```
/**

@mainpage 15-410 Project 3

@author Name1 (id1)
@author Name2 (id2)

Using assert() vs affirm(). We have used assert() to catch our own mess ups to
aid our own debugging. On the other hand, affirm() is used to catch user mess
ups when they use our interface function. For example, we affirm pointers
passed as arguments to interface functions.

Currently using README.dox as a bug tracker and TODO list.

BUGS:
    - Currently accessing invalid memory with eip. This happens during call
      to smemalign (it seems). Eip is set to a value just above USER_MEM_START,
      and at that time no user memory has been allocated. This probably means some
interrupt
      is triggering a handler which is supposed to be located at that eip.

      Very puzzling. Maybe something to do with the console/keyboard interrupts????
CK1:
    - Write gettid handler              [essential]
    - Store tid at top of kernel stack  [nice to have]
    - Pass gettid_test1
    - Cleanup unecessary files and organize into folders [nice to have]

CK2:
    - Synchronization design
    - Context switch
    - Proper physical memory management
    - Trace facility                    [nice to have]
    - Fork or exec?

TODO:
    0. Figure out synchronization mechanism

    1. Fix gettid handler not executing
        + in stack manager, create int next_tid, and use atomic_add (from p2 mutex
implementation) to get next tid. Encapsulate in function.
        + store tid at highest address of kernel stack
        + create PR for merge into main (and have other person check)

    2. Physical page allocator component
        -> free list
        -> per task allocated-list (not too sure if we need a distinction between u
ser tasks, since kernel maintains all allocated physical frames across all user tas
ks)
        -> manipulation functions (some interface)

    3. Context switcher:
        -> receiver (register restoring) function
        -> register saving function

    4. Scheduler:
        -> Figure out necessary synchronization primitives
        -> Create scheduler component (aka, new file)
        -> Responds to timer interrupts by triggering context switch every 2ms (cur
rently do it for keyboard interrupts (maybe a specific character))
        -> Keep list of active, runnable and descheduled threads

    5. Implement fork call:
        -> Figure out VM copying mechanism (working with 2 page directories at once
, one for each task).

        Implementation suggestion:
        On fork, do not reallocate memory for all kernel memory. -> Just share page
 tables below user mem_start
        memcpy(A, B);
        pd_update(); // (point A to newly allocated physical frames);
        memcpy(B, A);
        unallocate(B);

    6. Create tracing facility:
        -> 3 priorities, just call printf

TODO Andre:
    - 0
    - 3
    - 4
    - 6

TODO Nick:
    - 0
    - 1
    - 2 Awaiting review
    - 5


*/
```

```
/** @file asm_interrupt_handler.h
 *
 *  @brief helper functions to save register values before calling C
 *         interrupt handler functions, and then restore registers and return
 *         to normal execution.
 *
 */

#ifndef _P1_ASM_INTERRUPT_HANDLER_H_
#define _P1_ASM_INTERRUPT_HANDLER_H_

/** @brief Saves all register values, calls timer_int_handler(), restores
 *         all register values and returns to normal execution.
 *
 *  pusha is used to save all registers. Though not needed, %esp is also pushed
 *  onto the stack for convenience of implentation. A call is made to
 *  timer_int_handler(). When timer_int_handler() returns, popa is used
 *  to restore all registers in the correct order, and iret to return to
 *  normal execution prior to the interrupt.
 *
 *  @return Void.
 */
void call_timer_int_handler(void);

/** @brief Saves all register values, calls keybd_int_handler(), restores
 *         all register values and returns to normal execution.
 *
 *  pusha is used to save all registers. Though not needed, %esp is also pushed
 *  onto the stack for convenience of implentation. A call is made to
 *  keybd_int_handler(). When keybd_int_handler() returns, popa is used
 *  to restore all registers in the correct order, and iret to return to
 *  normal execution prior to the interrupt.
 *
 *  @return Void.
 */
void call_keybd_int_handler(void);

#endif
```

```
/** @file asm_interrupt_handler.S
 *  @brief Implementations for assembly functions
 *  @author Nicklaus Choo (nchoo)
 *  @bugs No known bugs
 */

#include <asm_interrupt_handler_template.h>

CALL_HANDLER(timer_int_handler)

CALL_HANDLER(keybd_int_handler)

CALL_W_SINGLE_ARG(test_int_handler)
```

```
#include <seg.h>

/** @brief Macro to call a particular function handler with name
 *         HANDLER_NAME
 *
 *  Macro assembly instructions require a ; after each line
 */

#include <x86/seg.h> /* SEGSEL_KERNEL_{CS, DS} */


/** @define CALL_HANDLER(HANDLER_NAME)
 *  @brief Assembly wrapper to call interupt handlers that do not service
 *         syscalls (hence the need to save _all_ general registers
 *
 *  @param HANDLER_NAME name of handler function to call
 */
#define CALL_HANDLER(HANDLER_NAME)\
\
.globl call_##HANDLER_NAME; /* create the asm function call_HANDLER_NAME */\
\
call_ ## HANDLER_NAME ## :;\
    pusha; /* Pushes all registers onto the stack */\
    pushl %ds;\
    pushl %es;\
    pushl %fs;\
    pushl %gs;\
    /* set the new values for ds, es, fs, gs */\
    movl %ss, %ax;\
    movl %ax, %ds;\
    movl %ax, %es;\
    movl %ax, %fs;\
    movl %ax, %gs;\
    call HANDLER_NAME; /* calls timer interrupt handler */\
    popl %gs;\
    popl %fs;\
    popl %es;\
    popl %ds;\
    popa; /* Restores all registers onto the stack */\
    iret; /* Return to procedure before interrupt */

#define CALL_FAULT_HANDLER_TEMPLATE(HANDLER_SPECIFIC_CODE)\
\
    /* Make use of ebp for argument access */\
    pushl %ebp;\
    movl %esp, %ebp;\
\
    /* Save all registers */\
    pusha;\
\
    /* Save all segment registers on the stack */\
    pushl %ds;\
    pushl %es;\
    pushl %fs;\
    pushl %gs;\
\
    /* set the new values for ds, es, fs, gs */\
    /* TODO fix so clang does not complain */\
    movl %ss, %ax;\
    movl %ax, %ds;\
    movl %ax, %es;\
    movl %ax, %fs;\
    movl %ax, %gs;\
\
    HANDLER_SPECIFIC_CODE\
\
    /* Restores all segment registers from the stack */\
    popl %gs;\
    popl %fs;\
    popl %es;\
    popl %ds;\
\
    /* Restores all callee save registers from the stack */\
    popa;\
\
    /* Restore ebp */\
    popl %ebp;\
\
    /* Return to procedure before interrupt */\
    iret;

#define CALL_FAULT_HANDLER_TEMPLATE_W_ERROR(HANDLER_SPECIFIC_CODE)\
\
    /* Make use of ebp for argument access */\
    pushl %ebp;\
    movl %esp, %ebp;\
\
    /* Save all registers */\
    pusha;\
\
    /* Save all segment registers on the stack */\
    pushl %ds;\
    pushl %es;\
    pushl %fs;\
    pushl %gs;\
\
    /* set the new values for ds, es, fs, gs */\
    /* TODO fix so clang does not complain */\
    movl %ss, %ax;\
    movl %ax, %ds;\
    movl %ax, %es;\
    movl %ax, %fs;\
    movl %ax, %gs;\
\
    HANDLER_SPECIFIC_CODE\
\
    /* Restores all segment registers from the stack */\
    popl %gs;\
    popl %fs;\
    popl %es;\
    popl %ds;\
\
    /* Restores all callee save registers from the stack */\
    popa;\
\
    /* Restore ebp */\
    popl %ebp;\
\
    /* Return to procedure before interrupt */\
    addl $4, %esp;\
    iret;

/** @define CALL_HANDLER_TEMPLATE(HANDLER_SPECIFIC_CODE)
 *  @brief Wrapper code for syscall assembly wrappers that saves and restores
 *         general and segment registers.
 *
 *  @param HANDLER_SPECIFIC_CODE syscall handler specific wrapper code
 */
#define CALL_HANDLER_TEMPLATE(HANDLER_SPECIFIC_CODE)\
\
    /* Save all callee save registers */\
    pushl %ebp;\
    movl %esp, %ebp;\
    pushl %edi;\
```

```
    pushl %ebx;\
    pushl %esi;\
\
    /* Save all segment registers on the stack */\
    pushl %ds;\
    pushl %es;\
    pushl %fs;\
    pushl %gs;\
\
    /* set the new values for ds, es, fs, gs */\
    /* TODO fix so clang does not complain */\
    movl %ss, %ax;\
    movl %ax, %ds;\
    movl %ax, %es;\
    movl %ax, %fs;\
    movl %ax, %gs;\
\
    HANDLER_SPECIFIC_CODE\
\
    /* Restores all segment registers from the stack */\
    popl %gs;\
    popl %fs;\
    popl %es;\
    popl %ds;\
\
    /* Restores all callee save registers from the stack */\
    popl %esi;\
    popl %ebx;\
    popl %edi;\
    popl %ebp;\
\
    /* Return to procedure before interrupt */\
    iret;

/** @define CALL_W_RETVAL_HANDLER(HANDLER_NAME)
 *  @brief Assembly wrapper for a syscall handler that returns a value,
 *         takes in no arguments
 */
#define CALL_W_RETVAL_HANDLER(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_HANDLER_TEMPLATE\
    (\
        /* Call syscall handler */\
        call HANDLER_NAME;\
    )

#define SINGLE_MACRO_ARG_W_COMMAS(...) __VA_ARGS__

/** @define CALL_W_SINGLE_ARG(HANDLER_NAME)
 *  @brief Assembly wrapper for a syscall handler that takes in 1 argument.
 *
 *  Call convention dictates single argument is found in %esi
 *
 *  @param HANDLER_NAME handler name to call
 */
#define CALL_W_SINGLE_ARG(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_HANDLER_TEMPLATE(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl %esi;         /* push argument onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $4, %esp;      /* ignore argument */\
    ))

/** @define CALL_FAULT_HANDLER(HANDLER_NAME)
 *  @brief Assembly wrapper for a fault handler that takes in error code.
 *
 *  Error code is put on stack by processor before calling fault handler
 *
 *  @param HANDLER_NAME handler name to call
 */
#define CALL_FAULT_HANDLER(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_FAULT_HANDLER_TEMPLATE(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl 8(%ebp);      /* push cs onto stack */\
        pushl 4(%ebp);      /* push eip onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $8, %esp;      /* ignore arguments */\
    ))

/** @define CALL_FAULT_HANDLER_W_ERROR_CODE(HANDLER_NAME)
 *  @brief Assembly wrapper for a fault handler that takes in error code.
 *
 *  Error code is put on stack by processor before calling fault handler
 *
 *  @param HANDLER_NAME handler name to call
 */
#define CALL_FAULT_HANDLER_W_ERROR_CODE(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_FAULT_HANDLER_TEMPLATE_W_ERROR(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl 12(%ebp);     /* push cs onto stack */\
        pushl 8(%ebp);      /* push eip onto stack */\
        pushl 4(%ebp);      /* push error code onto stack  */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $12, %esp;     /* ignore arguments */\
    ))

#define CALL_VAR_ARGS_FAULT_HANDLER(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_FAULT_HANDLER_TEMPLATE(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl %ebp;         /* push ebp onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $4, %esp;      /* ignore arguments */\
    ))

#define CALL_VAR_ARGS_FAULT_HANDLER_W_ERROR(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
```

```
    CALL_FAULT_HANDLER_TEMPLATE_W_ERROR(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl %ebp;         /* push ebp onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $4, %esp;      /* ignore arguments */\
    ))


/** @def CALL_W_DOUBLE_ARG(HANDLER_NAME)
 *  @brief Macro for assembly wrapper for calling a syscall with 2 arguments
 *
 *  @param HANDLER_NAME handler name to call
 */
#define CALL_W_DOUBLE_ARG(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_HANDLER_TEMPLATE(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl 4(%esi);      /* push 2nd argument onto stack */\
        pushl (%esi);       /* push 1st argument onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $8, %esp;      /* ignore arguments */\
    ))


/** @def CALL_W_FOUR_ARG(HANDLER_NAME)
 *  @brief Macro for assembly wrapper for calling a syscall with 4 arguments
 *
 *  @param HANDLER_NAME handler name to call
 */
#define CALL_W_FOUR_ARG(HANDLER_NAME)\
\
/* Declare and define asm function call_HANDLER_NAME */\
.globl call_##HANDLER_NAME;\
call_ ## HANDLER_NAME ## :;\
\
    CALL_HANDLER_TEMPLATE(SINGLE_MACRO_ARG_W_COMMAS\
    (\
        pushl 12(%esi);     /* push 4th argument onto stack */\
        pushl 8(%esi);      /* push 3rd argument onto stack */\
        pushl 4(%esi);      /* push 2nd argument onto stack */\
        pushl (%esi);       /* push 1st argument onto stack */\
        call HANDLER_NAME;  /* calls syscall handler */\
        addl $16, %esp;     /* ignore arguments */\
    ))
```

```
.globl compare_and_swap_atomic
.globl add_one_atomic

/* int add_one_atomic( int *at ) */
add_one_atomic:
    movl 4(%esp), %ecx      /* Move address argument to ecx */
    movl $1, %eax           /* eax = 1 */
    lock xaddl %eax, (%ecx) /* Atomically: temp = eax + *ecx; eax = *ecx; *ecx = te
mp */
    ret

/* int CAS( int *at, int expect, int new_val ) */
compare_and_swap_atomic:
    mov 4(%esp), %edx           /* Load at into edx */
    mov 8(%esp), %eax           /* Get expect into ax */
    mov 12(%esp), %ecx          /* Get new_val into cx */
    lock cmpxchg %ecx, (%edx)   /* Atomically: if (expect == *addr)
                                           then %ax = *addr, *addr = new_val */
    ret
```

```
/** @file console.c
 *  @brief Implements functions in console.h
 *
 *  Since console.h contains comments, added comments will be written
 *  with this format with preceding and succeeding --
 *
 *  --
 *  < additional comments >
 *  --
 *
 *  @author Andre Nascimento (anascime)
 *  @author Nicklaus Choo (nchoo)
 */

#include <console.h>
#include <asm.h>            /* outb() */
#include <string.h>         /* memmove () */
#include <assert.h>         /* assert(), affirm() */
#include <video_defines.h>  /* CONSOLE_HEIGHT, CONSOLE_WIDTH */
#include <lib_thread_management/mutex.h> /* mutex_t */

static mutex_t draw_char_mux;
static mutex_t cursor_mux;

/* Default global console color. */
static int console_color = BGND_BLACK | FGND_WHITE;

/* Logical cursor row starts off at 0 */
static int cursor_row = 0;

/* Logical cursor col starts off at 0 */
static int cursor_col = 0;

/* Boolean for if cursor is hidden */
static int cursor_hidden = 0;

/* Background color mask to extract invalid set bits in color. FFFF FF00 */
#define INVALID_COLOR 0xFFFFFF00


void
init_console( void )
{
    mutex_init(&draw_char_mux);
    mutex_init(&cursor_mux);
    clear_console();
}


/** @brief Helper function to check if (row, col) is onscreen
 *
 *  @param row Row index
 *  @param col Col index
 *  @return 1 if onscreen, 0 otherwise
 */
static int
onscreen(int row, int col) {
    return 0 <= row && row < CONSOLE_HEIGHT && 0 <= col && col < CONSOLE_WIDTH;
}

/** @brief Sets the hardware cursor to any row or column. Should only be called
 *         by hide_cursor(), unhide_cursor(), set_cursor().
 *
 *  Invariant for row and col is such that the cursor is only ever set to
 *  be onscreen, or offscreen specifically at (CONSOLE_HEIGHT, CONSOLE_WIDTH).
 *
 *  @param row Row to set hardware cursor to.
 *  @param col Column to set hardware cursor to.
 *  @return Void.
 */
static void
set_hardware_cursor( int row, int col )
{
    /* Only values for row, col if called by hide, unhide, set cursor functions */
    assert(onscreen(row, col) || (row == CONSOLE_HEIGHT && col == CONSOLE_WIDTH));

    /* Calculate offset in row major form */
    short hardware_cursor_offset = row * CONSOLE_WIDTH + col;

    /* Set lower 8 bits */
    outb(CRTC_IDX_REG, CRTC_CURSOR_LSB_IDX);
    outb(CRTC_DATA_REG, hardware_cursor_offset);

    /* Set upper 8 bits */
    outb(CRTC_IDX_REG, CRTC_CURSOR_MSB_IDX);
    outb(CRTC_DATA_REG, hardware_cursor_offset >> 8);
}

/** @brief Scrolls the terminal up by 1 line and ensures that the position of
 *         the logical cursor remains fixed with respect to console output
 *
 *  @return Void.
 */
static void
scroll( void )
{
    /* Move screen contents all up 1 row. */
    memmove((void *) CONSOLE_MEM_BASE,
            (void *) (CONSOLE_MEM_BASE + 2 * CONSOLE_WIDTH),
            2 * CONSOLE_WIDTH * (CONSOLE_HEIGHT - 1));

    /* The new last row should be an empty row of spaces */
    for (int col = 0; col < CONSOLE_WIDTH; col++) {
        draw_char(CONSOLE_HEIGHT - 1, col, ' ', console_color);
    }
    /* Cursor is stationary relative to output. */
    set_cursor(cursor_row - 1, cursor_col);
}

/** @brief Modification of the putbyte() specification which takes in
 *         arguments for starting row and column. Useful for readline()
 *
 *  putbyte() is a wrapper around this function.
 *
 *  @param ch The character to print
 *  @param start_rowp Pointer to starting row
 *  @param start_colp Pointer to starting column
 *  @return The input character
 */
int
scrolled_putbyte( char ch, int *start_rowp, int *start_colp )
{
    assert(start_rowp);
    assert(start_colp);
    assert(onscreen(*start_rowp, *start_colp));
    assert(onscreen(cursor_row, cursor_col));

    switch (ch) {
        case '\n': {

            /* Scroll if at screen bottom */
            if (cursor_row + 1 >= CONSOLE_HEIGHT) {
                scroll();
                // TODO you can scroll offscreen though
```

```
                *start_rowp -= 1;
            }
            /* Always update the cursor position relative to content */
            draw_char(cursor_row + 1, 0, ' ', console_color);
            set_cursor(cursor_row + 1, 0);
            break;
        }
        case '\r': {
            set_cursor(*start_rowp, *start_colp);
            break;
        }
        case '\b': {

            /* Not at leftmost column */
            if (cursor_col > 0) {

              /* Always draw space before moving else cursor blotted out */
              draw_char(cursor_row, cursor_col - 1, ' ', console_color);
              set_cursor(cursor_row, cursor_col - 1);

            /* At leftmost column, backspace goes to previous row if not top */
            } else {
                if (cursor_row > 0) {
                    draw_char(cursor_row - 1, CONSOLE_WIDTH - 1, ' ',
                              console_color);
                    set_cursor(cursor_row - 1, CONSOLE_WIDTH - 1);
                }
            }
            break;
        }
        default: {

            /* Print the character */
            draw_char(cursor_row, cursor_col, ch, console_color);

            /* If we are at the end of a line, set cursor on new line */
            if (cursor_col + 1 >= CONSOLE_WIDTH) {

                /* Scroll if necessary */
                if (cursor_row + 1 >= CONSOLE_HEIGHT) {
                    scroll();
                    *start_rowp -= 1;
                }
                /* Start printing below, update color if needed */
                char next_ch = get_char(cursor_row + 1, 0);
                draw_char(cursor_row + 1, 0, next_ch, console_color);
                set_cursor(cursor_row + 1, 0);
            } else {
                char next_ch = get_char(cursor_row, cursor_col + 1);
                draw_char(cursor_row, cursor_col + 1, next_ch, console_color);
                set_cursor(cursor_row, cursor_col + 1);
            }
            break;
        }
    }
    assert(onscreen(cursor_row, cursor_col));
    return ch;
}

/** @brief Prints character ch at the current location
 *         of the cursor.
 *
 *  If the character is a newline ('\n'), the cursor is moved
 *  to the beginning of the next line (scrolling if necessary).
 *  If the character is a carriage return ('\r'), the cursor is
 *  immediately reset to the beginning of the current line,
 *  causing any future output to overwrite any existing output
 *  on the line.  If backspace ('\b') is encountered, the previous
 *  character is erased.  See the main console.c description found
 *  on the handout web page for more backspace behavior.
 *  --
 *  We move the cursors as we write. Since there is no notion of a console
 *  prompt for putbyte(), the call to scrolled_putbyte() will have
 *  start_row == cursor_row and start_col == 0
 *  --
 *
 *  @param ch the character to print
 *  @return The input character
 */
int putbyte(char ch) {

    /* Get starting row and column, but set starting column to 0 */
    int start_row;
    int start_col;
    get_cursor(&start_row, &start_col);
    start_col = 0;

    return scrolled_putbyte(ch, &start_row, &start_col);
}
/** @brief Prints the string s, starting at the current
 *         location of the cursor.
 *
 *  If the string is longer than the current line, the
 *  string fills up the current line and then
 *  continues on the next line. If the string exceeds
 *  available space on the entire console, the screen
 *  scrolls up one line, and then the string
 *  continues on the new line.  If '\n', '\r', and '\b' are
 *  encountered within the string, they are handled
 *  as per putbyte. If len is not a positive integer or s
 *  is null, the function has no effect.
 *
 *  @param s The string to be printed.
 *  @param len The length of the string s.
 *  @return Void.
 */
void
putbytes( const char *s, int len )
{
    affirm(s);
    if (len < 0) {
        return;
    }
    /* s is null string doesn't mean s == NULL */
    if (s[0] == '\0') {
        return;
    }
    for (int i = 0; i < len; i++) {
        char ch = s[i];
        putbyte(ch);
    }
}
/** @brief Changes the foreground and background color
 *         of future characters printed on the console.
 *
 *  If the color code is invalid, the function has no effect.
 *
 *  NOTE: No need for synchronization here, since it's just
 *  a blind write over the current color
 *
 *  @param color The new color code.
 *  @return 0 on success or integer error code less than 0 if
 *          color code is invalid.
 */
```

```
int
set_term_color( int color )
{
    /* No effect if invalid color passed */
    if (color & INVALID_COLOR) {
        return -1;
    }
    /* Else set console_color */
    console_color = color;
    return 0;
}

/** @brief Writes the current foreground and background
 *         color of characters printed on the console
 *         into the argument color.
 *  @param color The address to which the current color
 *         information will be written.
