unmapped
    // accessing it will cause a page fault.
    for (phys_t addr = PAGE_SIZE; addr <= kernel_end; addr += PAGE_SIZE)
    {
        size_t page_dir_i = addr / 4096 / 1024;
        size_t page_tab_i = addr / 4096 % 1024;
```



```
        uint32_t page_dir_ent = page_directory[page_dir_i];
        if (!page_dir_ent)
        {
            page_dir_ent = page_directory[page_dir_i] =
                alloc_zeroed_page() | PE_PRESENT | PE_READ_WRITE;
        }
        uint32_t *page_tab = (uint32_t *) (page_dir_ent & PE_ADDR_MASK);
        page_tab[page_tab_i] = addr | PE_PRESENT | PE_READ_WRITE;
    }
}

static void recursively_map_page_directory(uint32_t * page_directory)
{
    page_directory[1023] =
        (phys_t) page_directory | PE_PRESENT | PE_READ_WRITE;
}

void paging_set_allocatable_start(phys_t addr)
{
    if (addr > kernel_end)
    {
        kernel_end = round_up(addr, PAGE_SIZE);
    }
}

void paging_init(multiboot_info_t * mb)
{
    extern int end_of_image;
    paging_set_allocatable_start((phys_t) & end_of_image);

    size_t pages_registered = register_available_memory(mb);
    printf("%d MiB available useful memory.\n",
           pages_registered * PAGE_SIZE / 1024 / 1024);

    uint32_t *page_directory = (uint32_t *) alloc_zeroed_page();

    create_page_tables_for_kernel_space(page_directory);
    identity_map_kernel(page_directory);
    recursively_map_page_directory(page_directory);

    set_page_directory((phys_t) page_directory);

    kernel_page_init(kernel_end, KERNEL_STACK_BEGIN);
}
```

```c
#include "console.h"
#include "panic.h"
#include "stdarg.h"
#include "types.h"

#define RECORD_SIZE 32

typedef struct
{
    uint32_t addr;
    char name[RECORD_SIZE - sizeof(uint32_t)];
}
symbol_record_t;

#define SYMBOL_ENTRIES (65536 / RECORD_SIZE)

extern symbol_record_t panic_symbols[SYMBOL_ENTRIES];

void panic_print_backtrace() __attribute__ ((noreturn));

void panic(const char *format, ...)
{
    __asm__ volatile ("cli");

    va_list va;
    va_start(va, format);
    printf("\npanic: ");
    vprintf(format, va);
    printf("\n");
    va_end(va);

    panic_print_backtrace();
}

void panic_print_backtrace_item(uint32_t addr)
{
    uint32_t base = 0;
    const char *name = "?";

    for (size_t i = 0; i < SYMBOL_ENTRIES; i++)
    {
        if (panic_symbols[i].addr == 0)
        {
            break;
        }
        if (panic_symbols[i].addr < addr && panic_symbols[i].addr > base)
        {
            base = panic_symbols[i].addr;
            name = panic_symbols[i].name;
        }
    }

    printf("  [<0x%x>] %s +%d\n", addr, name, addr - base);
}
```

```
global panic_print_backtrace
global panic_symbols

extern printf
extern vprintf
extern panic_print_backtrace_item

section .rodata

panic_symbols:
    db '@@@ PANIC SYMBOL TABLE @@@'
    times 65536 - ($ - panic_symbols) db 0

section .text
panic_print_backtrace:
    mov esp, ebp
    pop ebp
    cmp dword [esp], 0
    je .end_loop
    call panic_print_backtrace_item
    jmp panic_print_backtrace
.end_loop:
    hlt
```

```
1  #ifndef PANIC_H
2  #define PANIC_H
3
4  void panic(const char *message, ...) __attribute__ ((noreturn));
5
6  #endif
```

```c
#include "io.h"
#include "panic.h"
#include "pit.h"

#define PIT_CLOCK_FREQUENCY 1193180

void pit_set_frequency(uint32_t hz)
{
    uint32_t divisor = PIT_CLOCK_FREQUENCY / hz;

    if (divisor > 0xffff)
    {
        panic("frequency too low");
    }

    outb(0x43, 0x36);
    outb(0x40, divisor & 0xff);
    outb(0x40, (divisor >> 8) & 0xff);
}
```