 *  @return Void.
 */
void
get_term_color( int* color )
{
    affirm(color);
    *color = console_color;
}

/** @brief Sets the position of the cursor to the
 *         position (row, col).
 *
 *  Subsequent calls to putbytes should cause the console
 *  output to begin at the new position. If the cursor is
 *  currently hidden, a call to set_cursor() does not show
 *  the cursor.
 *  --
 *  If cursor_hidden, the logical cursor is set without setting the
 *  hardware cursor. This is because the hardware cursor is always
 *  the one that is visible.
 *
 *  Else, if there is a change in row, col, the logical cursor and the hardware
 *  cursor are set.
 *  --
 *
 *  @param row The new row for the cursor.
 *  @param col The new column for the cursor.
 *  @return 0 on success or integer error code less than 0 if
 *          cursor location is invalid.
 */
int
set_cursor( int row, int col )
{
    mutex_lock(&cursor_mux);
    assert(onscreen(cursor_row, cursor_col));
    /* set logical cursor */
    if (onscreen(row, col)) {
        cursor_row = row;
        cursor_col = col;

        /* If cursor not hidden and change in row, col, set hardware cursor */
        if (!cursor_hidden && (cursor_row != row || cursor_col != col)) {
            set_hardware_cursor(row, col);
        }
        assert(onscreen(cursor_row, cursor_col));
        mutex_unlock(&cursor_mux);
        return 0;
    }
    /* cursor location is invalid, do nothing and return -1 */
    assert(onscreen(cursor_row, cursor_col));
    mutex_unlock(&cursor_mux);
    return -1;
}

/** @brief Writes the current position of the cursor
 *         into the arguments row and col.
 *
 *  --
 *  Only writes to row, col if they are non-null, throws affirm() error
 *  otherwise.
 *  --
 *
 *  @param row The address to which the current cursor
 *         row will be written.
 *  @param col The address to which the current cursor
 *         column will be written.
 *  @return Void.
 */
void
get_cursor( int* row, int* col )
{
    mutex_lock(&cursor_mux);
    affirm(row);
    affirm(col);
    *row = cursor_row;
    *col = cursor_col;
    mutex_unlock(&cursor_mux);
}

/** @brief Shows the cursor.
 *
 *  If the cursor is already shown, the function has no effect.
 *  --
 *  Hides the cursor by setting the hardware cursor to
 *  (CONSOLE_HEIGHT, CONSOLE_WIDTH) and toggles cursor_hidden to true i.e. 1.
 *  Note that this function is idempotent.
 *  --
 *
 *  @return Void.
 */
void
hide_cursor( void )
{
    assert(onscreen(cursor_row, cursor_col));
    set_hardware_cursor(CONSOLE_HEIGHT, CONSOLE_WIDTH);
    cursor_hidden = 1;
}

/** @brief Shows the cursor.
 *
 *  If the cursor is already shown, the function has no effect.
 *  --
 *  Shows the cursor by setting the hardware cursor to the current location
 *  of the logical cursor and toggles cursor_hidden to false i.e. 0.
 *  Note that this function is idempotent.
 *  --
 *
 *  @return Void.
 */
void
show_cursor( void )
{
    assert(onscreen(cursor_row, cursor_col));
    set_hardware_cursor(cursor_row, cursor_col);
    cursor_hidden = 0;
}
```

```
/** @brief Clears the entire console.
 *
 * The cursor is reset to the first row and column
 *
 *  @return Void.
 */
void
clear_console( void )
{
    /* Replace everything onscreen with a blank space */
    for (size_t row = 0; row < CONSOLE_HEIGHT; row++) {
        for (size_t col = 0; col < CONSOLE_WIDTH; col++) {
            draw_char(row, col, ' ', console_color);
        }
    }
    /* Set cursor to the top left corner */
    set_cursor(0, 0);
}

/** @brief Prints character ch with the specified color
 *         at position (row, col).
 *
 *  If any argument is invalid, the function has no effect.
 *
 *  @param row The row in which to display the character.
 *  @param col The column in which to display the character.
 *  @param ch The character to display.
 *  @param color The color to use to display the character.
 *  @return Void.
 */
void
draw_char( int row, int col, int ch, int color )
{
    mutex_lock(&draw_char_mux);
    /* If row or col out of range, invalid row, no effect. */
    if (!onscreen(row, col)) {
        goto draw_char_cleanup;
    }
    /* If background color not supported, invalid color, no effect. */
    if (color & INVALID_COLOR) {
        goto draw_char_cleanup;
    }
    /* All arguments valid, draw character with specified color */
    char *chp = (char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col));
    *chp = ch;
    *(chp + 1) = color;

draw_char_cleanup:
    mutex_unlock(&draw_char_mux);
}

/** @brief Returns the character displayed at position (row, col).
 *
 *  --
 *  Offscreen characters are always NULL or simply 0 by default.
 *  --
 *
 *  @param row Row of the character.
 *  @param col Column of the character.
 *  @return The character at (row, col).
 */
char
get_char( int row, int col )
{
    /* If out of range, return 0. */
    if (!onscreen(row, col)) {
        return 0;
    }
    /* Else return char at row, col. */
    return *(char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col));
}
```

```
#include <seg.h>

/**
 * Since we direct map kernel memory, changing the value in cr3 should
 * have no effect on the stack values being stored on the kernel stack
 */
.globl context_switch

context_switch:
    pushl %ebp
    movl %esp, %ebp /* update %ebp */
    pushl %eax
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %edi
    pushl %esi

    movl %cr3, %ecx
    pushl %ecx

    movl 8(%ebp), %ecx
    movl %esp, (%ecx)   /* save current %esp to first arg */
    movl 12(%ebp), %esp /* restore %esp from second arg */

    popl %ecx
    movl %ecx, %cr3     /* update page directory */

    popl %esi
    popl %edi
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax
    popl %ebp

    sti     /* enable interrupts */

    ret
```

```c
/** @file alignment_check_handler.c
 *  @brief Functions for alignment check faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */
#include <simics.h>

void
alignment_check_handler( int error_code, int eip, int cs )
{
    assert(error_code == 0);
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Alignment check fault  encountered error at "
                "0x%x.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("[User mode] Alignment check fault encountered at 0x%x", eip);
}
```

```
#include <asm_interrupt_handler_template.h>

CALL_FAULT_HANDLER(divide_handler)
CALL_FAULT_HANDLER(debug_handler)
CALL_FAULT_HANDLER(breakpoint_handler)
CALL_FAULT_HANDLER(overflow_handler)
CALL_FAULT_HANDLER(bound_handler)
CALL_FAULT_HANDLER(invalid_opcode_handler)
CALL_FAULT_HANDLER(float_handler)
CALL_FAULT_HANDLER_W_ERROR_CODE(segment_not_present_handler)
CALL_FAULT_HANDLER_W_ERROR_CODE(stack_fault_handler)
CALL_VAR_ARGS_FAULT_HANDLER_W_ERROR(general_protection_handler)
CALL_FAULT_HANDLER_W_ERROR_CODE(alignment_check_handler)
CALL_FAULT_HANDLER(non_maskable_handler)
CALL_FAULT_HANDLER(machine_check_handler)
```

```
/** @file bound_handler.c
 *  @brief Functions for handling bound range faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
bound_handler( int eip, int cs)
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Bound-range-exceeded fault encountered at 0x%x."
                "Please contact kernel developers.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled bound-range-exceeded fault encountered at 0x%x", eip);
}
```

```
/** @file breakpoint_handler.c
 *  @brief Functions for handling breakpoint traps
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
breakpoint_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Breakpoint encountered at 0x%x."
                "Please contact kernel developers.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled breakpoint fault encountered before 0x%x", eip);
}
```

```
/** @file debug_handler.c
 *  @brief Functions for handling debug traps/faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
debug_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Debug condition encountered at 0x%x."
                "Please contact kernel developers.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled debug trap or fault encountered at 0x%x", eip);
}
```

```
/** @file divide_handler.c
 *  @brief Functions for handling division faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

/** @brief Prints out the offending address on and calls panic()
 *
 *  @return Void.
 */
void
divide_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Divide by 0 exception at 0x%x."
                "Please contact kernel developers.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled divide by 0 exception at instruction 0x%x", eip);
}
```

```
/** @file float_handler.c
 *  @brief Functions for handling device not available faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
float_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Floating point operation encountered at 0x%x."
                "Please contact kernel developers.", eip);
    }

    /* acknowledge signal and call user handler?
     * OR
     * acknowledge signal and just kill user thread? */

    panic("Unhandled device not available fault (due to floating-point op)"
            " at instruction 0x%x", eip);
}
```

```
/** @file general_protection_handler.c
 *  @brief Functions for alignment check faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */
#include <simics.h>
#include <stdint.h> /* uint32_t */
void
general_protection_handler( uint32_t *ebp )
{
    int error_code = *(ebp + 1);
    int eip = *(ebp + 2);
    int cs = *(ebp + 3);
    int eflags = *(ebp + 4);
    int esp, ss;

    /* More args on stack if it was from user space */
    if (cs == SEGSEL_USER_CS) {
        esp = *(ebp + 5);
        ss = *(ebp + 6);
        affirm_msg(error_code == 0, "General protection fault while loading a "
                   "segment descriptor\n"
                   "error_code:0x%08x\n "
                   "eip:0x%08x\n "
                   "cs:0x%08x\n "
                   "eflags:0x%08x\n "
                   "esp:0x%08x\n "
                   "ss:0x%08x\n ",
                   error_code, eip, cs, eflags, esp, ss);
    } else {
        affirm_msg(error_code == 0, "General protection fault while loading a "
                   "segment descriptor\n"
                   "error_code:0x%08x\n "
                   "eip:0x%08x\n "
                   "cs:0x%08x\n "
                   "eflags:0x%08x\n ",
                   error_code, eip, cs, eflags);
    }
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] General protection fault encountered error at "
                   "segment descriptor\n"
                   "error_code:0x%08x\n "
                   "eip:0x%08x\n "
                   "cs:0x%08x\n "
                   "eflags:0x%08x\n ",
                   error_code, eip, cs, eflags);

    }
    /* TODO: acknowledge signal and call user handler  */
    panic("[User mode] General protection fault encountered at "
                   "segment descriptor\n"
                   "error_code:0x%08x\n "
                   "eip:0x%08x\n "
                   "cs:0x%08x\n "
                   "eflags:0x%08x\n ",
                   error_code, eip, cs, eflags);


}
```

```
/** @file invalid_opcode_handler.c
 *  @brief Functions for handling invalid opcode faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
invalid_opcode_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Invalid opcode fault encountered at 0x%x."
                "Please contact kernel developers.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled invalid opcode fault encountered at 0x%x", eip);

}
```

```
/** @file machine_check_handler.c
 *  @brief Functions for handling overflow traps
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */
#include <simics.h>

void
machine_check_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Machine check error encountered at 0x%x.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("[User mode] Machine check error encountered at 0x%x", eip);
}
```

```
/** @file non_maskable_handler.c
 *  @brief Functions for NMIs
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */
#include <simics.h>

void
non_maskable_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] NMI encountered at 0x%x.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("[User mode] NMI encountered at 0x%x", eip);
}
```

```
/** @file overflow_handler.c
 *  @brief Functions for handling overflow traps
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */
#include <simics.h>

void
overflow_handler( int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Overflow encountered at 0x%x.", eip);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("[User mode] Unhandled overflow fault encountered at 0x%x", eip);
}
```

```
/** @file segment_not_present_handler.c
 *  @brief Functions for handling segment not present faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
segment_not_present_handler( int error_code, int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Segment not present fault encountered at 0x%x "
                "for segment with index %d", eip, error_code);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled segment not present fault encountered at 0x%x "
            "for segment with index %d", eip, error_code);
}
```

```
/** @file stack_fault_handler.c
 *  @brief Functions for handling stack faults
 */
#include <seg.h>    /* SEGSEL_KERNEL_CS */
#include <assert.h> /* panic() */

void
stack_fault_handler( int error_code, int eip, int cs )
{
    if (cs == SEGSEL_KERNEL_CS) {
        panic("[Kernel mode] Stack fault encountered at 0x%x "
                "for stack with segment %d", eip, error_code);
    }
    /* TODO: acknowledge signal and call user handler  */

    panic("Unhandled segment not present fault encountered at 0x%x "
            "for stack with segment %d", eip, error_code);
}
```

```
/** @file asm_console_handlers.h
 *  @brief Assembly wrapper to call console syscall handlers
 */

#ifndef ASM_CONSOLE_HANDLERS_H_
#define ASM_CONSOLE_HANDLERS_H_

void call_print(void);
void call_readline(void);
void call_get_cursor_pos(void);
void call_set_cursor_pos(void);
void call_set_term_color_handler(void);

#endif /* ASM_CONSOLE_HANDLERS_H_ */
```

```
/** @file asm_fault_handlers.h
 *  @brief Fault handlers */

#ifndef ASM_FAULT_HANDLERS_H_
#define ASM_FAULT_HANDLERS_H_

void call_divide_handler( void );
void call_debug_handler( void );
void call_breakpoint_handler( void );
void call_overflow_handler( void );
void call_bound_handler( void );
void call_invalid_opcode_handler( void );
void call_float_handler( void );
void call_segment_not_present_handler( void );
void call_stack_fault_handler( void );
void call_general_protection_handler( void );
void call_alignment_check_handler( void );
void call_non_maskable_handler( void );
void call_machine_check_handler( void );

#endif /* ASM_FAULT_HANDLERS_H_ */
```

```
/** @file asm_life_cycle_handlers.h
 *  @brief Assembly wrapper to call life cycle syscall handlers
 *
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef ASM_LIFE_CYCLE_HANDLERS_H_
#define ASM_LIFE_CYCLE_HANDLERS_H_

extern void call_fork( void );
extern void call_exec( void );
extern void call_vanish( void );
extern void call_task_vanish( void );
extern void call_set_status( void );

#endif /* ASM_LIFE_CYCLE_HANDLERS_H_ */
```

```
/** @file asm_memory_management_handlers.h
 *  @brief Assembly wrapper to call memory management syscall handlers
 *
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef ASM_MEMORY_MANAGEMENT_HANDLERS_H_
#define ASM_MEMORY_MANAGEMENT_HANDLERS_H_

/** @brief Assembly wrapper for calling pagefault_handler() interrupt handler
 */
void call_pagefault_handler( void );

void call_new_pages( void );

void call_remove_pages( void );

#endif /* ASM_MEMORY_MANAGEMENT_HANDLERS_H_ */
```

```
/** @file asm_misc_handlers.h
 *  @brief Assembly wrappers to call misc syscall handlers
 */

#ifndef ASM_MISC_HANDLERS_H_
#define ASM_MISC_HANDLERS_H

void call_halt(void);
void call_readfile(void);

#endif /* ASM_MISC_HANDLERS_H_ */
```

```
/** @file asm_thread_management_handlers.h
 *  @brief Assembly wrapper to call syscall handlers
 *
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef ASM_THREAD_MANAGEMENT_HANDLERS_H_
#define ASM_THREAD_MANAGEMENT_HANDLERS_H_

extern void call_gettid( void );

extern void call_get_ticks( void );

extern void call_yield( void );

extern void call_deschedule( void );

extern void call_make_runnable( void );

extern void call_sleep( void );

#endif /* ASM_THREAD_MANAGEMENT_HANDLERS_H_ */
```

```
/** @file atomic_utils.h
 *  @brief Atomic utilities. */

#ifndef ATOMIC_UTILS_H_
#define ATOMIC_UTILS_H_

#include <stdint.h> /* uin32_t */

uint32_t compare_and_swap_atomic( uint32_t *at,
        uint32_t expect, uint32_t new_val );
uint32_t add_one_atomic( uint32_t *at );

#endif /* ATOMIC_UTILS_H_ */
```

```
/*
 *
 *  #     #
 *  ##    #   ####    #####     #     ####   ######
 *  # #   #  #    #     #       #    #    #  #
 *  #  #  #  #    #     #       #    #       #####
 *  #   # #  #    #     #       #    #       #
 *  #    ##  #    #     #       #    #    #  #
 *  #     #   ####      #       #     ####   ######
 *
 * Now that it's P3 instead of P1 you are allowed
 * to edit this file if it suits you.
 *
 * Please delete this notice.
 *
 */

/** @file console.h
 *  @brief Function prototypes for the console driver.
 *
 *  This contains the prototypes and global variables for the console
 *  driver
 *
 *  @author Michael Berman (mberman)
 *  @bug No known bugs.
 */

#ifndef _CONSOLE_H
#define _CONSOLE_H

#include <video_defines.h>

void init_console( void );

/** @brief Prints character ch at the current location
 *         of the cursor.
 *
 *  If the character is a newline ('\n'), the cursor is
 *  be moved to the beginning of the next line (scrolling if necessary).  If
 *  the character is a carriage return ('\r'), the cursor
 *  is immediately reset to the beginning of the current
 *  line, causing any future output to overwrite any existing
 *  output on the line.  If backsapce ('\b') is encountered,
 *  the previous character is erased.  See the main console.c description
 *  for more backspace behavior.
 *
 *  @param ch the character to print
 *  @return The input character
 */
int putbyte( char ch );

/** @brief Prints the string s, starting at the current
 *         location of the cursor.
 *
 *  If the string is longer than the current line, the
 *  string fills up the current line and then
 *  continues on the next line. If the string exceeds
 *  available space on the entire console, the screen
 *  scrolls up one line, and then the string
 *  continues on the new line.  If '\n', '\r', and '\b' are
 *  encountered within the string, they are handled
 *  as per putbyte. If len is not a positive integer or s
 *  is null, the function has no effect.
 *
 *  @param s The string to be printed.
 *  @param len The length of the string s.
 *  @return Void.
 */
void putbytes(const char* s, int len);

/** @brief Changes the foreground and background color
 *         of future characters printed on the console.
 *
 *  If the color code is invalid, the function has no effect.
 *
 *  @param color The new color code.
 *  @return 0 on success or integer error code less than 0 if
 *          color code is invalid.
 */
int set_term_color(int color);

/** @brief Writes the current foreground and background
 *         color of characters printed on the console
 *         into the argument color.
 *  @param color The address to which the current color
 *         information will be written.
 *  @return Void.
 */
void get_term_color(int* color);

/** @brief Sets the position of the cursor to the
 *         position (row, col).
 *
 *  Subsequent calls to putbytes should cause the console
 *  output to begin at the new position. If the cursor is
 *  currently hidden, a call to set_cursor() does not show
 *  the cursor.
 *
 *  @param row The new row for the cursor.
 *  @param col The new column for the cursor.
 *  @return 0 on success or integer error code less than 0 if
 *          cursor location is invalid.
 */
int set_cursor(int row, int col);

/** @brief Writes the current position of the cursor
 *         into the arguments row and col.
 *  @param row The address to which the current cursor
 *         row will be written.
 *  @param col The address to which the current cursor
 *         column will be written.
 *  @return Void.
 */
void get_cursor(int* row, int* col);

/** @brief Hides the cursor.
 *
 *  Subsequent calls to putbytes do not cause the
 *  cursor to show again.
 *
 *  @return Void.
 */
void hide_cursor(void);

/** @brief Shows the cursor.
 *
 *  If the cursor is already shown, the function has no effect.
 *
 *  @return Void.
 */
void show_cursor(void);

/** @brief Clears the entire console.
 *
```

```
 * The cursor is reset to the first row and column
 *
 *  @return Void.
 */
void clear_console(void);

/** @brief Prints character ch with the specified color
 *         at position (row, col).
 *
 *  If any argument is invalid, the function has no effect.
 *
 *  @param row The row in which to display the character.
 *  @param col The column in which to display the character.
 *  @param ch The character to display.
 *  @param color The color to use to display the character.
 *  @return Void.
 */
void draw_char(int row, int col, int ch, int color);

/** @brief Returns the character displayed at position (row, col).
 *  @param row Row of the character.
 *  @param col Column of the character.
 *  @return The character at (row, col).
 */
char get_char(int row, int col);

/* helper for keybd */
int scrolled_putbyte( char ch, int *start_rowp, int *start_colp );

#endif /* _CONSOLE_H */
```

```
#ifndef _CONTEXT_SWITCH_H_
#define _CONTEXT_SWITCH_H_

void context_switch( void **save_esp, void *restore_esp );


#endif /* _CONTEXT_SWITCH_H_ */
```

```
#ifndef _INSTALL_HANDLER_H_
#define _INSTALL_HANDLER_H_

/* Number of bits in a byte */
#define BYTE_LEN 8

/* Number of bytes that a trap gate occupies */
#define BYTES_PER_GATE 8

/* Mask for function handler address for upper bits */
#define OFFSET_UPPER_MASK   0xFFFF0000

/* Mask for function handler address for lower bits */
#define OFFSET_LOWER_MASK   0x0000FFFF

/* Trap gate flag masks */
#define PRESENT             0x00008000
#define DPL_0               (0x00000000 << 13)
#define DPL_3               (0x00000003 << 13)

#define D16                 0x00000700
#define D32_TRAP (0xF << 8)
#define D32_INTERRUPT (0xE << 8)

#define RESERVED_UPPER_MASK 0x0000000F

/* Hander installation error codes */
#define E_NO_INSTALL_KEYBOARD_HANDLER -2
#define E_NO_INSTALL_TIMER_HANDLER -3

/* Type of assembly wrapper for C interrupt handler functions */
typedef void asm_wrapper_t(void);
typedef void init_func_t(void);

int handler_install(void (*tick)(unsigned int));
int install_handler_in_idt( int idt_entry, asm_wrapper_t *asm_wrapper, int dpl,
                            int gate_type );
int install_handler( int idt_entry, init_func_t *init,
                     asm_wrapper_t *asm_wrapper, int dpl, int gate_type );
#endif
```

```
/** @file iret_travel.h
 *  @brief Asm facilities for moving to user mode */

#ifndef _IRET_TRAVEL_H
#define _IRET_TRAVEL_H

#include <stdint.h> /* uint32_t */

/** @brief Sets processor state. This can be used for "travelling"
 *         to user mode. Only to be called from kernel mode.
 *
 *         This function does not return.
 *
 *         Note: arguments are passed in the right order so that
 *         IRET may be called on them.
 *         */
void iret_travel( uint32_t eip, uint32_t cs,
        uint32_t eflags, uint32_t esp, uint32_t ss );

#endif /* _IRET_TRAVEL_H */
```

```
/** @file keybd_driver.h
 *  @brief Contains functions that can be called by interrupt handler assembly
 *         wrappers
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bugs No known bugs.
 */

# ifndef _P1_KEYBD_DRIVER_H_
# define _P1_KEYBD_DRIVER_H_

#include <stdint.h>

typedef int aug_char;
typedef uint8_t raw_byte;

void init_keybd(void);
void keybd_int_handler(void);
int get_next_aug_char( aug_char *next_char );
//int old_readline(char *buf, int len);

/** @brief unbounded circular array heavily inspired by 15-122 unbounded array
 *
 *  first is the index of the earliest unread element in the buffer. Once
 *  all functions that will ever need to read a buffer element has read
 *  the element at index 'first', first++ modulo limit.
 *
 *  last is one index after the latest element added to the buffer. Whenever
 *  an element is added to the buffer, last++ modulo limit.
 */
typedef struct {
  uint32_t size; /* 0 <= size && size < limit */
  uint32_t limit; /* 0 < limit */
  uint32_t first; /* if first <= last, then size == last - first */
  uint32_t last; /* else last < first, then size == limit - first + last */
  raw_byte *data;
} uba;

int is_uba(uba *arr);
uba *uba_new(int limit);
uba *uba_resize(uba *arr);
void uba_add(uba *arr, uint8_t elem);
uint8_t uba_rem(uba *arr);
int uba_empty(uba *arr);

#endif
```

```
#ifndef LIFE_CYCLE_H_
#define LIFE_CYCLE_H_

int _fork( void );
int _exec( void );

#endif /* LIFE_CYCLE_H_ */
```

```
/* The 15-410 kernel project
 *
 *     loader.h
 *
 * Structure definitions, #defines, and function prototypes
 * for the user process loader.
 */

#ifndef _LOADER_H
#define _LOADER_H

#include <elf_410.h>    /* simple_elf_t */

/* --- Prototypes --- */

int getbytes( const char *filename, int offset, int size, char *buf );

int execute_user_program( char *fname, int argc, char **argv);

#endif /* _LOADER_H */
```

```
/** @brief Header file with VM API. */

#ifndef _MEMORY_MANAGER_H
#define _MEMORY_MANAGER_H

#include <elf_410.h> /* simple_elf_t */
#include <stdint.h> /* uint32_t */

#define PAGE_DIRECTORY_INDEX 0xFFC00000
#define PAGE_TABLE_INDEX 0x003FF000

#define PAGE_DIRECTORY_SHIFT 22
#define PAGE_TABLE_SHIFT 12

/* Get page directory index from logical address */
#define PD_INDEX(addr) \
    ((PAGE_DIRECTORY_INDEX & ((uint32_t)(addr))) >> PAGE_DIRECTORY_SHIFT)

/* Get page table index from logical address */
#define PT_INDEX(addr) \
    ((PAGE_TABLE_INDEX & ((uint32_t)(addr))) >> PAGE_TABLE_SHIFT)

/* True if address if paged align, false otherwise */
#define PAGE_ALIGNED(address) ((((uint32_t) address) & (PAGE_SIZE - 1)) == 0)
#define USER_STR_LEN 256
#define NUM_USER_ARGS 16
#define TABLE_ADDRESS(PD_ENTRY) (((uint32_t) (PD_ENTRY)) & ~(PAGE_SIZE - 1))

#ifndef STACK_ALIGNED
#define STACK_ALIGNED(address) (((uint32_t) (address)) % 4 == 0)
#endif


void initialize_zero_frame( void );

/** Whether page is read only or also writable. */
typedef enum write_mode write_mode_t;
enum write_mode { READ_ONLY, READ_WRITE };

void *new_pd_from_elf( simple_elf_t *elf,
        uint32_t stack_lo, uint32_t stack_len );
void *new_pd_from_parent( void *parent_pd );
void vm_enable_task( void *ptd );
void enable_write_protection( void );
void disable_write_protection( void );
int vm_new_pages ( void *ptd, void *base, int len );

int is_valid_pt( uint32_t *pt, int pd_index );
int is_valid_pd( void *pd );

int is_user_pointer_allocated( void *ptr );

int is_valid_user_pointer( void *ptr, write_mode_t write_mode );
int is_valid_user_string( char *s, int max_len );
int is_valid_null_terminated_user_string( char *s, int max_len );
int is_valid_user_argvec( char *execname, char **argvec );
void free_pd_memory( void *pd );

int allocate_user_zero_frame( uint32_t **pd, uint32_t virtual_address,
                              uint32_t sys_prog_flag );
void unallocate_user_zero_frame( uint32_t **pd, uint32_t virtual_address);

void *get_pd( void );
int zero_page_pf_handler( uint32_t faulting_address );

#endif /* _MEMORY_MANAGER_H */
```

```
/** @file physalloc.h
 *  @brief Contains the interface for allocating and freeing physical frames.
 *
 *  Note that physical frame addresses are always represented as uint32_t so
 *  we never have to worry about accidentally dereferencing them when paging
 *  is turned on.
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */

#ifndef _PHYSALLOC_H_
#define _PHYSALLOC_H_

#include <stdint.h> /* uint32_t */

/* Function prototypes */
int is_physframe( uint32_t phys_address );
uint32_t physalloc( void );
void physfree( uint32_t phys_address );
uint32_t num_free_phys_frames( void );

/* Test functions */
void test_physalloc( void );

#endif /* _PHYSALLOC_H_ */
```

```
/** @file task_manager.h
 *  @brief Facilities for task creation and management.
 *
 *  @author Andre Nascimento (anascime)
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef TASK_MANAGER_H_
#define TASK_MANAGER_H_

#include <stdint.h>    /* uint32_t */
#include <elf_410.h>   /* simple_elf_t */

typedef enum status status_t;

/* PCB and TCB data structures */
typedef struct pcb pcb_t;
typedef struct tcb tcb_t;



/* Functions for task and thread creation */
void task_manager_init( void );
int create_pcb( uint32_t *pid, void *pd, pcb_t *parent_pcb );
int create_tcb( uint32_t pid , uint32_t *tid );
int create_task( uint32_t *pid, uint32_t *tid, simple_elf_t *elf );
int activate_task_memory( uint32_t pid );
void task_set_active( uint32_t tid );
void task_start( uint32_t tid, uint32_t esp, uint32_t entry_point );

/* Utility functions for getting and setting task and thread information */
tcb_t *find_tcb( uint32_t tid );
pcb_t *find_pcb( uint32_t pid );
uint32_t get_pid( void );
status_t get_tcb_status( tcb_t *tcb );
uint32_t get_tcb_tid(tcb_t *tcb);
void set_task_exit_status( int status );


int get_num_threads_in_owning_task( tcb_t *tcbp );
void *get_kern_stack_lo( tcb_t *tcbp );
void *get_kern_stack_hi( tcb_t *tcbp );
void set_kern_esp( tcb_t *tcbp, uint32_t *kernel_esp );
void *swap_task_pd( void *new_pd );
void *get_tcb_pd(tcb_t *tcb);
void free_tcb(tcb_t *tcb);

#endif /* TASK_MANAGER_H_ */
```

```
/** @file timer_driver.h
 *  @brief Contains internal functions that implement the timer driver.
 *
 *  This header is to be included by install_handler.c
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */

#ifndef _P1_TIMER_DRIVER_H_
#define _P1_TIMER_DRIVER_H_

void init_timer( void (*tickback)(unsigned int) );
unsigned int get_total_ticks( void );

#endif /* _P1_TIMER_DRIVER_H_ */
```

```c
/** @file install_handler.c
 *  @brief Contains functions that install the timer and keyboard handlers
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */

#include <install_handler.h>
#include <asm.h>             /* idt_base() */
#include <idt.h>             /* IDT_PF */
#include <seg.h>             /* SEGSEL_KERNEL_CS */
#include <assert.h>          /* assert() */
#include <stddef.h>          /* NULL */
#include <keyhelp.h>         // FIXME: ????
#include <timer_driver.h>    /* init_timer() */
#include <keybd_driver.h>    /* init_keybd() */
#include <timer_defines.h>   /* TIMER_IDT_ENTRY */
#include <lib_console/readline.h> /* init_readline() */
#include <interrupt_defines.h>
#include <asm_interrupt_handler.h>          /* call_timer_int_handler(),
                                               call_keybd_int_handler() */
#include <asm_misc_handlers.h>
#include <asm_fault_handlers.h>
#include <asm_console_handlers.h>
#include <asm_life_cycle_handlers.h>
#include <asm_thread_management_handlers.h>
#include <asm_memory_management_handlers.h>
#include <tests.h>                          /* install_test_handler() */

#include <syscall_int.h> /* *_INT */


#define TEST_INT SYSCALL_RESERVED_0

/**********************************************************************/
/*                                                                    */
/* Internal helper functions                                          */
/*                                                                    */
/**********************************************************************/

/** @brief Installs an interrupt handler at idt_entry
 *
 *  If a tickback function pointer is provided, init_timer() will be called.
 *  Else it is assumed that init_keybd() should be called instead, since there
 *  are only 2 possible idt_entry values to accept.
 *
 *  Furthermore, when casting from pointer types to integer types to pack
 *  bits into the trap gate data structure, the pointer is first cast to
 *  unsigned long then unsigned int, therefore there is no room for undefined
 *  behavior.
 *
 *  @param idt_entry Index into IDT table.
 *  @param asm_wrapper Assembly wrapper to call C interrupt handler
 *  @param tickback Application provided callback function for timer interrupts.
 *  @return 0 on success, -1 on error.
 */
int
install_handler_in_idt(int idt_entry, asm_wrapper_t *asm_wrapper, int dpl,
                       int gate_type)
{
  if (!asm_wrapper) {
      return -1;
  }
  /* Get gate address */
  void *idt_base_addr = idt_base();
  void *idt_entry_addr = idt_base_addr + (idt_entry * BYTES_PER_GATE);

  /* Exact offsets of each 32-bit word in the trap gate */
  unsigned int *idt_entry_addr_lower = idt_entry_addr;
  unsigned int *idt_entry_addr_upper = idt_entry_addr + (BYTES_PER_GATE / 2);
  if (idt_entry_addr_lower == NULL || idt_entry_addr_upper == NULL) return -1;

  /* Construct data for upper 32-bit word with necessary flags */
  unsigned int data_upper = 0;
  unsigned int offset_upper =
    ((unsigned int) (unsigned long) asm_wrapper) & OFFSET_UPPER_MASK;
  data_upper = offset_upper | PRESENT | dpl | gate_type;

  /* Zero all bits that are not reserved, pack data into upper 32-bit word */
  *idt_entry_addr_upper = *idt_entry_addr_upper & RESERVED_UPPER_MASK;
  *idt_entry_addr_upper = *idt_entry_addr_upper | data_upper;

  /* Construct data for lower 32-bit word with necessary flags */
  unsigned int data_lower = 0;
  unsigned int offset_lower =
    ((unsigned int) (unsigned long) asm_wrapper) & OFFSET_LOWER_MASK;
  data_lower = (SEGSEL_KERNEL_CS << (2 * BYTE_LEN)) | offset_lower;

  /* Pack data into lower 32-bit word */
  *idt_entry_addr_lower = data_lower;

  /* This should always be the case after writing to IDT */
  assert(*idt_entry_addr_lower == data_lower);
  assert(*idt_entry_addr_upper == data_upper);

  return 0;
}

/** @brief Install timer interrupt handler
 */
int
install_timer_handler(int idt_entry, asm_wrapper_t *asm_wrapper,
                      void (*tickback)(unsigned int))
{
    if (!asm_wrapper) {
        return -1;
    }
    init_timer(tickback);
    return install_handler_in_idt(idt_entry, asm_wrapper, DPL_0, D32_TRAP);
}

/** @brief General function to install a handler without an init function
 *
 *  @param idt_entry Index in IDT to install
 *  @param init Initialization function if needed for handler installation
 *  @param asm_wrapper Assembly wrapper to call the handler
 *  @param dpl DPL
 *  @param gate_type Wether it is a trap gate or interrupt gate
 *  @return 0 on success, -1 on error
 */
int
install_handler( int idt_entry, init_func_t *init, asm_wrapper_t *asm_wrapper,
                 int dpl, int gate_type )
{
    if (!asm_wrapper) {
        return -1;
    }
    if ((gate_type != D32_TRAP) && (gate_type != D32_INTERRUPT)) {
        return -1;
    }
    if (init) {
        init();
    }
    return install_handler_in_idt(idt_entry, asm_wrapper, dpl, gate_type);
```

```
}

/** @brief Install keyboard interrupt handler
 */
int
install_keyboard_handler(int idt_entry, asm_wrapper_t *asm_wrapper)
{
    if (!asm_wrapper) {
        return -1;
    }
    init_keybd();

    /* The keyboard handler is made an interrupt gate. This is because if we
     * make it a trap gate instead, the timer interrupt could trigger a context
     * switch before we acknowledge the keybd interrupt signal, thereby
     * invalidating all user input until we context switch back to the thread
     * handling the keyboard interrupt. */
    return install_handler_in_idt(idt_entry, asm_wrapper, DPL_0, D32_INTERRUPT);
}

/*********************************************************************/
/*                                                                   */
/* Interface for device-driver initialization and timer callback     */
/*                                                                   */
/*********************************************************************/

/** @brief Installs all interrupt handlers by calling the correct function
 *
 *   After installing both the timer and keyboard handler successfully,
 *   interrupts are enabled.
 *
 *   Requires that interrupts are disabled.
 *
 *   @param tickback Pointer to clock-tick callback function
 *   @return A negative error code on error, or 0 on success
 **/
int
handler_install(void (*tick)(unsigned int))
{
    /* Install test syscall for tests spanning user and kernel mode. */
    if (install_test_handler(TEST_INT, call_test_int_handler) < 0) {
        return -1;
    }
    if (install_timer_handler(TIMER_IDT_ENTRY, call_timer_int_handler,
                              tick) < 0) {
        return -1;
    }
    if (install_keyboard_handler(KEY_IDT_ENTRY, call_keybd_int_handler) < 0) {
        return -1;
    }

    /* Lib thread management */
    if (install_handler(GETTID_INT, NULL, call_gettid, DPL_3, D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(GET_TICKS_INT, NULL, call_get_ticks, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(YIELD_INT, NULL, call_yield, DPL_3, D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(DESCHEDULE_INT, NULL, call_deschedule, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(MAKE_RUNNABLE_INT, NULL, call_make_runnable, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(SLEEP_INT, NULL, call_sleep, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    /* Lib lifecycle*/
    //TODO are these DPL_3 or DPL_0
    if (install_handler(FORK_INT, NULL, call_fork, DPL_3, D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(EXEC_INT, NULL, call_exec, DPL_3, D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(VANISH_INT, NULL, call_vanish, DPL_3, D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(TASK_VANISH_INT, NULL, call_task_vanish, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(SET_STATUS_INT, NULL, call_set_status, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }



    /* Lib memory management */
    if (install_handler(NEW_PAGES_INT, NULL, call_new_pages, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(REMOVE_PAGES_INT, NULL, call_remove_pages, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(IDT_PF, NULL, call_pagefault_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    /* Lib console */
    if (install_handler(READLINE_INT, init_readline,
        call_readline, DPL_3, D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(PRINT_INT, NULL, call_print, DPL_3, D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(GET_CURSOR_POS_INT, NULL, call_get_cursor_pos, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(SET_CURSOR_POS_INT, NULL, call_set_cursor_pos, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    if (install_handler(SET_TERM_COLOR_INT, NULL, call_set_term_color_handler,
        DPL_3, D32_TRAP) < 0) {
```

```
        return -1;
    }

    /* Lib misc */
    if (install_handler(READFILE_INT, NULL, call_readfile, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(HALT_INT, NULL, call_halt, DPL_3, D32_TRAP) < 0) {
        return -1;
    }

    /* Fault handlers */
    if (install_handler(IDT_DE, NULL, call_divide_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_DB, NULL, call_debug_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_BP, NULL, call_breakpoint_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_OF, NULL, call_overflow_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_BR, NULL, call_bound_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_UD, NULL, call_invalid_opcode_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_NM, NULL, call_float_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_NP, NULL, call_segment_not_present_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_SS, NULL, call_stack_fault_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_GP, NULL, call_general_protection_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_AC, NULL, call_alignment_check_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_NMI, NULL, call_non_maskable_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }

    if (install_handler(IDT_MC, NULL, call_machine_check_handler, DPL_3,
        D32_TRAP) < 0) {
        return -1;
    }
    return 0;
}
```

```
#include <seg.h> /* SEGSEL_USER_DS */

.globl iret_travel

# iret_travel ( eip, cs, eflags, esp, ss )
iret_travel:
    movl $SEGSEL_USER_DS, %ax
    movl %ax, %ds   # Update ds to user data segment
    movl %ax, %es
    movl %ax, %fs
    movl %ax, %gs
    addl $4, %esp   # Point esp to last argument
    iret            # Consume all arguments and go to user mode!
```

```
/** @file kernel.c
 *  @brief An initial kernel.c
 *
 *  You should initialize things in kernel_main(),
 *  and then run stuff.
 *
 *  @author Harry Q. Bovik (hqbovik)
 *  @author Fred Hacker (fhacker)
 *  @bug No known bugs.
 */

#include <logger.h> /* log(), log_med(), log_hi() */
#include <install_handler.h> /* handler_install() */
#include <common_kern.h>

/* libc includes. */
#include <stdio.h>

/* multiboot header file */
#include <multiboot.h>              /* boot_info */

/* x86 specific includes */
#include <cr.h>     /* get/set_cr0() */
#include <asm.h>    /* enable_interrupts() */

#include <exec2obj.h> /* MAX_EXECNAME_LEN */

#include <logger.h>         /* log_info() */
#include <limits.h>         /* UINT_MAX */
#include <loader.h>         /* execute_user_program() */
#include <console.h>        /* init_console() */
#include <scheduler.h>      /* scheduler_on_tick() */
#include <task_manager.h>   /* task_manager_init() */
#include <memory_manager.h> /* initialize_zero_frame() */
#include <keybd_driver.h>   /* readline() */
#include <lib_thread_management/sleep.h>    /* sleep_on_tick() */
#include <simics.h>

volatile static int __kernel_all_done = 0;

#define PROTECTION_ENABLE_FLAG (1 << 0)

/* Level of logging, set to 4 to turn logging off,
 * 1 to print logs for log priorities lo, med, hi
 * 2 to print logs for log priorities med, hi
 * 3 to print logs for log priorities hi
 *
 * defining the NDEBUG flag will also turn logging off
 */
int log_level = 3;

void tick(unsigned int numTicks) {
    /* At our tickrate of 1000Hz, after around 48 days numTicks will overflow
     * and break a lot of things. Let the user know they should be more polite
     * and restart their computer every other month! */
    if (numTicks == UINT_MAX) {
        panic("System has been running for too long. Please reboot every "
                "other month!");
    }

    /* The amount of work done before scheduler tick handler should be
     * small, as it will consume time from the thread being context switched
     * to (in most cases). */
    sleep_on_tick(numTicks);

    //if (get_running_thread()) {
    //  assert(*((uint32_t *) get_kern_stack_hi((get_running_thread())))
    //  == 0xcafebabe);
    //  assert(*((uint32_t *) get_kern_stack_lo((get_running_thread())))
    //  == 0xdeadbeef);

    //}
    if (get_running_thread()) {
        if (*((uint32_t *) get_kern_stack_hi((get_running_thread())))
        != 0xcafebabe) {
            MAGIC_BREAK;
        }

        if (*((uint32_t *) get_kern_stack_lo((get_running_thread())))
        != 0xdeadbeef) {
            MAGIC_BREAK;
        }
    }

    /* Scheduler tick handler should be last, as it triggers context_switch */
    scheduler_on_tick(numTicks);
    if (get_running_thread()) {
        if (*((uint32_t *) get_kern_stack_hi((get_running_thread())))
        != 0xcafebabe) {
            MAGIC_BREAK;
        }

        if (*((uint32_t *) get_kern_stack_lo((get_running_thread())))
        != 0xdeadbeef) {
            MAGIC_BREAK;
        }
    }



}

void hard_code_test( char *s )
{
    char *argv[] = {s, 0};
    execute_user_program(s, 1, argv);
}

/** @brief Kernel entrypoint.
 *
 *  This is the entrypoint for the kernel.
 *
 * @return Does not return
 */
int
kernel_main( mbinfo_t *mbinfo, int argc, char **argv, char **envp )
{
    /* FIXME: What to do with mbinfo and envp? */
    (void)mbinfo;
    (void)envp;

    /* initialize handlers and enable interrupts */
    if (handler_install(tick) < 0) {
        panic("cannot install handlers");
    }
    enable_interrupts();

    init_console();

    task_manager_init();

    // TODO: maybe should be somewhere else
    initialize_zero_frame();
```

```
    log("this is DEBUG");
    log_info("this is INFO");
    log_warn("this is WARN");

    char *args[] = {"init", 0};
    execute_user_program("init", 1, args);
    return 0;
}
```

```
/** @file keybd_driver.c
 *  @brief Contains functions that help the user type into the console
 *
 *  @bug No known bugs.
 *
 *
 *  Since unbounded arrays are used, let 'size' be the number of elements in the
 *  array that we care about, and 'limit' be the actual length of the array.
 *  Whenever the size of the array == its limit, the array limit is doubled.
 *  Doubling is only allowed if the current limit <= UINT32_MAX to prevent
 *  overflow.
 *
 *  Indexing into circular arrays is just done modulo the limit of the array.
 *
 *  The two functions in the keyboard driver interface readchar() amd readline()
 *  are closely connected to one another. The specification for readline()
 *  states that "Characters not placed into the specified buffer should remain
 *  available for other calls to readline() and/or readchar()." To put it
 *  concisely, we have the implication:
 *
 *    char not committed to any buffer => char available for other calls
 *
 *    i.e.
 *
 *    char not available for other calls => char committed to some buffer
 *
 *  The question now is under what circumstance do we promise that a char
 *  is not available? It is reasonable to conclude that the above implication
 *  is in fact a bi-implication. Therefore:
 *
 *    char not available for other calls <=> char committed to some buffer
 *
 *  Now since readchar() always reads the next character in the keyboard buffer,
 *  it is then conceivable that if a readchar() not called by readline() takes
 *  the next character off the keyboard buffer, then readline() would "skip"
 *  a character. But then, the specification says that:
 *
 *  "Since we are operating in a single-threaded environment, only one of
 *  readline() or readchar() can be executing at any given point."