```
1  #ifndef PIT_H
2  #define PIT_H
3
4  #include "types.h"
5
6  void pit_set_frequency(uint32_t hz);
7
8  #endif
```

```
#ifndef RADIUM_H
#define RADIUM_H

#define ENOSYS 1 // no such syscall
#define EFAULT 2 // bad address

// debug syscalls:
#define SYS_REGDUMP     0
#define SYS_CONSOLE_LOG 1

// process management syscalls:
#define SYS_EXIT        2
#define SYS_YIELD       3
#define SYS_FORK        4
#define SYS_WAIT        5

#endif
```

```
# Radium

Hobby OS.

## Memory Map

| Begin       | End (incl.) | Purpose
| ----------- | ----------- | -------
| `0000_0000` | `0000_0fff` | Null page. Not mapped in.
| `0000_1000` | `000f_ffff` | Low memory. Untouched.
| `0010_0000` | *???*       | The kernel is loaded here by GRUB.
| *???*       | `0fbf_ffff` | Kernel dynamic allocation area.
| `0fc0_0000` | `0fc0_0fff` | Kernel stack guard page. Not mapped in.
| `0fc0_1000` | `0fff_ffff` | Kernel stack.
| `1000_0000` | `ffbf_ffff` | User address space.
| `ffc0_0000` | `ffff_efff` | Recursively mapped page tables
| `ffff_f000` | `ffff_ffff` | Recursively mapped page directory
```

```
1  #ifndef SCHED_H
2  #define SCHED_H
3
4  void sched_begin() __attribute__ ((noreturn));
5
6  void sched_switch();
7
8  #endif
```

```
use32

global sched_begin
global sched_switch
global task_fork

extern current_task
extern sched_next
extern task_fork_inner

%define USER_CODE (0x18 | 3)
%define USER_DATA (0x20 | 3)

%define task_fpu_state(task)     [(task) + 0]
%define task_esp(task)           [(task) + 512]
%define task_eip(task)           [(task) + 516]
%define task_kernel_stack(task)  [(task) + 520]
%define task_page_dir_phys(task) [(task) + 528]

sched_begin:
    mov ax, USER_DATA
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ecx, 0xffc00000 ; userland stack end
    mov edx, 0x10000000 ; task entry point
    sti
    sysexit

sched_switch:
    ; save old task state
    pusha
    mov eax, [current_task]
    fxsave task_fpu_state(eax)
    mov task_esp(eax), esp
    mov task_eip(eax), dword .return

    call sched_next
    mov [current_task], eax

    mov ebx, task_page_dir_phys(eax)
    mov cr3, ebx

    fxrstor task_fpu_state(eax)
    mov esp, task_esp(eax)
    jmp task_eip(eax)
.return:
    popa
    ret

task_fork:
    xor eax, eax

    pusha
    call task_fork_inner

    ; we need to save the return value of task_fork_inner so that the parent
    ; task that calls this function gets the right task_t* for the newly
    ; created child task:
    mov [esp + 7*4], eax ; EAX from pusha

    mov task_esp(eax), esp
    mov task_eip(eax), dword .return
.return:
    popa
    ret
```

```
#include <radium.h>

#include "console.h"
#include "paging.h"
#include "panic.h"
#include "sched.h"
#include "syscall.h"
#include "task.h"
#include "util.h"

#define REG_VECTOR(regs) ((regs)->eax)
#define REG_RETURN(regs) ((regs)->eax)

#define REG_ARG1(regs) ((regs)->ebx)
#define REG_ARG2(regs) ((regs)->edi)
#define REG_ARG3(regs) ((regs)->esi)

static bool valid_user_buffer(virt_t ptr, size_t len)
{
    if ((0xffffffffu - len) < ptr)
    {
        return false;
    }

    size_t page_offset = ptr % PAGE_SIZE;

    ptr -= page_offset;
    len += page_offset;

    virt_t end = ptr + len;

    while (ptr < end)
    {
        if (!page_is_user_mapped(ptr))
        {
            return false;
        }

        ptr += PAGE_SIZE;
    }

    return true;
}

static uint32_t syscall_regdump(registers_t * regs)
{
    printf("process %d register dump:\n", current_task->pid);
    printf("    eax=%x, ebx=%x, ecx=%x, edx=%x\n", regs->eax, regs->ebx,
           regs->ecx, regs->edx);
    printf("    esp=%x, ebp=%x, esi=%x, edi=%x\n", regs->esp, regs->ebp,
           regs->esi, regs->edi);

    return 0;
}

static uint32_t syscall_exit(registers_t * regs)
{
    uint8_t status = REG_ARG1(regs) & 0xff;

    if (current_task->pid == 1)
    {
        panic("init exited with status: %d", status);
    }

    task_kill(current_task, status);