 *
 *  Therefore we will never have readline() and readchar() concurrently
 *  executing in seperate threads in the context of the same process, since
 *  when we speak of threads, we refer to threads in the same process.
 *
 *  Therefore we need not worry about the case where a readchar() that is
 *  not invoked by readline() is called in the middle of another call to
 *  readline().
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs
 */

#include <keybd_driver.h>

#include <asm.h>                    /* process_scancode() */
#include <ctype.h>                  /* isprint() */
#include <malloc.h>                 /* calloc */
#include <stddef.h>                 /* NULL */
#include <assert.h>                 /* assert() */
#include <string.h>                 /* memcpy() */
#include <console.h>                /* putbyte() */
#include <keyhelp.h>                /* process_scancode() */
#include <video_defines.h>          /* CONSOLE_HEIGHT, CONSOLE_WIDTH */
#include <variable_buffer.h>        /* generic buffer macros */
#include <interrupt_defines.h>      /* INT_CTL_PORT */
#include <lib_console/readline.h>   /* readline_char_arrived_handler() */

/* Keyboard buffer */
new_buf(keyboard_buffer_t, uint8_t, CONSOLE_WIDTH * CONSOLE_HEIGHT);
static keyboard_buffer_t key_buf;

int readchar(void);

/** @brief Interrupt handler which reads in raw bytes from keystrokes. Reads
 *         incoming bytes to the keyboard buffer key_buf, which has an
 *         amortized constant time complexity for adding elements. So it
 *         returns quickly
 *
 *  @return Void.
 */
void keybd_int_handler(void)
{
    /* Read raw byte and put into raw character buffer */
    uint8_t raw_byte = inb(KEYBOARD_PORT);
    buf_insert(&key_buf, raw_byte);

    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    /* Let readline module know new characters have arrived. */
    readline_char_arrived_handler();
}

/** @brief Initialize the keyboard interrupt handler and associated data
 *         structures
 *
 *  Memory for keybd_buf is allocated here.
 *
 *  @return Void.
 */
void
init_keybd( void )
{
    init_buf(&key_buf, uint8_t, CONSOLE_WIDTH * CONSOLE_HEIGHT);
    is_buf(&key_buf);
    return;
}

/** @brief Keeps calling readchar() until another valid char is read and
 *         returns it.
 *
 *  @return A valid character when readchar() doesn't return -1.
 */
char get_next_char(void) {

    /* Get the next char value off the keyboard buffer */
    int res;
    while((res = readchar()) == -1) continue;
    assert(res >= 0);

    /* Tricky type conversions to avoid undefined behavior */
    char char_value = (uint8_t) (unsigned int) res;
    return char_value;
}

/*****************************************************************/
/*                                                               */
/* Keyboard driver interface                                     */
/*                                                               */
/*****************************************************************/

int
get_next_aug_char( aug_char *next_char )
{
```

```
    disable_interrupts();
    if (buf_empty(&key_buf)) {
        enable_interrupts();
        return -1;
    }
    uint8_t next_byte;
    //uint8_t *next_bytep = &next_byte;
    buf_remove(&key_buf, &next_byte);
    is_buf(&key_buf);
    enable_interrupts();

    *next_char = process_scancode(next_byte);
    return 0;
}

/** @brief Returns the next character in the keyboard buffer
 *
 *  This function does not block if there are no characters in the keyboard
 *  buffer
 *  --
 *  No other process will call readchar() concurrently with readline() since
 *  we only have 1 kernal process running and have a single thread.
 *  --
 *  @return The next character in the keyboard buffer, or -1 if the keyboard
 *          buffer is currently empty
 **/
int
readchar( void )
{
  /* uba invariants are checked whenever we access the data structure, and so
   * interrupts are disabled during the entire call in order to prevent
   * changes to the data structure at the start of and at the end of a call.
   */


    /* Get simplified character */
    aug_char next_char;
    while (get_next_aug_char(&next_char) == 0) {
        if (KH_HASDATA(next_char) && KH_ISMAKE(next_char)) {
            unsigned char next_char_value = KH_GETCHAR(next_char);
            return (int) (unsigned int) next_char_value;
        }
    }
    return -1;
}

/** @brief Reads a line of characters into a specified buffer
 *
 * If the keyboard buffer does not already contain a line of input,
 * readline() will spin until a line of input becomes available.
 *
 * If the line is smaller than the buffer, then the complete line,
 * including the newline character, is copied into the buffer.
 *
 * If the length of the line exceeds the length of the buffer, only
 * len characters should be copied into buf.
 *
 * Available characters should not be committed into buf until
 * there is a newline character available, so the user has a
 * chance to backspace over typing mistakes.
 *
 * While a readline() call is active, the user should receive
 * ongoing visual feedback in response to typing, so that it
 * is clear to the user what text line will be returned by
 * readline().
 * --
 * the definition of a line in readline() is different from a row. A
 * carriage-return will return the cursor to its initial position at the
 * start of the call to readline(). Backspaces will always work and only
 * do nothing if the cursor is at the initial position at the start of the
 * call to readline.
 *
 * Since it is only meaningful that the user can see exactly what was written
 * to buf, If len is valid the moment the user types len bytes readline() will
 * return. We prevent the user from typing more than len characters into the
 * console.
 * --
 *  @param buf Starting address of buffer to fill with a text line
 *  @param len Length of the buffer
 *  @return The number of characters in the line buffer,
 *          or -1 if len is invalid or unreasonably large.
 */
int old_readline(char *buf, int len) {

  /* buf == NULL so invalid buf */
  if (buf == NULL) return -1;

  /* len < 0 so invalid len */
  if (len < 0) return -1;

  /* len == 0 so no need to copy */
  if (len == 0) return 0;

  /* len too large */
  if (len > CONSOLE_WIDTH * CONSOLE_HEIGHT) return -1;

  /* get original cursor position for start of line relative to scroll */
  int start_row, start_col;
  get_cursor(&start_row, &start_col);

  /* Allocate space on stack for temporary buffer */
  char temp_buf[len];

  /* Initialize index into temp_buf */
  int i = 0;
  int written = 0; /* characters written so far */
  char ch;

  while ((ch = get_next_char()) != '\n' && written < len) {

    /* ch, i, written is always in range */
    assert(0 <= i && i < len);
    assert(0 <= written && written < len);

    /* If at front of buffer, Delete the character if backspace */
    if (ch == '\b') {

      /* If at start_row, start_col, do nothing as don't delete prompt */
      int row, col;
      get_cursor(&row, &col);
      assert (row * CONSOLE_WIDTH + col >= start_row * CONSOLE_WIDTH + start_col);
      if (!(row == start_row && col == start_col)) {
        assert(i > 0);

        /* Print to screen and update intial cursor position if needed*/
        scrolled_putbyte(ch, &start_row, &start_col);

        /* update i and buffer */
        i--;
        temp_buf[i] = ' ';
      }
    /* '\r' sets cursor to position at start of call. Don't overwrite prompt */
    } else if (ch == '\r') {
```

```
      /* Set cursor to start of line w.r.t start of call, i to buffer start */
      set_cursor(start_row, start_col);
      i = 0;

    /* Regular characters just write, unprintables do nothing */
    } else {

      /* print on screen and update initial cursor position if needed */
      scrolled_putbyte(ch, &start_row, &start_col);

      /* write to buffer */
      if (isprint(ch)) {
        temp_buf[i] = ch;
        i++;
        if (i > written) written = i;
      }
    }
  }
  assert(written <= len);
  if (ch == '\n') {

    /* Only write the newline if there's space for it in the buffer */
    if (written < len) {
      putbyte(ch);
      temp_buf[i] = '\n';
      i++;
      if (i > written) written = i;
    }
  } else {
    assert(written == len);
  }
  memcpy(buf, temp_buf, written);
  return written;
}
```

```
#include <asm_interrupt_handler_template.h>

CALL_W_DOUBLE_ARG(print)
CALL_W_DOUBLE_ARG(readline)
CALL_W_DOUBLE_ARG(get_cursor_pos)
CALL_W_DOUBLE_ARG(set_cursor_pos)
CALL_W_SINGLE_ARG(set_term_color_handler)
```

```
/** @file get_cursor_pos.c
 *  @brief Get cursor position syscall handler
 */
#include <asm.h>                /* outb */
#include <console.h>            /* get_cursor */
#include <memory_manager.h>     /* is_user_pointer_valid */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

/** @brief Handler for get_cursor_pos syscall. */
int
get_cursor_pos( int *row, int *col )
{
    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    if (!is_valid_user_pointer(row, READ_WRITE)
            || !is_valid_user_pointer(col, READ_WRITE))
        return -1;

    get_cursor(row, col);
    return 0;
}
```

```
/** @file print.c
 *  @brief Print syscall handler and facilities for managing
 *         concurrent prints
 */
#include <asm.h>                /* outb */
#include <assert.h>             /* affirm */
#include <console.h>            /* putbytes */
#include <memory_manager.h>     /* is_valid_user_pointer */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <lib_thread_management/mutex.h>    /* mutex_t */

static mutex_t print_mux;

/* 0 if uninitialized, 1 if initialized */
static int print_initialized = 0;

static void
init_print( void )
{
    affirm(!print_initialized);
    mutex_init(&print_mux);
    print_initialized = 1;
}

/** @brief Handler for print syscall. */
int
print( int len, char *buf )
{
    /* Check init before acknowledging interrupt to ensure
     * no race conditions among concurrent init_print() calls */
    if (!print_initialized)
        init_print();

    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    if (!is_valid_user_string(buf, len))
        return -1;

    mutex_lock(&print_mux);

    // Check here that print_mux has actually been locked, maybe print on mux_lock
or smth

    putbytes(buf, len);

    mutex_unlock(&print_mux);

    return 0;
}
```

```
/** @file  readline.c
 *  @brief Readline syscall handler and facilities for managing
 *         concurrent readline's
 */
#include <asm.h>            /* outb */
#include <ctype.h>          /* isprint() */
#include <assert.h>         /* affirm */
#include <string.h>         /* memcpy */
#include <stdint.h>         /* uint32_t */
#include <stddef.h>         /* NULL */
#include <console.h>        /* putbyte() */
#include <keyhelp.h>        /* process_scancode() */
#include <scheduler.h>      /* status_t, queue_t, make_runnable  */
#include <keybd_driver.h>   /* aug_char*/
#include <atomic_utils.h>   /* compare_and_swap_atomic */
#include <keybd_driver.h>   /* get_next_aug_char */
#include <task_manager.h>   /* tcb_t */
#include <video_defines.h>  /* CONSOLE_HEIGHT, CONSOLE_WIDTH */
#include <variable_queue.h> /* Q_ macros */
#include <memory_manager.h> /* READ_WRITE, is_valid_user_pointer */
#include <task_manager_internal.h> /* struct tcb */
#include <lib_thread_management/mutex.h> /* mutex_t */

// TODO: Delete, debugging
#include <simics.h>
#include <malloc.h>

static void mark_curr_blocked( tcb_t *tcb, void *data );
static int readchar( void );
static char get_next_char( void );
static int _readline(char *buf, int len);

/** @brief Queue of threads blocked on readline call. */
//static queue_t    readline_q;

/** @brief Thread being served. Can be blocked, running or runnable. */
static tcb_t   *readline_curr;

/** @brief Whether thread being served is blocked.
 *
 *  Readline_curr may blind write to this, keybd int handler
 *  should atomically compare-and-swap to avoid race conditions. */
static uint32_t curr_blocked;

/** @brief Mutex for readline_curr and readline_q. */
static mutex_t  readline_mux;


/** @brief Initialize readline */
void
init_readline( void )
{
    mutex_init(&readline_mux);
    readline_curr = NULL;
    curr_blocked = 0;
}


/** @brief Readline syscall handler
 *
 *  @param buf Buffer in which to write characters
 *  @param len Max number of characters to write in buf */
int
readline( int len, char *buf )
{
    if (len < 0) return -1;
    if (len == 0) return 0;
    if (len > CONSOLE_WIDTH * CONSOLE_HEIGHT) return -1;
    for (int i=0; i < len; ++i) {
        if (!is_valid_user_pointer(buf + i, READ_WRITE))
            return -1;
    }

    lprintf("Passed validity checks");

    /* Acquire readline mux. Put ourselves at the back of the queue. */
    mutex_lock(&readline_mux);

    //TODO does not seem to be used
    readline_curr = get_running_thread();

    int res = _readline(buf, len);

    /* Done. Check if anyone else waiting in line */
    mutex_unlock(&readline_mux);

    return res;
}


static int
_readline(char *buf, int len)
{
    int start_row, start_col;
    get_cursor(&start_row, &start_col);

    assert(len <= CONSOLE_WIDTH * CONSOLE_HEIGHT);
    char *temp_buf;
    temp_buf = smalloc(CONSOLE_WIDTH * CONSOLE_HEIGHT);
    //char temp_buf[CONSOLE_WIDTH * CONSOLE_HEIGHT];

    int i = 0;
    int written = 0; /* characters written so far */
    char ch;

    while ((ch = get_next_char()) != '\n' && written < len) {

        /* ch, i, written is always in range */
        assert(0 <= i && i < len);
        assert(0 <= written && written < len);

        /* If at front of buffer, Delete the character if backspace */
        if (ch == '\b') {

            /* If at start_row/col, do nothing so as to not delete prompt */
            int row, col;
            get_cursor(&row, &col);

            if (!(row * CONSOLE_WIDTH + col >= start_row * CONSOLE_WIDTH + start_co
l)) {
                continue;
            }

            if (i > 0) {
                assert((row != start_row || col != start_col));

                /* Print to screen and update intial cursor position if needed*/
                scrolled_putbyte(ch, &start_row, &start_col);

                /* update i and buffer */
                i--;
                temp_buf[i] = ' ';
            }
        /* '\r' sets cursor to position at start of call. Don't overwrite prompt */
```

```
        } else if (ch == '\r') {

              /* Set cursor to start of line w.r.t start of call, i to buffer start
 */
              set_cursor(start_row, start_col);
              i = 0;

        /* Regular characters just write, unprintables do nothing */
        } else {

            /* print on screen and update initial cursor position if needed */
            scrolled_putbyte(ch, &start_row, &start_col);

            /* write to buffer */
            temp_buf[i] = ch;
            i++;
            if (i > written) written = i;
        }
    }
    assert(written <= len);
    if (ch == '\n') {
        /* Only write the newline if there's space for it in the buffer */
        if (written < len) {
            putbyte(ch);
            temp_buf[i] = '\n';
            i++;
            if (i > written) written = i;
        }
    } else {
        assert(written == len);
    }
    memcpy(buf, temp_buf, written);

    // TODO: Delte
    sfree(temp_buf, CONSOLE_HEIGHT * CONSOLE_WIDTH);

    return written;
}

/** @brief Call to let readline know new characters have arrived.
 *  This is called within the keybd interrupt handler, after the
 *  signal is acknowledged. */
void
readline_char_arrived_handler( void )
{
    /* If curr_blocked, readline_char_arrived_handler is the only one who can
     * operate on curr_blocked and readline_curr. However, we can get another
     * keybd interrupt, leading to a race condition. A simple way to ensure
     * make_runnable is called only once is a CAS on curr_blocked. */
    if (compare_and_swap_atomic(&curr_blocked, 1, 0)) {
        switch_safe_make_thread_runnable(readline_curr->tid);
    }
}

/* --- HELPERS --- */

static void
mark_curr_blocked( tcb_t *tcb, void *data )
{
    assert(readline_curr == tcb);
    curr_blocked = 1;
}

static int
readchar( void )
{
    aug_char next_char;
    while (get_next_aug_char(&next_char) == 0) {
        if (KH_HASDATA(next_char) && KH_ISMAKE(next_char)) {
            unsigned char next_char_value = KH_GETCHAR(next_char);
            return (int) (unsigned int) next_char_value;
        }
    }
    return -1;
}

static char
get_next_char( void )
{
    /* Get the next char value off the keyboard buffer */
    int res;
    /* If no character, deschedule ourselves and wait for user input. */
    while ((res = readchar()) == -1) {
        yield_execution(BLOCKED, -1, mark_curr_blocked, NULL);
    }
    assert(res >= 0);

    /* Tricky type conversions to avoid undefined behavior */
    char char_value = (uint8_t) (unsigned int) res;
    return char_value;
}
```

```
/** @file Readline definitions for others to use
 *
 *  @brief Exports read_char_arrived handler, essential
 *  for not wasting cycles checking if new characters
 *  arrived inside the readline syscall.
 *  */

#ifndef READLINE_H_
#define READLINE_H_

void init_readline( void );
void readline_char_arrived_handler( void );

#endif /* READLINE_H_ */
```

```
/** @file set_cursor_pos.c
 *  @brief Set cursor position syscall handler
 */
#include <asm.h>                /* outb */
#include <console.h>            /* set_cursor */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

/** @brief Handler for set_cursor_pos syscall. */
int
set_cursor_pos( int row, int col )
{
    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return set_cursor(row, col);
}
```

```
/** @file set_term_color.c
 *  @brief Set terminal color syscall handler
 */
#include <asm.h>                /* outb */
#include <console.h>            /* set_term_color */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

/** @brief Handler for set_term_color syscall.
 *
 *  NOTE: appended _handler to distinguish from console.c function */
int
set_term_color_handler( int color )
{
    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return set_term_color(color);
}
```

```
#include <asm_interrupt_handler_template.h>

/* should be call w void val? */
CALL_W_RETVAL_HANDLER(vanish)
CALL_W_SINGLE_ARG(task_vanish)
CALL_W_SINGLE_ARG(set_status)

CALL_W_RETVAL_HANDLER(fork)
CALL_W_DOUBLE_ARG(exec)
```

```
/** @file exec.c
 *  @brief Contains exec interrupt handler and helper functions
 *
 *  @author Nicklaus Choo (nchoo)
 */

#include <string.h>    /* strlen(), memcpy(), memset() */
#include <loader.h>    /* exec_user_program() */
#include <logger.h>    /* log() */
#include <assert.h>    /* affirm() */
#include <x86/asm.h>   /* outb() */
#include <scheduler.h> /* get_running_tid() */
#include <task_manager.h>   /* get_num_threads_in_owning_task() */
#include <memory_manager.h> /* is_valid_user_string(), is_valid_user_argvec() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <simics.h>

/** @brief Prints arguments passed to exec() when log level is DEBUG
 *
 *  @param execname Executable name
 *  @param argvec Argument vector
 *  @return Void.
 */
static void
log_exec_args( char *execname, char **argvec )
{
    log("exec name is '%s'", execname);
    int i = 0;
    while (argvec[i]) {
        log("argvec[%d]:'%s'", i, argvec[i]);
        ++i;
    }
    log("argvec has %d elements", i);
}

/** @brief Executes execname with arguments in argvec
 *
 *  argvec can have a maximum of NUM_USER_ARGS, which is admittedly an arbitrary
 *  power of 2, but most executable invocations usually use less than 16
 *  parameters.
 *
 *  execname and each of the string arguments can have < USER_STR_LEN
 *  characters, which is a gain another reasonable limit for string length.
 *
 *  (NUM_USER_ARGS, USER_STR_LEN defined in memory_manager.h)
 *
 *  @param execname Executable name
 *  @param argvec String array of arguments for executable execname
 *  @return Does not return on success, -1 on error
 */
int
exec( char *execname, char **argvec )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);
//  assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    /* Only allow exec of task that has 1 thread */
    tcb_t *tcb = get_running_thread();
    assert(tcb);

    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    int num_threads = get_num_threads_in_owning_task(tcb);
    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    log("Exec() task with number of threads:%ld", num_threads);
    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    if (num_threads > 1) {
        return -1;
    }
    assert(num_threads == 1);
    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    /* Validate execname */
    if (!is_valid_null_terminated_user_string(execname, USER_STR_LEN)) {
        return -1;
    }
    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));
    /* Validate argvec */
    int argc = 0;
    if (!(argc = is_valid_user_argvec(execname, argvec))) {
        return -1;
    }
    // TODO ensure no software exception handler registered
    //assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    log_exec_args(execname, argvec);

    /* Execute */
    if (execute_user_program(execname, argc, argvec)
        < 0) {
        return -1;
    }
    panic("exec() should not come here!");
    return -1;
}
```

```
/** @file fork.c
 *  @brief Contains fork interrupt handler and helper functions for
 *         installation
 *
 *
 */

#include <logger.h> /* log() */
#include <assert.h>
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <x86/cr.h> /* get_cr3() */
#include <common_kern.h> /* USER_MEM_START */
#include <malloc.h> /* smemalign() */
#include <string.h> /* memcpy() */
#include <page.h> /* PAGE_SIZE */
#include <task_manager.h>
#include <memory_manager.h> /* new_pd_from_parent, PAGE_ALIGNED() */
#include <simics.h>
#include <scheduler.h>
#include <x86/asm.h> /* outb() */

// saves regs and returns new esp
void *save_child_regs(void *parent_kern_esp, void *child_kern_esp,
                      void *child_cr3 );

/** @brief Prints the parent and child stacks on call to fork()
 *
 *  @parent_tcb Parent's thread control block
 *  @child_tcb Child's thread control block
 *  @return Void.
 */
void
log_print_parent_and_child_stacks( tcb_t *parent_tcb, tcb_t *child_tcb )
{
    log("print parent stack");
    for (int i = 0; i < 32; ++i) {
        log("address:%p, value:0x%lx",
        (uint32_t *) get_kern_stack_hi(parent_tcb) - i,
        *((uint32_t *) get_kern_stack_hi(parent_tcb) - i));
    }
    log("print child stack");
    for (int i = 0; i < 32; ++i) {
        log("address:%p, value:0x%lx",
        (uint32_t *) get_kern_stack_hi(child_tcb) - i,
        *((uint32_t *) get_kern_stack_hi(child_tcb) - i));
    }
    log("result from get_running_tid():%d", get_running_tid());
}

/** @brief Fork task into two.
 *
 *  @return 0 for child thread, child tid for parent thread, -1 on error  */
int
fork( void )
{
    //assert(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));

    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);


    /* Only allow forking of task that has 1 thread */
    tcb_t *parent_tcb = get_running_thread();
    affirm(parent_tcb);
    pcb_t *parent_pcb = get_running_task();
    affirm(parent_pcb);


    int num_threads = get_num_threads_in_owning_task(parent_tcb);
    log_info("fork(): "
             "Forking task with number of threads:%ld", num_threads);

    if (num_threads > 1) {
        return -1;
    }
    //assert(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));

    assert(num_threads == 1);
    //affirm(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));

    /* Get parent_pd in kernel memory, unaffected by paging */
    uint32_t cr3 = get_cr3();
    //affirm(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));

    uint32_t *parent_pd = (uint32_t *) (cr3 & ~(PAGE_SIZE - 1));
    assert((uint32_t) parent_pd < USER_MEM_START);
    //affirm(is_valid_pd(parent_pd));


    /* Create child_pd as a deep copy */
    uint32_t *child_pd = new_pd_from_parent((void *)parent_pd);
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    //assert(PAGE_ALIGNED(child_pd));
    log_info("fork(): "
             "new child_pd at address:%p", child_pd);

    /* Create child pcb and tcb */
    uint32_t child_pid, child_tid;
    if (create_pcb(&child_pid, child_pd, parent_pcb) < 0) {
        // TODO: delete page directory
        return -1;
    }
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));


    if (create_tcb(child_pid, &child_tid) < 0) {
        // TODO: delete page directory
        // TODO: delete_pcb of parent
        return -1;
    }
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    tcb_t *child_tcb;
    assert(child_tcb = find_tcb(child_tid));
#ifndef NDEBUG
    /* Register this task with simics for better debugging */
    sim_reg_child(child_pd, parent_pd);
#endif
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));


    uint32_t *child_kernel_esp_on_ctx_switch;
    uint32_t *parent_kern_stack_hi = get_kern_stack_hi(parent_tcb);
    uint32_t *child_kern_stack_hi = get_kern_stack_hi(child_tcb);

    child_kernel_esp_on_ctx_switch = save_child_regs(parent_kern_stack_hi,
                                                     child_kern_stack_hi,
                                                     child_pd);
```

```
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    /* Set child's kernel esp */
    //affirm(child_kernel_esp_on_ctx_switch);
    set_kern_esp(child_tcb, child_kernel_esp_on_ctx_switch);
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    /* If logging is set to debug, this will print stuff */
    log_print_parent_and_child_stacks(parent_tcb, child_tcb );
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    /* After setting up child stack and VM, register with scheduler */
    if (make_thread_runnable(get_tcb_tid(child_tcb)) < 0)
        return -1;
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    /* Only parent will return here */
    assert(get_running_tid() == get_tcb_tid(parent_tcb));
    //affirm(is_valid_pd(parent_pd));
    //affirm(is_valid_pd(child_pd));

    return get_tcb_tid(child_tcb);
}
```

```
#ifndef LIFE_CYCLE_H_
#define LIFE_CYCLE_H_

void _vanish( void );


#endif /* LIFE_CYCLE_H_ */
```

```
.globl update_child_esp

# void update_child_esp(uint32_t *child_esp,uint32_t child_stack_lo, uint32_t paren
t_lowest)
update_child_esp:
    leal 4(%esp), %eax # save parent esp
    movl %eax, 4(%esp)



.globl save_child_regs
//TODO if possible do it in C
/* void *save_child_regs(void *parent_kern_esp, void *child_kern_esp
 *                       void *child_cr3 ); */
save_child_regs:
    /*************************************************************************/
    /* header: necessary steps to guarantee correct function return          */
    pushl %ebp /* save %ebp                                                  */
    movl %esp, %ebp /* update %ebp                                           */
    /*************************************************************************/

    movl 8(%ebp), %ecx /* %ecx holds parent_kern_esp, points to parent tid */
    movl 12(%ebp), %esp /* %esp set to child stack highest address */

    /* This part copies relevant parts of the parent stack to child stack.
     * Although tedious, we avoid loops for clarity
     */
    movl -4(%ecx), %edx /* parent SS in %edx since fork() privilege change */
    pushl %edx
    movl -8(%ecx), %edx /* parent ESP in %edx since fork() privilege change */
    pushl %edx
    movl -12(%ecx), %edx /* parent EFLAGS in %edx */
    pushl %edx
    movl -16(%ecx), %edx /* parent CS in %edx */
    pushl %edx
    movl -20(%ecx), %edx /* parent EIP in %edx */
    pushl %edx
    //movl -24(%ecx), %edx /* assume no parent Error code in %edx */
    //pushl %edx
    movl -24(%ecx), %edx /* parent ebp in %edx from call_fork() */
    pushl %edx
    movl -28(%ecx), %edx /* parent edi in %edx from call_fork() */
    pushl %edx
    movl -32(%ecx), %edx /* parent ebx in %edx from call_fork() */
    pushl %edx
    movl -36(%ecx), %edx /* parent esi in %edx from call_fork() */
    pushl %edx
    movl -40(%ecx), %edx /* parent ds in %edx from call_fork() */
    pushl %edx
    movl -44(%ecx), %edx /* parent es in %edx from call_fork() */
    pushl %edx
    movl -48(%ecx), %edx /* parent fs in %edx from call_fork() */
    pushl %edx
    movl -52(%ecx), %edx /* parent gs in %edx from call_fork() */
    pushl %edx
    movl -56(%ecx), %edx /* return address to call_fork() in %edx */
    pushl %edx

    /* This part onwards must match with context switcher */
    leal 32(%esp), %edx /* parent ebp in %edx */
    pushl %edx

    /* stuff to store child tid in stack position for future eax value */
    //movl 12(%ebp), %edx /* parent eax in %edx, contains child tid now */
    //movl (%edx), %edx /* dereference child stack highest address to get child
                      //* tid */
    movl $0, %edx

    pushl %edx

# TODO: Just subl 20 to child esp instead
#       Or just 5 times, subl 4 so we can comment what's up
    movl -68(%ecx), %edx /* parent ebx in %edx */
    pushl %edx
    movl -72(%ecx), %edx /* parent ecx in %edx */
    pushl %edx
    movl -76(%ecx), %edx /* parent edx in %edx */
    pushl %edx
    movl -80(%ecx), %edx /* parent edi in %edx */
    pushl %edx
    movl -84(%ecx), %edx /* parent esi in %edx */
    pushl %edx
    movl 16(%ebp), %edx /* parent cr3 in %edx */
    pushl %edx

    movl %esp, %eax /* return the stack pointer for child to use */

    /*************************************************************************/
    /* footer: necessary steps to guarantee correct function return          */
    movl %ebp, %esp /* %esp points to return address's address + 4           */
    popl %ebp /* restore %ebp, %esp points to return address                 */
    ret
    /*************************************************************************/
```

```
/** @file set_status.c
 *  @brief Implements fake set_status()
 */
#include <x86/asm.h>   /* outb() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <logger.h>    /* log() */
#include <timer_driver.h>       /* get_total_ticks() */
#include <scheduler.h> /* get_running_thread() */
#include <task_manager.h> /* set_task_exit_status() */
void
set_status( int status )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);
    log_info("set_status(): "
             "status: %d", status);

    set_task_exit_status(status);
}
```

```
/** @file task_vanish.c
 *  @brief Implements fake task_vanish()
 */
#include <x86/asm.h>   /* outb() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <logger.h>    /* log() */
#include <timer_driver.h>       /* get_total_ticks() */
#include <lib_thread_management/thread_management.h> /* _deschedule() */

#include <lib_life_cycle/life_cycle.h>

void
task_vanish( int status )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);
    log_info("call task_vanish");

    // Does not return
    _vanish();

    //while(1)
    //{
    //  continue;
    //}
}
```

```
/** @file vanish.c
 *  @brief Implements vanish()
 */
#include <x86/asm.h>   /* outb() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <logger.h>    /* log() */
#include <timer_driver.h>       /* get_total_ticks() */
#include <scheduler.h>          /* yield_execution() */
#include <task_manager_internal.h>
#include <lib_thread_management/hashmap.h>

//static mutex_t hacky_vanish_mux;
//static int hacky_vanish_mux_init = 0;

void
free_tcb_callback(tcb_t *tcb, void *unused)
{
    /* remove tcb from hashmap */
    map_remove(tcb->tid);
    free_tcb(tcb);
}

void
_vanish( void ) // int on_error )
{
    log("_vanish(): started executing _vanish()");

    // TODO _vanish on error (killed thread)

    /* Get TCB and PCB and obtain critical section */
    tcb_t *tcb = get_running_thread();
    pcb_t *owning_task = tcb->owning_task;
    mutex_lock(&(owning_task->set_status_vanish_wait_mux));

    /* Remove from PCB's list of threads and update task thread count */
    affirm(Q_GET_FRONT(&(owning_task->owned_threads)) == tcb);
    Q_REMOVE(&(owning_task->owned_threads), tcb, owning_task_thread_list);
    (owning_task->num_threads)--;

    /* At this point we have possibly broken invariant that num_threads > 0 */


    /* Not the last task, clean up and yield execution */
    if (get_num_threads_in_owning_task(tcb) > 0) {
        mutex_unlock(&(owning_task->set_status_vanish_wait_mux));
        affirm(yield_execution(DEAD, -1, free_tcb_callback, NULL) == 0);

    /* Last task thread contacts parent PCB */
    } else {

        /* TODO Tell all children tasks that their parent is init() now */


        mutex_unlock(&(owning_task->set_status_vanish_wait_mux));



        /* Insert into parent PCB's list of vanished child tasks */
        pcb_t *parent_pcb = owning_task->parent_pcb;
        affirm(parent_pcb);

        mutex_lock(&(parent_pcb->set_status_vanish_wait_mux));
        Q_INSERT_TAIL(&(parent_pcb->vanished_child_tasks_list),
                      owning_task, vanished_child_tasks_link);

        /* Look at list of waiting parent threads, if non-empty, wake them up */
        int parent_tid = -1;
        tcb_t *waiting_tcb = Q_GET_FRONT(&(parent_pcb->waiting_threads_list));
        if (waiting_tcb) {
            affirm(waiting_tcb);
            parent_tid = get_tcb_tid(waiting_tcb);
            Q_REMOVE(&(parent_pcb->waiting_threads_list), waiting_tcb,
                     waiting_threads_link);
        }
        mutex_unlock(&(parent_pcb->set_status_vanish_wait_mux));
        affirm(yield_execution(DEAD, parent_tid, free_tcb_callback, NULL)
               == 0);
    }
}


void
vanish( void )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);
    log_info("call vanish");
    _vanish();
}
```

```
#include <asm_interrupt_handler_template.h>


CALL_W_DOUBLE_ARG(new_pages)

CALL_W_SINGLE_ARG(remove_pages)

CALL_VAR_ARGS_FAULT_HANDLER_W_ERROR(pagefault_handler)
```

```
/** @file is_valid_pd.c
 *  @brief Implements page directory and page consistency checks via functions
 *         is_valid_pd() and is_valid_pt()
 */

#include <memory_manager.h>
#include <common_kern.h>/* USER_MEM_START */
#include <logger.h>     /* log */
#include <page.h>       /* PAGE_SIZE */
#include <assert.h>     /* assert, affirm */
#include <physalloc.h>  /* is_physframe() */
#include <simics.h>     /* MAGIC_BREAK */
#include <memory_manager_internal.h>

/** @brief Checks if paget table at index i of a page directory is valid or not.
 *
 *  If the address is < USER_MEM_START, does not validate the address.
 *  TODO perhaps find a way to validate such addresses as well?
 *
 *  @param pt Page table address to check
 *  @param pd_index The index this page table address was stored in the
 *         page directory
 */
int
is_valid_pt( uint32_t *pt, int pd_index )
{
    /* Basic page table address checks */
    if (!pt) {
        log_warn("is_valid_pt(): "
                 "pt: %p is NULL!", pt);
        return 0;
    }
    if (!PAGE_ALIGNED(pt)) {
        log_warn("is_valid_pd(): "
                 "pt: %p is not page aligned!", pt);
        return 0;
    }
    if ((uint32_t) pt >= USER_MEM_START) {
        log_warn("is_valid_pt(): pt: %p is above USER_MEM_START!", pt);
        return 0;
    }

    /* Iterate over page table and check each entry */
    for (int i = 0; i < PAGE_SIZE / sizeof(uint32_t); ++i) {
        uint32_t pt_entry = pt[i];
        //if (pd_index == 0x3ff) {
        //  if (pt_entry) log("pt_entry:0x%08lx", pt_entry);
        //}
        assert((((pt_entry) != 0) && (TABLE_ADDRESS(pt_entry) != 0))
               || (pt_entry == 0));


        assert(TABLE_ENTRY_INVARIANT(pt_entry));

        /* Check only if entry is non-NULL, ignoring bottom 12 bits */
        if (TABLE_ADDRESS(pt_entry)) {

            uint32_t phys_address = TABLE_ADDRESS(pt_entry);

            /* Present bit must be set */
            if (!(pt_entry & PRESENT_FLAG)) {
                log_warn("is_valid_pt(): "
                         "present bit not set for "
                         "pt:%p "
                         "pd_index:0x%08lx "
                         "pt_entry:0x%08lx "
                         "phys_address:0x%08lx "
                         "pt_index:0x%08lx",
                         pt, pd_index, pt_entry, phys_address, i);
                log_warn("virtual address of pt_entry:%p", &pt_entry);
                return 0;
            }
            /* pt holds physical frames for user memory */
            if (pd_index >= (USER_MEM_START >> PAGE_DIRECTORY_SHIFT)) {

                if (pt_entry & GLOBAL_FLAG) {
                    log_warn("User page cannot have global flag enabled!"
                             "pt:%p "
                             "pd_index:0x%08lx "
                             "pt_entry:0x%08lx "
                             "phys_address:0x%08lx "
                             "pt_index:0x%08lx",
                             pt, pd_index, pt_entry, phys_address, i);
                    MAGIC_BREAK;
                    return 0;
                }

                /* Frame cannot be equal to pt */
                if (TABLE_ADDRESS(pt_entry) == (uint32_t) pt) {
                    log_warn("is_valid_pt(): "
                             "pt at address:%p same address as frame physical "
                             "address: %p with pt_entry: 0x%08lx at "
                             "index: 0x%08lx",
                             pt, (uint32_t *) phys_address, pt_entry, i);
                }


                /* Frame must be a valid physical address by physalloc */
                if (!is_physframe(phys_address)) {
                    log_warn("is_valid_pt(): "
                             "pt at address: %p has invalid frame physical "
                             "address: %p with pt_entry: 0x%08lx at "
                             "index: 0x%08lx",
                             pt, (uint32_t *) phys_address, pt_entry, i);
                    return 0;
                }
            /* pt holds physical frame in kernel VM */
            } else {

                /* Frame must be < USER_MEM_START */
                if (phys_address >= USER_MEM_START) {
                    log_warn("is_valid_pt(): "
                             "pt at address: %p has invalid frame physical "
                             "address: %p >= USER_MEM_START with pt_entry: "
                             "0x%08lx at index: 0x%08lx",
                             pt, (uint32_t *) phys_address, pt_entry, i);
                    return 0;
                }
            }
        }
    }
    return 1;
}

/** @brief Checks if supplied page directory is valid or not
 *
 *  TODO keep a record of all page directory addresses ever given out?
 *
 *  @param pd Page directory to check
 *  @return 1 if pd points to a valid page directory, 0 otherwise
 */
int
is_valid_pd( void *pd )
{
```

```
    return 1; // All page directories are hecking cute and valid!
    /* Basic page directory address checks */
    if (!pd) {
        log_warn("is_valid_pd(): pd: %p is NULL!", pd);
        return 0;
    }
    if (!PAGE_ALIGNED(pd)) {
        log_warn("is_valid_pd(): pd: %p is not page aligned!", pd);
        return 0;
    }
    if ((uint32_t) pd >= USER_MEM_START) {
        log_warn("is_valid_pd(): pd: %p is above USER_MEM_START!", pd);
        return 0;
    }
    /* Iterate over page directory and check each entry */
    uint32_t **pd_cast = (uint32_t **) pd;
    for (int i = 0; i < PAGE_SIZE / sizeof(uint32_t); ++i) {
        uint32_t *pd_entry = pd_cast[i];
        assert(TABLE_ENTRY_INVARIANT(pd_entry));


        /* Check only if entry is non-NULL, ignoring bottom 12 bits */
        if (TABLE_ADDRESS(pd_entry)) {
            uint32_t *pt = (uint32_t *) TABLE_ADDRESS(pd_entry);

            /* Present bit must be set */
            if (!((uint32_t) pd_entry & PRESENT_FLAG)) {
                log_warn("is_valid_pd(): "
                "pd at address: %p has non-present pt at address: %p "
                         "with pd_entry: 0x%08lx at index: 0x%08lx",
                         pd, pt, pd_entry, i);
                return 0;
            }
            /* Page table at address must be valid */
            if (!is_valid_pt(pt, i)) {
                log_warn("is_valid_pd(): "
                "pd at address: %p has invalid pt at address: %p "
                         "with pd_entry: 0x%08lx at index: 0x%08lx",
                         pd, pt, pd_entry, i);
                return 0;
            }
        }
    }
    return 1;
}
```

```
/** @file new_pages.c
 *  @brief new_pages syscall handler
 *
 *  @author Nicklaus Choo (nchoo)
 */

#include <common_kern.h>/* USER_MEM_START */
#include <x86/cr.h>
#include <logger.h>
#include <memory_manager.h>
#include <assert.h>
#include <task_manager.h>
#include <scheduler.h>
#include <x86/asm.h>   /* outb() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <page.h> /* PAGE_SIZE */
#include <simics.h>
#include <physalloc.h>
#include <memory_manager_internal.h>


// TODO locking
int
new_pages( void *base, int len )
{
    assert(is_valid_pd(get_tcb_pd(get_running_thread())));

    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    log("new_pages(): "
        "base:%p, len:0x%08lx", base, len);

    if ((uint32_t)base < USER_MEM_START) {
        log_info("new_pages(): "
                 "base < USER_MEM_START");
        return -1;
    }
    if (!PAGE_ALIGNED(base)) {
        log_info("new_pages(): "
                 "base not page aligned!");
        return -1;
    }
    if (len <= 0) {
        log_info("new_pages(): "
                 "len <= 0!");
        return -1;
    }
    if (len % PAGE_SIZE != 0) {
        log_info("new_pages(): "
                 "len is not a multiple of PAGE_SIZE!");
        return -1;
    }
    /* Check if enough frames to fulfill request */
    uint32_t pages_to_alloc = len / PAGE_SIZE;
    if (num_free_phys_frames() < pages_to_alloc) {
        log_info("new_pages(): "
                 "not enough free frames to satisfy request!");
        return -1;
    }
    /* Check if any portion is currently allocated in task address space */
    char *base_char = (char *) base;
    for (uint32_t i = 0; i < len; ++i) {
        if (is_user_pointer_allocated(base_char + i)) {
            log_info("new_pages(): "
                     "%p is already allocated!", base_char + i);
            return -1;
        }
    }
    /* Allocate a zero frame to each PAGE_SIZE region of memory */
    int res = 0;
    for (uint32_t i = 0; i < len / PAGE_SIZE; ++i) {
        assert(res == 0);

        /* We mark in the page table if the allocated page is the first */
        if (i == 0) {
            res += allocate_user_zero_frame((void *)TABLE_ADDRESS(get_cr3()),
                                            (uint32_t) base + (i * PAGE_SIZE),
                                            NEW_PAGE_BASE_FLAG);
        } else {
            res += allocate_user_zero_frame((void *)TABLE_ADDRESS(get_cr3()),
                                            (uint32_t) base + (i * PAGE_SIZE),
                                            NEW_PAGE_CONTINUE_FROM_BASE_FLAG);
        }
        /* If any step fails, unallocate zero frame, return -1 */
        if (res < 0) {
            log_info("new_pages(): "
                     "unable to allocate zero frame");

            /* Cleanup */
            for (uint32_t j = 0; j < i; ++j) {
                unallocate_user_zero_frame((void *)TABLE_ADDRESS(get_cr3()),
                                           (uint32_t) base + (j * PAGE_SIZE));
            }
            return -1;
        }
    }
    /* TODO jank get and set cr3() to flush TLB entries */
    set_cr3(get_cr3());
    return res;
}
```

```
/** @file pagefault_handler.c
 *  @brief Functions for page fault handling
 *  TODO write a macro for syscall handler/install handler wrappers
 *  @author Nicklaus Choo (nchoo)
 */

#include <cr.h>                 /* get_cr2() */
#include <asm.h>                /* outb() */
#include <seg.h>                /* SEGSEL_USER_CS */
#include <page.h>               /* PAGE_SIZE */
#include <assert.h>             /* panic() */
#include <scheduler.h>          /* get_running_tid */
#include <common_kern.h>        /* USER_MEM_START  */
#include <memory_manager.h>     /* zero_page_pf_handler */
#include <install_handler.h>    /* install_handler_in_idt() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */


#include <simics.h>
#include <logger.h> /* log() */
#include <lib_life_cycle/life_cycle.h>

/* Page fault error code flag definitions */

/** @brief if 0, the fault was caused by a non-present page
 *         if 1, the fault was caused by a page-level protection violation
 */
#define P_BIT    (1 << 0)

/** @brief if 0, the access causing the fault was a read
 *         if 1, the access causing the fault was a write
 */
#define WR_BIT   (1 << 1)

/** @brief if 0, the access causing the fault originated when the processor
 *                was executing in supervisor mode
 *         if 1, the access causing the fault originated when the processor
 *                was executing in user mode
 */
#define US_BIT   (1 << 2)

/** @brief if 0, the fault was not caused by reserved bit violation
 *         if 1, the fault was caused by reserved bits set to 1 in a
 *                page directory
 */
#define RSVD_BIT (1 << 3)


/** @brief Prints out the offending address on and calls panic().
 *
 *  @return Void.
 */
void
pagefault_handler( uint32_t *ebp )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    int error_code = *(ebp + 1);
    int eip = *(ebp + 2);
    int cs = *(ebp + 3);
    int eflags = *(ebp + 4);
    int esp, ss;

    /* More args on stack if it was from user space */
    if (cs == SEGSEL_USER_CS) {
        esp = *(ebp + 5);
        ss = *(ebp + 6);
    }
    (void) esp;
    (void) ss;

    uint32_t faulting_vm_address = get_cr2();

    // TODO Add the swexn() execution here
    /* TODO: acknowledge signal and call user handler  */

    (void)cs;

    char user_mode[] = "[USER-MODE]";
    char supervisor_mode[] = "[SUPERVISOR-MODE]";
    char *mode = (error_code & US_BIT) ?
        user_mode : supervisor_mode;

    /* If the fault happened while we were running in kernel AKA supervisor
     * mode, something really bad happened and we need to crash
     */
    if ((error_code & US_BIT) == 0) {
        panic("pagefault_handler(): %s "
              "pagefault while running in kernel mode! "
              "error_code:0x%x "
              "eip:0x%x "
              "cs:0x%x "
              "faulting_vm_address:%p",
              mode, error_code, eip, cs, faulting_vm_address);
    }
    /* If the fault happened because reserved bits were accidentally set
     * in the page directory, the page directory is corrupted and we need to
     * crash.