    // goodbye!
    sched_switch();

    return 0;
}

```



```
static uint32_t syscall_yield(registers_t * regs)
{
    (void)regs;
    sched_switch();
    return 0;
}

static uint32_t syscall_fork(registers_t * regs)
{
    (void)regs;

    task_t *new_task = task_fork();

    if (new_task)
    {
        // parent
        return new_task->pid;
    }
    else
    {
        // child
        return 0;
    }
}

static uint32_t syscall_wait(registers_t * regs)
{
    if (REG_ARG1(regs) != 0 && !valid_user_buffer(REG_ARG1(regs), sizeof(int)))
    {
        return -EFAULT;
    }

    int *stat_loc = (int *)REG_ARG1(regs);

  again:
    if (current_task->wait_queue.live.head)
    {
        task_t *child = current_task->wait_queue.live.head;
        current_task->wait_queue.live.head = child->wait_queue.dead.next;
        if (!current_task->wait_queue.live.head)
        {
            current_task->wait_queue.live.tail = NULL;
        }

        uint32_t child_pid = child->pid;
        uint8_t child_status = child->exit_status;

        task_destroy(child);

        if (stat_loc)
        {
            *stat_loc = child_status;
        }

        return child_pid;
    }

    current_task->state = TASK_BLOCK_WAIT;
    sched_switch();
    goto again;
}

static uint32_t syscall_console_log(registers_t * regs)
{
    if (valid_user_buffer(REG_ARG1(regs), REG_ARG2(regs)))
    {
        console_puts((const char *)REG_ARG1(regs), REG_ARG2(regs));
        return 0;
    }
    else
    {
        return -EFAULT;
```

```
    }
}

typedef uint32_t(syscall_t) (registers_t *);

static syscall_t *syscall_table[] = {
    [SYS_REGDUMP] = syscall_regdump,
    [SYS_EXIT] = syscall_exit,
    [SYS_YIELD] = syscall_yield,
    [SYS_FORK] = syscall_fork,
    [SYS_WAIT] = syscall_wait,
    [SYS_CONSOLE_LOG] = syscall_console_log,
};

void syscall_dispatch(registers_t * regs)
{
    if (REG_VECTOR(regs) > countof(syscall_table))
    {
        REG_RETURN(regs) = -ENOSYS;
        return;
    }

    syscall_t *func = syscall_table[regs->eax];

    if (!func)
    {
        REG_RETURN(regs) = -ENOSYS;
        return;
    }

    current_task->syscall_registers = regs;

    REG_RETURN(regs) = func(regs);

    current_task->syscall_registers = NULL;
}
```

```
use32

extern syscall_dispatch

global syscall_entry
global syscall_init

%define IA32_SYSENTER_CS  0x174
%define IA32_SYSENTER_ESP 0x175
%define IA32_SYSENTER_EIP 0x176

%define KERNEL_CODE 0x08
%define KERNEL_DATA 0x10

syscall_init:
    xor edx, edx

    mov ecx, IA32_SYSENTER_CS
    mov eax, KERNEL_CODE
    wrmsr

    mov ecx, IA32_SYSENTER_ESP
    mov eax, 0x0ffffffc
    wrmsr

    mov ecx, IA32_SYSENTER_EIP
    mov eax, syscall_entry
    wrmsr

    ret

syscall_entry:
    pusha

    push dword 0
    push dword 0
    mov ebp, esp

    lea eax, [esp + 8]
    push eax
    call syscall_dispatch
    add esp, 12 ; 4 bytes for the argument we passed to syscall_dispatch, and
                ; another 8 to compensate for the fake stack frame we pushed.