     */
    if (error_code & RSVD_BIT) {

        /* Get offending page directory entry */
        uint32_t pd_index = PD_INDEX(faulting_vm_address);
        uint32_t **pd = (uint32_t **) TABLE_ADDRESS(get_cr3());
        affirm(pd);
        uint32_t *pd_entry = pd[pd_index];

        panic("pagefault_handler(): %s "
              "pagefault due to corrupted page directory entry (pd_entry) "
              "reserved bits "
              "error_code:0x%x "
              "eip:0x%x "
              "cs:0x%x "
              "faulting_vm_address:%p "
              "pd_entry:%p ",
              mode, error_code, eip, cs, faulting_vm_address, pd_entry);
    }

    /* Fault was a write */
    if (error_code & WR_BIT) {
        /* Check if this was a ZFOD allocated page, since those pages are
         * marked as read-only in the page directory */
        if (zero_page_pf_handler(faulting_vm_address) == 0) {
            return;
        }
        log_info("%s Page fault at vm address:0x%lx at instruction 0x%lx! "
                 "Writing into read-only page",
                 mode, faulting_vm_address, eip);


        _vanish();
    } else {
```

```
        log_info("%s Page fault at vm address:0x%lx at instruction 0x%lx! "
                "reading permissions wrong",
                 mode, faulting_vm_address, eip);

        _vanish();
    }


    if (!(error_code & P_BIT)) {
        log_info("%s Page fault at vm address:0x%lx at instruction 0x%lx! %s",
                 mode, faulting_vm_address, eip,
                faulting_vm_address < PAGE_SIZE ?
                "Null dereference." : "Page not present.");
        _vanish();
    } else {

        log_info("%s Page fault at vm address:0x%lx at instruction 0x%lx! %s",
                mode, faulting_vm_address, eip,
                faulting_vm_address < PAGE_SIZE ?
                "Null dereference." : "page-level protection violation.");
        _vanish();
    }

    // TODO not sure if this is needed
    /* Jank check but reasonable for now */
    if (cs == SEGSEL_USER_CS && eip < USER_MEM_START) {
        log_info("[tid %d] %s Page fault at vm address:0x%lx at instruction 0x%lx!
"
                "User mode trying to access kernel memory",
                get_running_tid(),mode, faulting_vm_address, eip);
        _vanish();
    }

    MAGIC_BREAK;
    panic("PAGEFAULT HANDLER BROKEN!\n "
                  "error_code:0x%08x\n "
                  "eip:0x%08x\n "
                  "cs:0x%08x\n "
                  "eflags:0x%08x\n "
                  "faulting_vm_address: 0x%081x",
                  error_code, eip, cs, eflags, faulting_vm_address);


}
```

```
/** @file physalloc.c
 *  @brief Contains functions implementing interface functions for physalloc.h
 *
 *  A doubly linked list is used to store physical frames that have been
 *  used at least once but are currently free. This list is called reuse_list.
 *
 *  Whenever a new physical frame address is requested, first check the
 *  reuse_list for any free physical frame addresses. If there is one, return
 *  that free physical frame address. Else, return max_free_phys_address and
 *  update max_free_phys_address
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */

#include <physalloc.h>

#include <string.h>         /* memcpy() */
#include <assert.h>         /* affirm() */
#include <variable_queue.h> /* Q_NEW_LINK() */
#include <malloc.h>         /* malloc() */
#include <common_kern.h>    /* USER_MEM_START */
#include <page.h>           /* PAGE_SIZE */
#include <logger.h>         /* log */
#include <lib_thread_management/mutex.h> /* mutex_t */

#define PHYS_FRAME_ADDRESS_ALIGNMENT(phys_address)\
    ((phys_address & (PAGE_SIZE - 1)) == 0)

/** Number of pages as of yet unclaimed from system.
 *  Equals to machine_phys_frames() - (max_free_address / PAGE_SIZE) */
#define TOTAL_USER_FRAMES (machine_phys_frames() - (USER_MEM_START / PAGE_SIZE))

/** Number of pages as of yet unclaimed from system.
 *  Equals to machine_phys_frames() - (max_free_address / PAGE_SIZE) */
#define UNCLAIMED_PAGES (machine_phys_frames() - (max_free_address / PAGE_SIZE))

/* Whether physical frame allocater is initialized */
static int physalloc_init = 0;

/* Address of the highest free physical frame currently in list */
static uint32_t max_free_address;

typedef struct {
    uint32_t top; /* Index of next empty address in array */
    uint32_t len;
    uint32_t *data;
} stack_t;

static stack_t reuse_stack;

static mutex_t mux;

/** @brief Checks if a physical address is page aligned and could have
 *         been given out by physalloc
 *
 *  @param physaddress Physical address
 *  @return 1 if valid, 0 otherwise
 */
int
is_physframe( uint32_t phys_address )
{
    if (!PHYS_FRAME_ADDRESS_ALIGNMENT(phys_address)) {
        log_warn("0x%08lx is not page aligned!", phys_address);
        return 0;
    }
    if (!(USER_MEM_START <= phys_address && phys_address <= max_free_address)) {
        log_warn("0x%08lx is not in valid address range!", phys_address);
        return 0;
    }
    return 1;
}

/** @brief Returns number of free physical frames with physical address at
 *         least USER_MEM_START
 *
 *  @return Number of free physical frames
 */
uint32_t
num_free_phys_frames( void )
{
    return UNCLAIMED_PAGES + reuse_stack.top; //TODO why do we add this guy
}

/** @brief Initializes physical allocator family of functions
 *
 *  Must be called once and only once
 *
 *  @param Void.
 */
void
init_physalloc( void )
{
    /* Initialize once and only once */
    affirm(!physalloc_init);

    reuse_stack.top = 0;
    reuse_stack.len = PAGE_SIZE / sizeof(uint32_t);
    reuse_stack.data = smalloc(PAGE_SIZE);

    /* Crash kernel if we can't initialize phys frame allocator */
    affirm(reuse_stack.data);

    /* USER_MEM_START is the system wide 0 frame */
    max_free_address = USER_MEM_START + PAGE_SIZE;
    mutex_init(&mux);
    physalloc_init = 1;
}

/** @brief Allocates a physical frame by returning its address
 *
 *  If there is a reusable free frame, we use that frame first.
 *
 *  @return Next free physical frame address
 */
uint32_t
physalloc( void )
{
    /* First time running, initialize */
    if (!physalloc_init)
        init_physalloc();

    mutex_lock(&mux);

    if (reuse_stack.top > 0) {
        mutex_unlock(&mux);
        return reuse_stack.data[--reuse_stack.top];
    }

    if (UNCLAIMED_PAGES == 0) {
        mutex_unlock(&mux);
        return 0; /* No more pages to allocate */
    }
```

```
    uint32_t frame = max_free_address;
    max_free_address += PAGE_SIZE;

    mutex_unlock(&mux);

    log("physalloc(): returned frame 0x%lx", frame);
    assert(is_physframe(frame));
    return frame;
}

/** @brief Frees a physical frame address
 *
 *  Does necessary but not sufficient checksto see if it is indeed a valid
 *  address before freeing (double freeing is not checked).
 *  Requires that this phys_address was returned from a call to
 *  physalloc(), else behavior is undefined. Free every allocated physical
 *  address exactly once.
 *
 *  @param phys_address Physical address to be freed.
 *  @return Void.
 */
void
physfree(uint32_t phys_address)
{
    mutex_lock(&mux);
    affirm(is_physframe(phys_address));

    /* Add phys_address to stack, growing it if necessary. */
    if (reuse_stack.top >= reuse_stack.len) {
        assert(reuse_stack.top == reuse_stack.len);

        uint32_t *new_data = smalloc(reuse_stack.len * 2 * sizeof(uint32_t));
        if (!new_data) {
            log_warn("[ERROR] Losing free physical frames \
                    - no more kernel space.");
            mutex_unlock(&mux);
            return;
        }

        memcpy(new_data, reuse_stack.data, reuse_stack.len * sizeof(uint32_t));

        sfree(reuse_stack.data, reuse_stack.len * sizeof(uint32_t));
        reuse_stack.data = new_data;
        reuse_stack.len *= 2;
    }

    reuse_stack.data[reuse_stack.top++] = phys_address;
    mutex_unlock(&mux);
    log("physfree freed frame 0x%lx", phys_address);
}

/** @brief Tests physalloc and physfree
 *
 *  @return Void.
 */
void
test_physalloc( void )
{
    log_info("Testing physalloc(), physfree()");
    uint32_t a, b, c;
    /* Quick test for alignment, we allocate in consecutive order */
    a = physalloc();
    assert(a == USER_MEM_START);
    b = physalloc();
    assert(b == a + PAGE_SIZE);

    /* Quick test for reusing free physical frames */
    physfree(a);
    c = physalloc(); /* c reuses a */
    assert(a == c);
    a = physalloc();
    assert(a == USER_MEM_START + 2 * PAGE_SIZE);

    /* Test reuse the latest freed phys frame */
    physfree(b);
    physfree(c);
    physfree(a);
    assert(physalloc() == a);
    assert(physalloc() == c);
    assert(physalloc() == b);
    physfree(USER_MEM_START + 2 * PAGE_SIZE);
    physfree(USER_MEM_START + 1 * PAGE_SIZE);
    physfree(USER_MEM_START);

    /* Use all phys frames */
    int total = TOTAL_USER_FRAMES;
    int i = 0;
    uint32_t all_phys[1024];
    log("after all_phys");
    while (i < 1024) {
        all_phys[i] = physalloc();
        assert(all_phys[i]);
        i++;
        total--;
    }
    log("total frames supported:%08x",
            (unsigned int) TOTAL_USER_FRAMES);
    assert(total == TOTAL_USER_FRAMES - 1024);
    /* all phys frames, populate reuse list */
    assert(i == 1024);
    while (i > 0) {
        i--;
        physfree(all_phys[i]);
        total++;
    }
    assert(total == TOTAL_USER_FRAMES);
    assert(i == 0);

    /* exhaust reuse list, check implicit stack ordering of reuse list */
    while (i < 1024) {
        uint32_t addr = physalloc();
        (void) addr;
        assert(all_phys[i] == addr);
        assert(addr);
        if (i == 0) assert(all_phys[i] == USER_MEM_START);
        else assert(all_phys[i-1] + PAGE_SIZE == all_phys[i]);
        i++;
        total--;
    }
    /* free everything */
    while (i > 0) {
        i--;
        physfree(all_phys[i]);
        total++;
    }
    /*use ALL phys frames */
    assert(i == 0);
    assert(total == TOTAL_USER_FRAMES);
    uint32_t x = 0;
    while (total > 0) {
        total--;
        x = physalloc();
    }
    log("last frame start address:%lx", x);
```

```
    assert(!physalloc());

    /* put them all back */
    log("put all into linked list");
    while (total < TOTAL_USER_FRAMES) {
        total++;
        physfree(x);
        x -= PAGE_SIZE;
    }
    log_info("Tests passed!");
}
```

```
/** @file remove_pages.c
 *  @brief remove_pages syscall handler
 *
 *  @author Nicklaus Choo (nchoo)
 */

#include <common_kern.h>/* USER_MEM_START */
#include <x86/cr.h>
#include <logger.h>
#include <memory_manager.h>
#include <assert.h>
#include <task_manager.h>
#include <scheduler.h>
#include <x86/asm.h>   /* outb() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <page.h> /* PAGE_SIZE */
#include <simics.h>
#include <physalloc.h>
#include <memory_manager_internal.h>

// TODO locking
int
remove_pages( void *base )
{
    uint32_t **pd = get_tcb_pd(get_running_thread());
    assert(is_valid_pd(pd));

    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    if ((uint32_t)base < USER_MEM_START) {
        log_info("remove_pages(): "
                 "base < USER_MEM_START");
        return -1;
    }
    if (!PAGE_ALIGNED(base)) {
        log_info("remove_pages(): "
                 "base not page aligned!");
        return -1;
    }
    /* Check if base is legitimately in page table. Since page table is
     * valid, if ptep is NULL then base was not even allocated */
    uint32_t *ptep = get_ptep((const uint32_t **) pd, (uint32_t) base);
    if (!ptep) {
        log_info("remove_pages(): "
                 "unable to get page table entry pointer:");
        return -1;
    }
    /* Check if base was allocated by previous call to new_pages() */
    int sys_prog_flag = SYS_PROG_FLAG(*ptep);
    assert(is_valid_sys_prog_flag(sys_prog_flag));

    if (sys_prog_flag != NEW_PAGE_BASE_FLAG) {
        log_info("remove_pages(): "
                 "base:%p not previously allocated by new_pages(), "
                 "sys_prog_flag:0x%08x",
                 base, sys_prog_flag);
        return -1;
    }
    /* Free the first frame */
    uint32_t curr = (uint32_t) base;
    unallocate_frame(pd, curr);
    curr += PAGE_SIZE;
    assert(is_valid_pd(pd));
    affirm(PAGE_ALIGNED(curr));

    /* Free remaining frames */
    ptep = get_ptep((const uint32_t **) pd, curr);
    while (ptep && (SYS_PROG_FLAG(*ptep) == NEW_PAGE_CONTINUE_FROM_BASE_FLAG)) {
        unallocate_frame(pd, curr);
        curr += PAGE_SIZE;
        assert(is_valid_pd(pd));
        affirm(PAGE_ALIGNED(curr));

        ptep = get_ptep((const uint32_t **) pd, curr);
    }
    log("remove_pages(): "
            "unallocated base:%p, len:%d", base,
            curr - ((uint32_t) base));
    /* TODO jank get and set cr3() to flush TLB entries */
    set_cr3(get_cr3());

    return 0;
}
```

```
#include <asm_interrupt_handler_template.h>

CALL_W_FOUR_ARG(readfile)

CALL_W_RETVAL_HANDLER(halt)
```

```
#ifndef CALL_HALT_H_
#define CALL_HALT_H_

/* Calls hlt asm instruction */
void call_hlt( void );

#endif /* CALL_HALT_H_ */
```

```
.globl call_hlt

call_hlt:
    hlt
    ret
```

```
/** @file halt.c
 *  @brief Halt syscall handler
 */
#include <assert.h>     /* panic */
#include <simics.h>     /* sim_halt */
#include "call_halt.h"  /* call_hlt */

/** @brief Halt OS. Also used as handler for halt syscall */
void
halt( void )
{
    /* HLT is no-op in simics, therefore call simics halt */
    sim_halt();

    call_hlt();

    /* NOTREACHED */
    panic("NOTREACHED: After halt instruction");
}
```

```
/** @file readfile.c
 *  @brief Readfile syscall handler
 */
#include <asm.h>                /* outb */
#include <assert.h>             /* affirm */
#include <console.h>            /* putbytes */
#include <exec2obj.h>           /* MAX_EXECNAME_LEN */
#include <memory_manager.h>     /* is_valid_user_pointer/string */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

/** @brief Handler for readfile syscall. */
int
readfile( char *filename, char *buf, int count, int offset )
{
#include <cr.h>
#include <page.h>
    affirm(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));

    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    if (!is_valid_user_string(filename, MAX_EXECNAME_LEN))
        return -1;

    /* Ensure all of buf is valid */
    for (int i=0; i < count; ++i) {
        if (!is_valid_user_pointer(buf + i, READ_WRITE))
            return -1;
    }

    return getbytes(filename, offset, count, buf);
}
```

```
#include <asm_interrupt_handler_template.h>

CALL_W_RETVAL_HANDLER(gettid)

CALL_W_RETVAL_HANDLER(get_ticks)

CALL_W_SINGLE_ARG(yield)

CALL_W_SINGLE_ARG(deschedule)

CALL_W_SINGLE_ARG(make_runnable)

CALL_W_SINGLE_ARG(sleep)
```

```
/** @file deschedule.c
 *  @brief Contains deschedule interrupt handler and helper functions for
 *         installation
 *
 */

#include <asm.h>                /* outb() */
#include <stddef.h>             /* NULL */
#include <scheduler.h>          /* yield_execution() */
#include <memory_manager.h>     /* is_valid_user_pointer() */
#include <install_handler.h>    /* install_handler_in_idt() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

int
_deschedule( int *reject )
{
    if (!is_valid_user_pointer(reject, READ_ONLY))
        return -1;

    if (*reject == 0)
        return yield_execution(DESCHEDULED, -1, NULL, NULL);
    return 0;
}

/** @brief Deschedules currently running thread if *reject == 0
 *         Atomic w.r.t make_runnable
 *
 *  @param reject Pointer to integer describing whether to deschedule
 *  @return 0 on success, negative value on failure */
int
deschedule( int *reject )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return _deschedule(reject);
}
```

```c
/** @file get_ticks.c
 *  @brief Contains get_ticks interrupt handler
 */
#include <asm.h>                /* outb() */
#include <timer_driver.h>       /* get_total_ticks() */
#include <install_handler.h>    /* install_handler_in_idt() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

int
get_ticks( void )
{
    /* Acknowledge interrupt and return */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return get_total_ticks();
}
```

```
/** @file gettid.c
 *  @brief Contains gettid interrupt handler and helper functions for
 *         installation
 *
 *
 */
#include <scheduler.h>          /* get_running_tid() */
#include <asm.h>                /* outb() */
#include <install_handler.h>    /* install_handler_in_idt() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

int
gettid( void )
{
    /* Acknowledge interrupt and return */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return get_running_tid();
}
```

```
/** @file hashmap.c
 *  @brief A hashmap for use in thread bookkeeping
 *
 *  Hash function taken from https://github.com/skeeto/hash-prospector
 *
 *  @author Andre Nascimento (anascime) */

//TODO locking

#include <task_manager_internal.h> /* Q MACRO for tcb */
#include <lib_thread_management/hashmap.h>
#include <stdint.h> /* uint32_t */
#include <stddef.h> /* NULL */
#include <stdlib.h> /* malloc, free */
#include <string.h> /* memset */
#include <assert.h> /* affirm */

/** @brief Hash for placement into map
 *
 *  @param x Key to be hashed
 *  @return Hash.
 *  */
static uint32_t
hash( uint32_t x )
{
    x ^= x >> 16;
    x *= 0x7feb352d;
    x ^= x >> 15;
    x *= 0x846ca68b;
    x ^= x >> 16;
    return x;
}

/** @brief Insert status into hashmap.
 *
 *  @param map Hashmap in which to insert status
 *  @param tcb Pointer to tcb to insert
 *  @return Void.
 *  */
void
map_insert( tcb_t *tcb )
{
    uint32_t index = hash(tcb->tid) % NUM_BUCKETS;
    hashmap_queue_t *headp = &map.buckets[index];

    Q_INSERT_TAIL(headp, tcb, tid2tcb_queue);
}

/** @brief Get status in hashmap.
 *
 *  @param map Map from which to get status
 *  @param tid Id of thread status to look for
 *  @return Status, NULL if not found. */
tcb_t *
map_get(uint32_t tid)
{
    uint32_t index = hash(tid) % NUM_BUCKETS;
    hashmap_queue_t *headp = &map.buckets[index];

    tcb_t *curr = Q_GET_FRONT(headp);
    while (curr && curr->tid != tid)
        curr = Q_GET_NEXT(curr, tid2tcb_queue);

    return curr;
}

/** @brief Remove status from hashmap.
 *
 *  @param map Map from which to remove value of key
 *  @param tid Id of thread to remove
 *  @return Pointer to removed value on exit,
 *          NULL on failure.
 *
 */
tcb_t *
map_remove(uint32_t tid)
{
    uint32_t index = hash(tid) % NUM_BUCKETS;
    hashmap_queue_t *headp = &map.buckets[index];

    tcb_t *curr = Q_GET_FRONT(headp);

    while (curr) {
        if (curr->tid == tid) {
            Q_REMOVE(headp, curr, tid2tcb_queue);
            return curr;
        }
        curr = Q_GET_NEXT(curr, tid2tcb_queue);
    }

    return NULL;
}

/** @brief Initialize new hashmap
 *
 *  @param map Memory location in which to initialize
 *  @param num_buckets Number of buckets for this map
 *  @return Void
 */
void
map_init( void )
{
    memset(map.buckets, 0, sizeof(hashmap_queue_t) * NUM_BUCKETS);
    for (int i=0; i < NUM_BUCKETS; ++i)
        Q_INIT_HEAD(&map.buckets[i]);
    return;
}
```

```
/** @file hashmap.h
 *
 *  Definitions for thread hashmap.*/

#ifndef HASHMAP_H_
#define HASHMAP_H_

#include <stdint.h>         /* uint32_t */
#include <task_manager.h>   /* tcb_t */

/* Number of buckets in hashmap. 1024 is large enough to
 * minimize collisions while avoiding being wasteful with
 * memory. */
#define NUM_BUCKETS 1024

Q_NEW_HEAD(hashmap_queue_t, tcb);

/** @brief Struct containing the hashmaps buckets.
 *
 *  @param buckets Array with each linked list start
 *  @param num_buckets Length of buckets array */
typedef struct {
    hashmap_queue_t buckets[NUM_BUCKETS];
} hashmap_t;

/* Hashmap (tid -> tcb) for use in thread library. */
hashmap_t map;

/* Hashmap functions */
void map_insert(tcb_t *tcb);
tcb_t *map_get(uint32_t tid);
tcb_t *map_remove(uint32_t tid);
void map_init(void);

#endif /* HASHMAP_H_ */
```

```
/** @file make_runnable.c
 *  @brief Contains make_runnable interrupt handler and helper functions for
 *         installation
 *
 */

#include <asm.h>                /* outb() */
#include <scheduler.h>          /* make_thread_runnable() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <task_manager.h>       /* get_tcb_status() */
/** @brief Makes a previously descheduled thread runnable.
 *         Atomic w.r.t. deschedule.