    popa

    ; STI apparently waits one instruction before enabling interrupts, so
    ; despite how it appears, this return sequence should be race-free.
    sti
    sysexit
```

```
 1  #ifndef SYSCALL_H
 2  #define SYSCALL_H
 3
 4  #include "types.h"
 5
 6  typedef struct
 7  {
 8      uint32_t edi;
 9      uint32_t esi;
10      uint32_t ebp;
11      uint32_t esp;
12      uint32_t ebx;
13      uint32_t edx;
14      uint32_t ecx;
15      uint32_t eax;
16  }
17  registers_t;
18
19  void syscall_init();
20
21  #endif
```

```
#include "console.h"
#include "gdt.h"
#include "kernel_page.h"
#include "paging.h"
#include "panic.h"
#include "sched.h"
#include "string.h"
#include "task.h"
#include "util.h"

static_assert(tss_t_is_0x68_bytes_long, sizeof(tss_t) == 0x68);

static tss_t tss;

task_t *current_task;

static task_t *tasks[1024];

task_t *task_for_pid(uint32_t pid)
{
    if (pid > countof(tasks))
    {
        return NULL;
    }

    return tasks[pid];
}

static task_t *alloc_empty_task()
{
    for (uint32_t pid = 1; pid < countof(tasks); pid++)
    {
        if (tasks[pid])
        {
            continue;
        }

        tasks[pid] = kernel_page_alloc_zeroed();
        tasks[pid]->state = TASK_READY;
        tasks[pid]->pid = pid;
        tasks[pid]->wait_queue.live.head = NULL;
        tasks[pid]->wait_queue.live.tail = NULL;
        return tasks[pid];
    }

    return NULL;
}

static void create_skeleton_page_directory(task_t * task)
{
    // initialise task page directory
    uint32_t *current_page_directory = (uint32_t *) CURRENT_PAGE_DIRECTORY;
    uint32_t *task_page_directory = kernel_page_alloc_zeroed();

    if (!task_page_directory)
    {
        panic("could not allocate page for new task's page directory");
    }

    for (size_t i = 0; i < KERNEL_STACK_BEGIN / (4 * 1024 * 1024); i++)
    {
        task_page_directory[i] = current_page_directory[i];
    }

    task->page_directory = task_page_directory;
    task->page_directory_phys = virt_to_phys((virt_t) task_page_directory);
    task->page_directory[1023] =
        task->page_directory_phys | PE_PRESENT | PE_READ_WRITE;

    uint32_t *kernel_stack_page_table = kernel_page_alloc_zeroed();
    task->page_directory[KERNEL_STACK_BEGIN / (4 * 1024 * 1024)] =
        virt_to_phys((virt_t) kernel_stack_page_table) | PE_PRESENT |
```



```
        PE_READ_WRITE;

    task->kernel_stack = kernel_page_alloc_zeroed();
    kernel_stack_page_table[1023] =
        virt_to_phys((virt_t) task->kernel_stack) | PE_PRESENT | PE_READ_WRITE;
}

void task_init()
{
    gdt_set_tss(GDT_TSS, (uint32_t) & tss, sizeof(tss));
    gdt_reload();

    tss.ss0 = GDT_KERNEL_DATA;
    tss.esp0 = KERNEL_STACK_END;

    // pointer to the IO permission bitmap is beyond the end of the segment
    tss.iopb = sizeof(tss);

    __asm__ volatile ("ltr ax"::"a" ((uint16_t) GDT_TSS | 3));

    task_t *init_task = alloc_empty_task();
    create_skeleton_page_directory(init_task);

    // user stack
    uint32_t *user_stack_page_table = kernel_page_alloc_zeroed();
    init_task->page_directory[1022] =
        virt_to_phys((virt_t) user_stack_page_table) | PE_PRESENT |
        PE_READ_WRITE | PE_USER;
    user_stack_page_table[1023] =
        page_alloc() | PE_PRESENT | PE_READ_WRITE | PE_USER;

    init_task->ppid = 0;

    current_task = init_task;
}

void task_boot_init(const char *text, size_t size)
{
    task_t *init_task = task_for_pid(1);

    set_page_directory(init_task->page_directory_phys);

    for (size_t i = 0; i < size; i += PAGE_SIZE)
    {
        phys_t page = page_alloc();
        page_map(USER_BEGIN + i, page, PE_PRESENT | PE_USER);
        size_t copy_size = size - i;
        if (copy_size > PAGE_SIZE)
        {
            copy_size = PAGE_SIZE;
        }
        memcpy((void *)(USER_BEGIN + i), (void *)(text + i), copy_size);
    }

    sched_begin();
}

static void copy_userland_pages(task_t * new_task)
{
    uint32_t *current_page_directory = (uint32_t *) CURRENT_PAGE_DIRECTORY;

    for (size_t dir_i = USER_BEGIN / (PAGE_SIZE * 1024); dir_i < 1023; dir_i++)
    {
        if (!current_page_directory[dir_i])
        {
            continue;
        }

        uint32_t *current_page_table =
            (uint32_t *) (CURRENT_PAGE_TABLE_BASE + dir_i * PAGE_SIZE);

        uint32_t *new_page_table = kernel_page_alloc();
```

```
        new_task->page_directory[dir_i] =
            virt_to_phys((virt_t) new_page_table) |
            (current_page_directory[dir_i] & PE_FLAG_MASK);

        for (size_t tab_i = 0; tab_i < 1024; tab_i++)
        {
            uint32_t current_entry = current_page_table[tab_i];
            void *current_virt =
                (void *)((dir_i * PAGE_SIZE * 1024) + (tab_i * PAGE_SIZE));

            if (!(current_entry & PE_PRESENT))
            {
                continue;
            }

            phys_t new_page = page_alloc();

            void *new_page_mapping = page_temp_map(new_page);
            memcpy(new_page_mapping, current_virt, PAGE_SIZE);
            page_temp_unmap();

            new_page_table[tab_i] = new_page | (current_entry & PE_FLAG_MASK);
        }
    }
}

task_t *task_fork_inner()
{
    task_t *new_task = alloc_empty_task();
    create_skeleton_page_directory(new_task);

    memcpy(&new_task->fpu_state, &current_task->fpu_state,
           sizeof(new_task->fpu_state));
    memcpy(new_task->kernel_stack, current_task->kernel_stack, PAGE_SIZE);

    copy_userland_pages(new_task);

    new_task->ppid = current_task->pid;

    new_task->syscall_registers = current_task->syscall_registers;

    return new_task;
}

void task_kill(task_t * task, uint8_t status)
{
    task->exit_status = status;

    task_t *parent = task_for_pid(task->ppid);

    // if this task has any children in the wait queue, just clean them up as
    // nothing is interested in them anymore.
    task_t *waitable_child = task->wait_queue.live.head;
    while (waitable_child)
    {
        task_t *next_waitable_child = waitable_child->wait_queue.dead.next;
        task_destroy(waitable_child);
        waitable_child = next_waitable_child;
    }

    // reparent children - TODO don't scan all processes...
    for (uint32_t i = 2; i < countof(tasks); i++)
    {
        task_t *child = task_for_pid(i);
        if (child && child->ppid == task->pid)
        {
            child->ppid = 1;
        }
    }

    // insert this task into the parent's wait queue
    if (parent->wait_queue.live.tail)
```



```
    {
        parent->wait_queue.live.tail->wait_queue.dead.next = task;
        parent->wait_queue.live.tail = task;
    }
    else
    {
        parent->wait_queue.live.head = task;
        parent->wait_queue.live.tail = task;
    }

    task->wait_queue.dead.next = NULL;

    // wake parent up if it's blocked in wait()
    if (parent->state == TASK_BLOCK_WAIT)
    {
        parent->state = TASK_READY;
    }

    // set state to EXITING so the scheduler never reschedules this task
    current_task->state = TASK_EXITING;
}

void task_destroy(task_t * task)
{
    tasks[task->pid] = NULL;

    kernel_page_free(task->kernel_stack);