 *
 *  @param tid Id of descheduled thread to make runnable
 *  @return 0 on success, negative error code if thread pointed to by tid
 *          does not exist or is not descheduled */
int
make_runnable( int tid )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    tcb_t *tcbp = find_tcb(tid);
    if (!tcbp || get_tcb_status(tcbp) != DESCHEDULED)
        return -1;

    /* move to runnable queue and mark as runnable */
    return make_thread_runnable(tid);
}
```

```c
/** @file mutex.c
 *  @brief A mutex object
 *
 *  Mutex uses atomically sections for synchronization
 *
 *  @author Andre Nascimento (anascime)
 *  */

#include "mutex.h"
#include <assert.h>     /* affirm_msg() */
#include <scheduler.h>  /* queue_t, make_thread_runnable, run_next_tcb */
#include <asm.h>        /* enable/disable_interrupts() */
#include <logger.h>     /* log */
#include <task_manager_internal.h> /* Q MACRO for tcb */

static void store_tcb_in_mutex_queue( tcb_t *tcb, void *data );

/** @brief Initialize a mutex
 *  @param mp Pointer to memory location where mutex should be initialized
 *
 *  @return 0 on success, negative number on error
 *  */
int
mutex_init( mutex_t *mp )
{
    if (!mp)
        return -1;

    Q_INIT_HEAD(&mp->waiters_queue);
    mp->initialized = 1;
    mp->owned = 0;

    return 0;
}

/** @brief Destroy a mutex
 *  @param mp Pointer to memory location where mutex should be destroyed
 *
 *  @return Void
 *  */
void
mutex_destroy( mutex_t *mp )
{
    affirm(0); /* TODO UNIMPLEMENTED */
}

/** @brief Lock mutex. Re-entrant.
 *  @param mp Mutex to lock
 *
 *  @return Void
 *  */
void
mutex_lock( mutex_t *mp )
{
    /* Exit if impossible to lock mutex, as just returning would give
     * thread the false impression that lock was acquired. */
    affirm(mp && mp->initialized);

    /* To simplify the mutex interface, we let a thread run mutex
     * guarded code even if the scheduler is not initialized. This
     * is fine because, as soon as we have 2 or more threads, the
     * scheduler must have been initialized. */
    if (!is_scheduler_init()) {
        mp->owned = 1;
        mp->owner_tid = get_running_tid();
        goto mutex_exit;
    }

    /* If calling thread already owns mutex, panic */
    if (mp->owned && get_running_tid() == mp->owner_tid) {
        panic("Thread trying to reacquire lock %p. Locks are not "
              "reentrant!", mp);
    }

    /* Atomically check if there is no owner, if so keep going, otherwise
     * add self to queue and let scheduler run next. */
    disable_interrupts(); /* ATOMICALLY { */
    if (!mp->owned) {
        mp->owned = 1;
        mp->owner_tid = get_running_tid();
        enable_interrupts(); /* } */
        goto mutex_exit;
    }
    log("Waiting on lock %p. mp->owned %d, mp->owner_tid %d",
            mp, mp->owned, mp->owner_tid);

    enable_interrupts();

    assert(yield_execution(BLOCKED, -1, store_tcb_in_mutex_queue, mp) == 0);

mutex_exit:
    assert(mp->owned);
    assert(mp->owner_tid == get_running_tid());
}

/**
 * Switch safe means its safe to call this during a context switch.
 * To ensure this is the case, we do not disable/enable interrupts
 * nor do we call another context switch (through make_thread_runnable) */
static void
mutex_unlock_helper( mutex_t *mp, int switch_safe )
{
    /* Ensure lock is valid, locked and owned by this thread*/
    assert(mp && mp->initialized);
    assert(mp->owned);
    assert(mp->owner_tid == get_running_tid());

    /* If scheduler is not initialized we must have a single
     * thread, so no one to make runnable. */
    if (!is_scheduler_init()) {
        mp->owned = 0;
        return;
    }

    /* Atomically check if someone is in waiters queue, if so,
     * add them to run queue. */

    if (!switch_safe)
        disable_interrupts();

    tcb_t *to_run;
    if ((to_run = Q_GET_FRONT(&mp->waiters_queue))) {
        Q_REMOVE(&mp->waiters_queue, to_run, scheduler_queue);
        mp->owner_tid = to_run->tid;
        if (switch_safe) {
            switch_safe_make_thread_runnable(to_run->tid);
        } else {
            enable_interrupts();
            make_thread_runnable(to_run->tid);
        }
    } else {
        mp->owned = 0;
    }
```

```
    if (!switch_safe)
        enable_interrupts();
}

/** @brief Unlock mutex.
 *  @param mp Mutex to unlock. Has to be previously locked
 *            by calling thread
 *
 *  @return Void
 *  */
void
mutex_unlock( mutex_t *mp )
{
    mutex_unlock_helper(mp, 0);
}

void
switch_safe_mutex_unlock( mutex_t *mp )
{
    mutex_unlock_helper(mp, 1);
}

static void
store_tcb_in_mutex_queue( tcb_t *tcb, void *data )
{
    affirm(tcb && data && tcb->status == BLOCKED);
    /* Since thread not running, might as well use the scheduler queue link! */
    mutex_t *mp = (mutex_t *)data;
    Q_INSERT_TAIL(&mp->waiters_queue, tcb, scheduler_queue);
}
```

```
#ifndef MUTEX_H_
#define MUTEX_H_

#include <scheduler.h> /* queue_t */

/* TODO: Move to internal .h file? */
struct mutex {
    queue_t waiters_queue;
    int initialized;
    int owner_tid;
    int owned;
};

typedef struct mutex mutex_t;

int mutex_init( mutex_t *mp );
void mutex_destroy( mutex_t *mp );
void mutex_lock( mutex_t *mp );
void mutex_unlock( mutex_t *mp );
void switch_safe_mutex_unlock( mutex_t *mp );

#endif /* MUTEX_H_ */
```

```
/** @file sleep.c
 *  @brief Sleep interrupt handler and facilities for managing
 *         sleeping threads
 */
#include <asm.h>                /* outb() */
#include <limits.h>             /* UINT_MAX */
#include <assert.h>             /* affirm */
#include <scheduler.h>          /* get_running_thread(), queue_t */
#include <timer_driver.h>       /* get_total_ticks() */
#include <install_handler.h>    /* install_handler_in_idt() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <task_manager_internal.h> /* Q MACROS on tcb */
#include <lib_thread_management/mutex.h>    /* mutex_t */

/* Using linked list as a naive priority queue implementation.
 * TODO: Use a heap instead! (or something else, like fibonnaci heaps...)
 * */

/** @brief Mux for sleep queue and earlies expiry date */
static mutex_t sleep_mux;

static unsigned int earliest_expiry_date;

/* Queue for all sleeping threads. Sorted by sleep expiration date */
static queue_t sleep_q;

/* 0 if uninitialized, 1 if initialized, -1 if failed to initialize */
static int sleep_initialized = 0;

static void store_tcb_in_sleep_queue( tcb_t *tcb, void *data );

static void
init_sleep( void )
{
    affirm(!sleep_initialized);
    Q_INIT_HEAD(&sleep_q);
    mutex_init(&sleep_mux);
    earliest_expiry_date = UINT_MAX;
    sleep_initialized = 1;
}

/** @brief Tick handler for sleep module. Wakes up any threads
 *         which have slept long enough.
 *
 *  To avoid hurting preemptibility too much, we keep track of
 *  the earliest expiry date and go over the list only if that
 *  date has been reached.
 *
 *  NOTE: If this does too much work we will hurt preemptibility,
 *  as this will eat up time of the thread being context switched to.
 *  Optionally set a limit on how many threads we wake up each given
 *  tick. */
void
sleep_on_tick( unsigned int total_ticks )
{
    if (!sleep_initialized)
        init_sleep();

    mutex_lock(&sleep_mux);

    if (total_ticks < earliest_expiry_date) {
        mutex_unlock(&sleep_mux);
        return;
    }

    /* Reset earliest_expiry_date since we will
     * remove earliest expired thread(s) */
    earliest_expiry_date = UINT_MAX;

    // FIXME: what if we context swap and someone calls
    // sleep, therefore modifying the sleep_q halfway
    // through our read
    tcb_t *curr = Q_GET_FRONT(&sleep_q);
    tcb_t *next;
    while (curr) {
        /* After curr is removed from the sleep queue and is made runnable, we
         * can no longer safely operated on its scheduler_queue link. As such,
         * we must get the next member of the queue before making it runnable.*/
        next = Q_GET_NEXT(curr, scheduler_queue);
        if (curr->sleep_expiry_date <= total_ticks) {
            Q_REMOVE(&sleep_q, curr, scheduler_queue);
            make_thread_runnable(curr->tid);
        } else {
            if (curr->sleep_expiry_date < earliest_expiry_date)
                earliest_expiry_date = curr->sleep_expiry_date;
        }
        curr = next;
    }

    mutex_unlock(&sleep_mux);
}

int
sleep( int ticks )
{
    /* Check for initialization before acknowledging interrupt
     * to avoid race conditions among multiple sleep calls */
    if (!sleep_initialized)
        init_sleep();

    /* Acknowledge interrupt */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    if (ticks < 0)
        return -1;

    if (ticks == 0)
        return 0;

    /* No race condition as only sleep manages sleep_expiry_date */
    tcb_t *me = get_running_thread();
    me->sleep_expiry_date = get_total_ticks() + ticks;

    /* TODO:
     * We could lock the sleep_q mux here and release it through the callback?
     * This should ensure no conflicts with the sleep q stuff */
    mutex_lock(&sleep_mux);
    affirm(yield_execution(BLOCKED, -1, store_tcb_in_sleep_queue, NULL) == 0);

    return 0;
}

/* Modifications to queue are not guarded by mux since they are done
 * atomically (disable/enable interrupts) inside of yield_execution */
static void
store_tcb_in_sleep_queue( tcb_t *tcb, void *data )
{
    affirm(tcb && tcb->status == BLOCKED);
    if (tcb->sleep_expiry_date < earliest_expiry_date)
        earliest_expiry_date = tcb->sleep_expiry_date;
    /* Since thread not running, might as well use the scheduler queue link! */
    Q_INIT_ELEM(tcb, scheduler_queue);
    Q_INSERT_TAIL(&sleep_q, tcb, scheduler_queue);
    switch_safe_mutex_unlock(&sleep_mux);
```

```
}
```

```
#ifndef SLEEP_H_
#define SLEEP_H_

void sleep_on_tick( unsigned int total_ticks );

#endif /* SLEEP_H_ */
```

```
#ifndef THREAD_MANAGEMENT_H_
#define THREAD_MANAGEMENT_H_

int _deschedule( int *reject );

#endif
```

```
/** @file yield.c
 *  @brief Contains yield interrupt handler and helper functions for
 *         installation
 *
 */

#include <asm.h>                /* outb() */
#include <scheduler.h>          /* yield_execution() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */

int
yield( int tid )
{
    /* Acknowledge interrupt immediately */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    return yield_execution(RUNNABLE, tid, NULL, NULL);
}
```

```
/**
 * The 15-410 kernel project.
 * @name loader.c
 *
 * Functions for the loading
 * of user programs from binary
 * files should be written in
 * this file. The function
 * elf_load_helper() is provided
 * for your use.
 *
 * The loader should never interact directly with
 * virtual memory. Rather it should call functions
 * defined in the process mananger module (which itself
 * will be responsible for talking to the VM module).
 *
 */
/*@{*/
/* --- Includes --- */
#include <x86/cr.h>     /* {get,set}_{cr0,cr3} */
#include <loader.h>
#include <malloc.h>     /* sfree() */
#include <page.h>       /* PAGE_SIZE */
#include <string.h>     /* strncmp, memcpy */
#include <exec2obj.h>   /* exec2obj_TOC */
#include <elf_410.h>    /* simple_elf_t, elf_load_helper */
#include <stdint.h>     /* UINT32_MAX */
#include <task_manager.h>   /* task_new, task_prepare, task_set, STACK_ALIGNED*/
#include <memory_manager.h> /* {disable,enable}_write_protection */
#include <logger.h>     /* log_warn() */
#include <scheduler.h> /* get_running_tid() */
#include <assert.h> /* assert() */
#include <simics.h>
#include <x86/asm.h> /* enable_interrupts(), disable_interrupts() */

#include <task_manager_internal.h>

/* --- Local function prototypes --- */

/* first_task is 1 if there is currently no user task running, so
 * execute_user_program will do some initialization
 */
static int first_task = 1;

/* TODO: Move this to a helper file.
 *
 * Having a helper means we evaluate the arguments before expanding _MIN
 *  and therefore avoid evaluating A and B multiple times. */
#define _MIN(A, B) ((A) < (B) ? (A) : (B))
#define MIN(A,B) _MIN(A,B)

/** Copies data from a file into a buffer.
 *
 *  @param filename   the name of the file to copy data from
 *  @param offset     the location in the file to begin copying from
 *  @param size       the number of bytes to be copied
 *  @param buf        the buffer to copy the data into
 *
 * @return number of bytes copied on success. Negative value on failure.
 */
int
getbytes( const char *filename, int offset, int size, char *buf )
{
    if (size == 0)
        return 0; /* Nothing to copy*/

    if (!filename || !buf || offset < 0 || size < 0) {
        log_warn("Loader [getbytes]: Invalid arguments.");
        return -1;
    }

    /* Find file in TOC */
    int i;
    for (i=0; i < exec2obj_userapp_count; ++i) {
        if (strncmp(filename, exec2obj_userapp_TOC[i].execname, MAX_EXECNAME_LEN) =
= 0) {
            break;
        }
    }

    if (i == exec2obj_userapp_count) {
        log_warn("Loader [getbytes]: Executable not found");
        return -1;
    }

    if (offset > exec2obj_userapp_TOC[i].execlen) {
        log_warn("Loader [getbytes]: Offset (%d) is greater than executable "
                 "size (%d)", offset, exec2obj_userapp_TOC[i].execlen);
        return -1;
    }

    int bytes_to_copy = MIN(size, exec2obj_userapp_TOC[i].execlen - offset);

    memcpy(buf, exec2obj_userapp_TOC[i].execbytes + offset, bytes_to_copy);

    return bytes_to_copy;
}

static void
zero_out_memory_region( uint32_t start, uint32_t len )
{
    uint32_t align_end = ((start + len + PAGE_SIZE - 1) / PAGE_SIZE) * PAGE_SIZE;
    memset((void *)start, 0, align_end - start);
}

/** @brief Transplants program data into virtual memory.
 *  Assumes paging is enabled and that virtual memory
 *  has been setup.
 *
 *  @param se_hdr Elf header
 *
 *  @return 0 on sucess, negative value on failure.
 *  */
static int
transplant_program_memory( simple_elf_t *se_hdr )
{
    /* TODO: Swap cr0 on context switch, for now just disabling
     * interrupts. */
    disable_interrupts();

    /* Disable write-protection temporarily so we may
     * copy data into read-only regions. */
    disable_write_protection();

    // FIXME: This error checking is kinda hacky
    int i = 0;

    /* Zero out bytes between memory regions (as well as inside them)  */
    zero_out_memory_region(se_hdr->e_txtstart, se_hdr->e_txtlen);
    zero_out_memory_region(se_hdr->e_rodatstart, se_hdr->e_rodatlen);
    zero_out_memory_region(se_hdr->e_datstart, se_hdr->e_datlen);
    zero_out_memory_region(se_hdr->e_bssstart, se_hdr->e_bsslen);

    /* We rely on the fact that virtual-memory is
```

```
     * enabled to "transplant" program data. Notice
     * that this is only possible because program data is
     * resident on kernel memory which is direct-mapped. */
    i += getbytes(se_hdr->e_fname,
            (unsigned int) se_hdr->e_txtoff,
            (unsigned int) se_hdr->e_txtlen,
            (char *) se_hdr->e_txtstart);
    i += getbytes(se_hdr->e_fname,
            (unsigned int) se_hdr->e_rodatoff,
            (unsigned int) se_hdr->e_rodatlen,
            (char *) se_hdr->e_rodatstart);
    i += getbytes(se_hdr->e_fname,
            (unsigned int) se_hdr->e_datoff,
            (unsigned int) se_hdr->e_datlen,
            (char *) se_hdr->e_datstart);

    /* Re-enable write-protection bit. */
    enable_write_protection();

    enable_interrupts();

    assert(is_valid_pd((void *)get_cr3()));
    return i;
}

/** @brief Puts arguments on stack with format required by _main entrypoint.
 *
 *  This entrypoint is defined in 410user/crt0.c and is used by all user
 *  programs.
 *
 *  Requires that argc and argv are validated
 */
//TODO does not set up stack properly for main
//void _main(int argc, char *argv[], void *stack_high, void *stack_low)
//
static uint32_t *
configure_stack( int argc, char **argv )
{
    /* Set highest addressable byte on stack */
    char *stack_high = (char *) UINT32_MAX;
    assert((uint32_t) stack_high == 0xFFFFFFFF);

    /* Set lowest addressable byte on stack */
    char *stack_low = stack_high - PAGE_SIZE + 1;
    assert(PAGE_ALIGNED(stack_low));

    char *esp_char = stack_high - sizeof(uint32_t) + 1;

    /* Transfer argv onto user stack */
    char *user_stack_argv[argc];
    memset(user_stack_argv, 0, argc * sizeof(char *));

    /* Put the char * onto the user stack */
    for (int i = 0; i < argc; ++i) {
        esp_char -= USER_STR_LEN;
        log("string of argv at address:%p", esp_char);
        assert(STACK_ALIGNED(esp_char));
        memset(esp_char, 0, USER_STR_LEN);

        affirm(esp_char);
        affirm(argv);
        affirm(argv[argc - 1 - i]);

        memcpy(esp_char, argv[argc - 1 - i], strlen(argv[argc - 1 - i]));
        user_stack_argv[argc - 1 - i] = esp_char;
    }
    uint32_t *esp = (uint32_t *) esp_char;

    /* Put the null terminated char ** onto the user stack */
    *(--esp) = 0;
    for (int i = 0; i < argc; ++i) {
        *(--esp) = (uint32_t) user_stack_argv[argc - 1 - i];
    }

    /* Save value of char **argv */
    char **argv_arg = (char **) esp;

    /* Store arguments on stack */
    *(--esp) = (uint32_t) stack_low;
    *(--esp) = (uint32_t) stack_high;
    *(--esp) = (uint32_t) argv_arg;
    *(--esp) = argc;

    /* Functions expect esp to point to return address on entry.
     * Therefore we just point it to some garbage, since _main
     * is never supposed to return. */
    esp--;

    log("stack top:%p", esp);
    log("argc:%d", *(esp + 1));
    log("argv[0]:%s", (char *) **((char ***)esp + 2));
    log("stack_hi:%p", (char *) *(esp + 3));
    log("stack_lo:%p", (char *) *(esp + 4));

    return esp;




    ///* TODO: Consider writing this with asm, as it might be simpler. */

    ///* TODO: In the future, when "receiver" function is implemented, loader
    // * should also add entry point, user registers and data segment selectors
    // * on the stack. For registers, just initialize most to 0 or something. */

    ///* As a pointer to uint32_t, it must point to the lowest address of
    // * the value. */
    //uint32_t *esp = (uint32_t *)(UINT32_MAX - (sizeof(uint32_t) - 1));
    //assert((uint32_t) esp % 4 == 0);

    //*esp = argc;

    //if (argc == 0) {
    //    *(--esp) = 0;
    //  assert((uint32_t) esp % 4 == 0);
    //    return esp;
    //}

    //esp -= argc; /* sizeof(char *) == sizeof(uint32_t *) */
    //assert((uint32_t) esp % 4 == 0);
    //// TODO what if argv has a string
    //memcpy(esp, argv, argc * sizeof(char *)); /* Put argv on stack */
    //esp--;
    //assert((uint32_t) esp % 4 == 0);

    //*(esp--) = UINT32_MAX; /* Put stack_high on stack */
    //assert((uint32_t) esp % 4 == 0);

    //*(esp) = UINT32_MAX - PAGE_SIZE - 1; /* Put stack_low on stack */

    //assert((uint32_t) esp % 4 == 0);
    //return esp;
}
```

```
/** @brief Run a user program indicated by fname.
 *
 *  This function requires no synchronization as it is only
 *  meant to be used to load the starter program (when we have
 *  a single thread) and for the syscall exec(). We disable calls to
 *  exec() when there is more than 1 thread in the invoking task.
 *
 *  TODO don't think this is the best requires
 *  @req that fname and argv are in kernel memory, unaffected by
 *       parent directory
 *
 *  @param fname Name of program to run.
 *  @return 0 on success, negative value on error.