    // TODO - free pages allocated for the task that are referenced from its
    // page directory

    kernel_page_free(task->page_directory);
    kernel_page_free(task);
}

task_t *sched_next()
{
    uint32_t current_pid = current_task->pid;

    for (size_t i = current_pid + 1; i < countof(tasks); i++)
    {
        task_t *task = task_for_pid(i);

        if (!task)
        {
            continue;
        }

        if (task->state == TASK_READY)
        {
            return task;
        }
    }

    for (size_t i = 1; i <= current_pid; i++)
    {
        task_t *task = task_for_pid(i);

        if (!task)
        {
            continue;
        }

        if (task->state == TASK_READY)
        {
            return task;
        }
    }

    panic("no tasks ready to schedule!");
}
```

```
#ifndef TASK_H
#define TASK_H

#include "types.h"
#include "syscall.h"

typedef struct
{
    // 0x00
    uint16_t link;
    uint16_t _res_1;
    // 0x04
    uint32_t esp0;
    // 0x08
    uint16_t ss0;
    uint16_t _res_2;
    // 0x0c
    uint32_t esp1;
    // 0x10
    uint16_t ss1;
    uint16_t _res_3;
    // 0x14
    uint32_t esp2;
    // 0x18
    uint16_t ss2;
    uint16_t _res_4;
    // 0x1c
    uint32_t cr3;
    // 0x20
    uint32_t eip;
    // 0x24
    uint32_t eflags;
    // 0x28
    uint32_t eax;
    // 0x2c
    uint32_t ecx;
    // 0x30
    uint32_t edx;
    // 0x34
    uint32_t ebx;
    // 0x38
    uint32_t esp;
    // 0x3c
    uint32_t ebp;
    // 0x40
    uint32_t esi;
    // 0x44
    uint32_t edi;
    // 0x48
    uint16_t es;
    uint16_t _res_5;
    // 0x4c
    uint16_t cs;
    uint16_t _res_6;
    // 0x50
    uint16_t ss;
    uint16_t _res_7;
    // 0x54
    uint16_t ds;
    uint16_t _res_8;
    // 0x58
    uint16_t fs;
    uint16_t _res_9;
    // 0x5c
    uint16_t gs;
    uint16_t _res_10;
    // 0x60
    uint16_t ldtr;
    uint16_t _res_11;
    // 0x64
    uint16_t _res_12;
    uint16_t iopb;
    // 0x68
} __attribute__ ((packed)) tss_t;

typedef enum
{
    TASK_READY = 1,
    TASK_BLOCK_WAIT = 2,
    TASK_EXITING = 3,
}
task_state_t;

// sched.asm refers to hardcoded offsets within this struct.
// make sure to change it when changing anything here.
typedef struct task
{
    /* 0 */ uint8_t fpu_state[512];
    /* 512 */ uint32_t esp;
    /* 516 */ uint32_t eip;
    /* 520 */ void *kernel_stack;
    // allocated within the kernel's identity-mapped region:
    /* 524 */ uint32_t *page_directory;
    /* 528 */ phys_t page_directory_phys;

    // struct members past this point may be freely rearranged without needing
    // to update any assembly source.

    registers_t *syscall_registers;

    uint32_t pid;
    uint32_t ppid;

    task_state_t state;
    uint8_t exit_status;

    union
    {
        struct
        {
            struct task *head;
            struct task *tail;
        } live;
        struct
        {
            struct task *next;
        } dead;
    } wait_queue;
}
task_t;

extern task_t *current_task;

void task_init();

void task_boot_init(const char *init_bin, size_t size) __attribute__ ((noreturn));

task_t *task_fork();

void task_kill(task_t * task, uint8_t status);

void task_destroy(task_t * task);

task_t *task_for_pid(uint32_t pid);

#endif
```

```
#ifndef TYPES_H
#define TYPES_H

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;
typedef unsigned long long uint64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed int int32_t;
typedef signed long long int64_t;

typedef uint32_t size_t;

typedef _Bool bool;
#define true  1
#define false 0

#define NULL ((void*)0)

typedef uint32_t virt_t;
typedef uint32_t phys_t;

#endif
```

```
#ifndef UTIL_H
#define UTIL_H

#include "types.h"

#define static_assert(name, expr) static char __static_assert__##name[(expr) ? 1 : -1]
__attribute__((unused))

#define countof(x) (sizeof(x) / sizeof(*(x)))

static inline size_t round_down(size_t val, size_t divisor)
{
    return val - val % divisor;
}

static inline size_t round_up(size_t val, size_t divisor)
{
    return round_down(val + divisor - 1, divisor);
}

#endif
```