 */
int
execute_user_program( char *fname, int argc, char **argv)
{
    //disable_interrupts();// FIXME why is this here gg
    //if (first_task)
    //  log_warn("Executing first task");
    //else
    //  log_warn("Executing not-first task");

    //log_warn("Executing pointer %s", fname);
    //enable_interrupts();
    log_warn("executing task fname:%s", fname);

    if (!first_task) {
        /* Validate execname */
        if (!is_valid_null_terminated_user_string(fname, USER_STR_LEN)) {
            return -1;
        }
        //assert(is_valid_pd(get_tcb_pd(get_running_thread())));
        /* Validate argvec */
        int argc = 0;
        if (!(argc = is_valid_user_argvec(fname, argv))) {
            return -1;
        }
    }

    /* Transfer execname to kernel stack so unaffected by page directory */
    char kern_stack_execname[USER_STR_LEN];
    memset(kern_stack_execname, 0, USER_STR_LEN);
    memcpy(kern_stack_execname, fname, strlen(fname));

    /* char array to store each argvec string on kernel stack */
    char *kern_stack_args = smalloc(NUM_USER_ARGS * USER_STR_LEN);
    if (!kern_stack_args) {
        return -1;
    }
    memset(kern_stack_args, 0, NUM_USER_ARGS * USER_STR_LEN);

    /* char * array for argvec on kernel stack */
    char *kern_stack_argvec[NUM_USER_ARGS];
    memset(kern_stack_argvec, 0, NUM_USER_ARGS);

    /* Transfer argvec to kernel memory so unaffected by page directory */
    int offset = 0;
    for (int i = 0; argv[i]; ++i) {
        char *arg = argv[i];
        memcpy(kern_stack_args + offset, arg, strlen(arg));
        kern_stack_argvec[i] = kern_stack_args + offset;
        offset += USER_STR_LEN;
    }
    /* Load user program information */
    simple_elf_t se_hdr;
    if (elf_check_header(kern_stack_execname) == ELF_NOTELF) {
        return -1;
    }
    if (elf_load_helper(&se_hdr, kern_stack_execname) == ELF_NOTELF) {
        return -1;
    }
    uint32_t pid, tid;

    /* First task, so create a new task */
    if (first_task) {
        if (create_task(&pid, &tid, &se_hdr) < 0)
            return -1;

    /* Not the first task, so we replace the current running task */
    } else {
        pid = get_pid();
        tid = get_running_tid();

        /* Create new pd */
        uint32_t stack_lo = UINT32_MAX - PAGE_SIZE + 1;
        uint32_t stack_len = PAGE_SIZE;
        void *new_pd = new_pd_from_elf(&se_hdr, stack_lo, stack_len);
        if (!new_pd) {
            return -1;
        }
        void *old_pd = swap_task_pd(new_pd);
        //assert(is_valid_pd(old_pd));
        free_pd_memory(old_pd);
        sfree(old_pd, PAGE_SIZE);
    }


#ifdef DEBUG
    tcb_t *tcb = find_tcb(tid);
    assert(tcb);
    /* Register this task's new binary with simics */
    sim_reg_process(get_tcb_pd(tcb), kern_stack_execname);
#endif

    /* Update page directory, enable VM if necessary */
    if (activate_task_memory(pid) < 0) {
        return -1;
    }

    if (transplant_program_memory(&se_hdr) < 0) {
        return -1;
    }
    uint32_t *esp = configure_stack(argc, kern_stack_argvec);

    /* If this is the first task we must activate it */
    if (first_task) {
        assert(is_valid_pd(get_tcb_pd(find_tcb(tid))));
        task_set_active(tid);
    }
    first_task = 0;

    /* Start the task */
    task_start(tid, (uint32_t)esp, se_hdr.e_entry);

    panic("execute_user_program does not return");
    return -1;
}
```

```
/** @file logger.c
 *  @brief 3 levels of logging are implemented
 *
 *  if NDEBUG is set no logs will print
 *  else if NDEBUG is not set, each log will print iff log_level <= that
 *  logging function's priority. We follow Python's convention:
 *
 *  https://docs.python.org/3/howto/logging.html
 *
 *  To be more explicit:
 *  - log() will print whenever log_level <= DEBUG_PRIORITY
 *
 *    Toggle log_level to DEBUG_PRIORITY for debugging
 *    Typically used for diagnosing problems
 *
 *  - log_med() will print whenever log_level <= INFO_PRIORITY
 *
 *    Toggle log_level to INFO_PRIORITY to print info on expected behavior
 *    Confirmation that things are working as expected
 *
 *  - log_hi() will print whenever log_level <= WARN_PRIORITY
 *
 *    Toggle log_level to WARN_PRIORITY to print only warnings on behavior
 *    Indication that something potentially dangerous happened, indicative of
 *    some issue in the near future (e.g. 'low number of free physical frames').
 *    However the kernel is still working as expected.
 *
 */

#include <stdarg.h> /* va_list(), va_end() */
#include <stdio.h> /* snprintf(), vsnprintf() */
#include <simics.h> /* sim_puts() */
#include <scheduler.h> /* get_running_tid() */
#include <logger.h>
#include <assert.h> /* affirm_msg() */

#define LEN 256

/** @brief Prepends the thread id to printed format. Takes in a va_list as
 *         argument.
 *
 *  This function is called by logging functions and panic(), which must
 *  convert their variable number of arguments into a va_list before passing
 *  them to vtprintf().
 *
 *  Passing in an unrecognized priority will just lead to
 *  printing out an error message to let the user know that it has supplied
 *  the wrong arguments, no need to cause an assertion failure as the error
 *  is promptly displayed.
 *
 *  @param format String format to print to
 *  @param args A va_list of all arguments to include in format
 *  @param priority Logging priority.
 *  @return Void.
 */
void
vtprintf( const char *format, va_list args, int priority )
{
    char str[LEN];

    /* Get tid and prepend to output*/
    int tid = get_running_tid();
    int offset = 0;

    switch (priority) {

        case DEBUG_PRIORITY:
            offset = snprintf(str, sizeof(str) - 1, "tid[%d]: DEBUG: ", tid);
            break;

        case INFO_PRIORITY:
            offset = snprintf(str, sizeof(str) - 1, "tid[%d]: INFO: ", tid);
            break;

        case WARN_PRIORITY:
            offset = snprintf(str, sizeof(str) - 1, "tid[%d]: WARN: ", tid);
            break;

        case CRITICAL_PRIORITY:
            offset = snprintf(str, sizeof(str) - 1, "tid[%d]: CRITICAL: ", tid);
            break;

        /* Improper use of vtprintf(), print error */
        default:
            snprintf(str, sizeof(str) - 1,
                     "tid[%d]: UNRECOGNIZED priority:%d for vtprintf()",
                     tid, priority);
            sim_puts(str);
            return;
    }
    /* Print rest of output, and a little extra for CRITICAL priority */
    offset += vsnprintf(str + offset, sizeof(str) - offset - 1, format, args);
    if (priority == CRITICAL_PRIORITY) {
        snprintf(str + offset, sizeof(str) - offset - 1,
                 "\nCrashing the kernel.");

        printf("%s", str);
    }

    sim_puts(str);
    //TODO print on actual hardware

    return;
}

/** @brief prints out log to console and kernel log as long as NDEBUG is not
 *         set. This log has priority DEBUG_PRIORITY
 *
 *  @param format String specifier to print
 *  @param ... Arguments to format string
 *  @return Void.
 */
void
log( const char *format, ... )
{
#ifndef NDEBUG
    if (log_level <= DEBUG_PRIORITY) {

        /* Construct va_list to pass on to vtprintf */
        va_list args;
        va_start(args, format);
        vtprintf(format, args, DEBUG_PRIORITY);
        va_end(args);
    }
#endif
    return;
}

/** @brief prints out log to console and kernel log as long as NDEBUG is not
 *         set. This log has priority INFO_PRIORITY
 *
 *  @param format String specifier to print
 *  @param ... Arguments to format string
 *  @return Void.
```

```
 */
void
log_info( const char *format, ... )
{
#ifndef NDEBUG
    if (log_level <= INFO_PRIORITY) {

        /* Construct va_list to pass on to vtprintf */
        va_list args;
        va_start(args, format);
        vtprintf(format, args, INFO_PRIORITY);
        va_end(args);
    }
#endif
    return;
}

/** @brief prints out log to console and kernel log as long as NDEBUG is not
 *         set. This log has priority WARN_PRIORITY
 *
 *  @param format String specifier to print
 *  @param ... Arguments to format string
 *  @return Void.
 */
void
log_warn( const char *format, ... )
{
#ifndef NDEBUG
    if (log_level <= WARN_PRIORITY) {
        /* Construct va_list to pass on to vtprintf */
        va_list args;
        va_start(args, format);
        vtprintf(format, args, WARN_PRIORITY);
        va_end(args);
    }
#endif
    return;
}

/** @brief prints out log to console and kernel log regardless of whether
 *         NDEBUG is set since this log has priority CRITICAL_PRIORITY
 *         which means that an unrecoverable failure in the kernel has occurred.
 *
 *  @pre Callers of log_crit must intend to crash the kernel
 *  @param format String specifier to print
 *  @param ... Arguments to format string
 *  @return Void.
 */
void
log_crit( const char *format, ... )
{
    if (log_level <= CRITICAL_PRIORITY) {

        /* Construct va_list to pass on to vtprintf */
        va_list args;
        va_start(args, format);
        vtprintf(format, args, CRITICAL_PRIORITY);
        va_end(args);
    }
}
```

```
/** @file logger.h
 *  @brief Functions that enable code tracing via logging
 *
 *  @author Nicklaus Choo (nchoo)
 */

#include <stdarg.h> /* va_list() */

#ifndef _LOGGER_H_
#define _LOGGER_H_

#define DEBUG_PRIORITY 1
#define INFO_PRIORITY 2
#define WARN_PRIORITY 3
#define CRITICAL_PRIORITY 4

/* Current logging level defined in kernel.c */
extern int log_level;

/* Function prototypes */
void vtprintf( const char *format, va_list args, int priority );
void log( const char *format, ... );
void log_info( const char *format, ...);
void log_warn( const char *format, ...);
void log_crit( const char *format, ...);

#endif /* _LOGGER_H_ */
```

```
#include <malloc.h>
#include <assert.h>             /* affirm */
#include <stddef.h>             /* size_t */
#include <malloc_internal.h>    /* _malloc family of functions */
#include <lib_thread_management/mutex.h> /* mutex_t */
#include <logger.h>

/* Where to put this mutex? We could initialize it in scheduler init? */
static mutex_t malloc_mux;
static int is_mutex_init = 0;

/* These macros allow us to easily initialize the malloc mutex
 * upon the first call to the malloc library. */
#define LOCK do\
{\
    if (!is_mutex_init) {\
        mutex_init(&malloc_mux);\
        is_mutex_init = 1;\
    }\
    mutex_lock(&malloc_mux);\
} while(0)\

#define UNLOCK do\
{\
    affirm(is_mutex_init);\
    mutex_unlock(&malloc_mux);\
} while(0)\


/* safe versions of malloc functions */
void *malloc(size_t size)
{
    LOCK;
    void *p = _malloc(size);
    log("malloc returned %p", p);
    UNLOCK;
    return p;
}

void *memalign(size_t alignment, size_t size)
{
    LOCK;
    void *p = _memalign(alignment, size);
    log("memalign returned %p, size 0x%08lx", p, size);
    UNLOCK;
    return p;
}

void *calloc(size_t nelt, size_t eltsize)
{
    LOCK;
    void *p = _calloc(nelt, eltsize);
    UNLOCK;
    return p;
}

void *realloc(void *buf, size_t new_size)
{
    LOCK;
    void *p = _realloc(buf, new_size);
    UNLOCK;
    return p;
}

void free(void *buf)
{
    LOCK;
    _free(buf);
    UNLOCK;
}

void *smalloc(size_t size)
{
    LOCK;
    void *p = _smalloc(size);
    log("smalloc returned %p, size 0x%08lx", p, size);

    UNLOCK;
    return p;
}

void *smemalign(size_t alignment, size_t size)
{
    LOCK;
    void *p = _smemalign(alignment, size);
    log("smemalign returned %p, size 0x%08lx", p, size);

    UNLOCK;
    return p;
}

void sfree(void *buf, size_t size)
{
    LOCK;
    _sfree(buf, size);
    log("sfree(): freed %p, size 0x%08lx", buf, size);

    UNLOCK;
}
```

```
/** @file memory_manager.c
 *  @brief Functions to initialize and manage virtual memory
 *
 *  Note that page directory and page table consistency/validity checks are
 *  expensive and thus carried out in assertions.
 *
 *  TODO need to figure out when to free physical pages, probably when
 *  cleaning up thread resources ?
 *
 *  @author Andre Nascimento (anascime)
 *  @author Nicklaus Choo (nchoo)
 *
 * */

#include <simics.h>
#include <physalloc.h> /* physalloc() */
#include <memory_manager.h>
#include <stdint.h>     /* uint32_t */
#include <stddef.h>     /* NULL */
#include <malloc.h>     /* smemalign, sfree */
#include <elf_410.h>    /* simple_elf_t */
#include <assert.h>     /* assert, affirm */
#include <page.h>       /* PAGE_SIZE */
#include <x86/cr.h>     /* {get,set}_{cr0,cr3} */
#include <string.h>     /* memset, memcpy */
#include <common_kern.h>/* USER_MEM_START */
#include <logger.h>     /* log */
#include <memory_manager_internal.h>

// TODO: delete
#include <asm.h>

/* 1 if VM is enabled, 0 otherwise */
static int paging_enabled = 0;
static uint32_t sys_zero_frame = USER_MEM_START;

static int allocate_frame( uint32_t **pd, uint32_t virtual_address,
                           write_mode_t write_mode, uint32_t sys_prog_flag );
static int allocate_region( uint32_t **pd, void *start, uint32_t len,
                            write_mode_t write_mode );
static void enable_paging( void );

static int valid_memory_regions( simple_elf_t *elf );
static void vm_set_pd( void *pd );
static void free_pt_memory( uint32_t *pt, int pd_index );

static void *allocate_new_pd( void );
static int add_new_pt_to_pd( uint32_t **pd, uint32_t virtual_address );

void
unallocate_frame( uint32_t **pd, uint32_t virtual_address )
{
    uint32_t *ptep = get_ptep((const uint32_t **) pd, virtual_address);
    affirm_msg(ptep, "unallocate_frame(): "
               "cannot free non existent page table, "
               "pd:%p, "
               "virtual_address:0x%08lx",
               pd, virtual_address);
    uint32_t pt_entry = *ptep;

    affirm(pt_entry & PRESENT_FLAG);

    uint32_t phys_address = TABLE_ADDRESS(pt_entry);
    physfree(phys_address);

    // Zero the entry as well
    *ptep = 0;
}


/** @brief Returns pointer to page directory from cr3(), guarantees that pointer
 *         is non-NULL and page aligned, and below USER_MEM_START
 *
 *  Does consistency check for valid page directory if NDEBUG is not defined.
 *
 *  @return Pointer to current active page directory
 */
void *
get_pd( void )
{
    void *pd = (void *) TABLE_ADDRESS(get_cr3());

    /* Basic checks for non-NULL, page aligned and < USER_MEM_START */
    affirm_msg(pd, "unable to get page directory");
    affirm_msg(PAGE_ALIGNED(pd), "page directory not page aligned!");
    affirm_msg((uint32_t) pd < USER_MEM_START,
               "page directory > USER_MEM_START");

    /* Expensive check, hence the assertion */
    assert(is_valid_pd(pd));
    return pd;
}

int
zero_page_pf_handler( uint32_t faulting_address )
{
    /* get_pd() guarantees basic consistency for valid page directory */
    uint32_t **pd = get_pd();

    uint32_t *ptep = get_ptep( (const uint32_t **) pd, faulting_address);

    /* Page table entry cannot be NULL frame */
    if (!ptep) {
        log_warn("zero_page_pf_handler(): "
                 "page table entry for vm 0x%08lx is NULL!",
                 faulting_address);
        return -1;
    }
    uint32_t pt_entry = *ptep;

    /* Page table entry must hold the system wide zero frame */
    if (TABLE_ADDRESS(pt_entry) != sys_zero_frame) {
        log_warn("zero_page_pf_handler(): "
                 "page table entry for vm 0x%08lx is not zero frame",
                 faulting_address);
        return -1;
    }
    /* Page table entry must be user readable since sys wide zero frame */
    assert((pt_entry & PE_USER_READABLE) == PE_USER_READABLE);

    /* Get sys_prog_flag */
    uint32_t sys_prog_flag = SYS_PROG_FLAG(pt_entry);
    assert(is_valid_sys_prog_flag(sys_prog_flag));

    /* Unallocate zero frame */
    unallocate_user_zero_frame(pd, faulting_address);

    /* Back up with actual frame */
    // TODO version of allocate_frame that throws an error if already allocated
    int res = allocate_frame(pd, faulting_address, READ_WRITE, sys_prog_flag);
    if (res) {
        log_warn("zero_page_pf_handler(): "
                 "Failed to allocate frame inside zero_page_pf_handler");
        return -1;
```

```
    }

    /* Flush TLB so we zero out the appropriate physical frame */
    set_cr3(get_cr3());
    log("memsetting faulting_address %p, (table addr %p) to 0.",
            (void *)faulting_address,(void *)TABLE_ADDRESS(faulting_address));
    memset((void *)TABLE_ADDRESS(faulting_address), 0, PAGE_SIZE);

    return 0;
}

void
initialize_zero_frame( void )
{
    affirm(!paging_enabled);

    /* Zero fill system wide zero frame */
    memset((uint32_t *) sys_zero_frame, 0, PAGE_SIZE);
}

/** @brief Sets up a new page directory by allocating physical memory for it.
 *         Does not transfer executable data into physical memory.
 *
 *  Assumes page table directory is empty. Sets appropriate
 *  read/write permissions. To copy memory over, set the WP flag
 *  in the CR0 register to 0 - this will make it so that write
 *  protection is ignored by the paging mechanism.
 *
 *  Allocated pages are initialized to 0.
 *
 *  TODO: Implement ZFOD here. (Handler should probably be defined
 *  elsewhere, though)
 *
 *  @return Page directory that is backed by physical memory
 *  */
void *
new_pd_from_elf( simple_elf_t *elf, uint32_t stack_lo, uint32_t stack_len )
{
    /* Allocate new pd that is zero filled */
    uint32_t **pd = allocate_new_pd();
    if (!pd) {
        log_warn("new_pd_from_elf(): "
                 "unable to allocate new page directory.");
        return NULL;
    }

    /* If there is a current page directory, the bottom 4 must be kernel mapped
     * so use those instead
     */
    if (get_cr3()) {
        uint32_t ** current_pd = (uint32_t **)(TABLE_ADDRESS(get_cr3()));
        for (int i = 0; i < NUM_KERN_PAGE_TABLES; ++i) {
            pd[i] = current_pd[i];
        }
    } else {

    /* Direct map all 16MB for kernel, setting correct permission bits */
    for (uint32_t addr = 0; addr < USER_MEM_START; addr += PAGE_SIZE) {

            /* This invariant must not break */
            uint32_t pd_index = PD_INDEX(addr);
            uint32_t *pd_entry = pd[pd_index];
            affirm_msg(TABLE_ENTRY_INVARIANT(pd_entry),
                       "new_pd_from_elf(): "
                       "pd entry invariant broken for "
                       "pd:%p pd_index:0x%08lx pd[pd_index]:%p",
                       pd, pd_index, pd_entry);

            /* Add new page table every time page directory entry is NULL */
            if (pd[pd_index] == NULL) {

                /* Since we are going in increasing virtual addresses, holds */
                assert((addr & ((1 << PAGE_DIRECTORY_SHIFT) - 1)) == 0);
                if (add_new_pt_to_pd(pd, addr) < 0) {
                    log_warn("new_pd_from_elf(): "
                             "unable to allocate new page table in pd:%p for "
                             "virtual_address: 0x%08lx", pd, addr);
                    return NULL;
                    //TODO clean up
                }
            }
            /* Now get a pointer to the corresponding page table entry */
            uint32_t *ptep = get_ptep((const uint32_t **) pd, addr);

            /* If NULL is returned, free all resources in page directory */
            if (!ptep) {
                free_pd_memory(pd);
                sfree(pd, PAGE_SIZE);
                log_warn("new_pd_from_elf(): "
                         "unable to get page table entry pointer.");
                return NULL;
            }
            /* Indicate page table entry permissions */
            if (addr == 0) {
                *ptep = addr | PE_UNMAPPED; /* Leave NULL unmapped. */
            } else {
                *ptep = addr | PE_KERN_WRITABLE;
            }
            assert(*ptep < USER_MEM_START);
        }
    }
    log("new_pd_from_elf(): direct map ended");
    /* Allocate regions with appropriate read/write permissions.
     * TODO: Free allocated regions if later allocation fails. */
    int i = 0;

    // TODO: implement valid_memory_regions
    if (!valid_memory_regions(elf)) {
        free_pd_memory(pd);
        sfree(pd, PAGE_SIZE);
        return NULL;
    }

    i += allocate_region(pd, (void *)elf->e_txtstart, elf->e_txtlen, READ_ONLY);
    i += allocate_region(pd, (void *)elf->e_datstart, elf->e_datlen, READ_WRITE);
    i += allocate_region(pd, (void *)elf->e_rodatstart, elf->e_rodatlen, READ_ONLY)
;
    i += allocate_region(pd, (void *)elf->e_bssstart, elf->e_bsslen, READ_WRITE);
    i += allocate_region(pd, (void *)stack_lo, stack_len, READ_WRITE);

    if (i < 0) {
        sfree(pd, PAGE_SIZE);
        return NULL;
    }

    assert(is_valid_pd(pd));
    return pd;
}

/** @brief Initialized child pd from parent pd. Deep copies writable
 *         entries, allocating new physical frame. Returns child_pd on success
 *
 *         TODO: Revisit to implement ZFOD. Also, avoid flushing tlb
 *         all the time.
```

```
 * */
void *
new_pd_from_parent( void *v_parent_pd )
{
    uint32_t *parent_pd = (uint32_t *)v_parent_pd;
    /* Create temp_buf for deep copy. Too large to stack allocate */
    uint32_t *child_pd = smemalign(PAGE_SIZE, PAGE_SIZE);
    if (!child_pd) {
        return NULL;
    }
    assert(PAGE_ALIGNED(child_pd));
    memset(child_pd, 0, PAGE_SIZE); /* Set all entries to empty initially */

    uint32_t *temp_buf = smemalign(PAGE_SIZE, PAGE_SIZE);
    if (!temp_buf) {
        sfree(child_pd, PAGE_SIZE);
        return NULL;
    }

    /* Just shallow copy kern memory page tables */
    for (int i=0; i < (PAGE_SIZE / sizeof(uint32_t)); ++i) {
        if (i < 4) {
            child_pd[i] = parent_pd[i];
            continue;
        }

        if (parent_pd[i] & PRESENT_FLAG) {
            /* Allocate new child page_table */
            uint32_t *child_pt = smemalign(PAGE_SIZE, PAGE_SIZE);
            if (!child_pt) {
                free_pd_memory(child_pd); // Cleanup previous allocs
                sfree(temp_buf, PAGE_SIZE);
                sfree(child_pd, PAGE_SIZE);
                return NULL;
            }
            memset(child_pt, 0, PAGE_SIZE);
            assert(PAGE_ALIGNED(child_pt));
            log("child_pt:%p", child_pt);

            // update child_pd[i]
            child_pd[i] = (uint32_t) child_pt;
            // OR the flags
            child_pd[i] |= (parent_pd[i] & (PAGE_SIZE - 1));

            // Get address of parent_pt
            uint32_t *parent_pt = (uint32_t *)(parent_pd[i] & ~(PAGE_SIZE - 1));
            assert(PAGE_ALIGNED(parent_pt));

            // parent_pt and child_pt are actual addresses without flags

            /* Copy entries in page tables */
            for (int j=0; j < (PAGE_SIZE / sizeof(uint32_t)); ++j) {

                // Constructing the virtual address
                uint32_t vm_address = ((i << 22) | (j << 12));
                assert(PAGE_ALIGNED(vm_address));
                assert(vm_address >= USER_MEM_START);

                if (parent_pt[j] & PRESENT_FLAG) {
                    /* Allocate new physical frame for child. */
                    child_pt[j] = physalloc();
                    assert(PAGE_ALIGNED(child_pt[j]));
                    if (!child_pt[j]) {
                        free_pd_memory(child_pd); // Cleanup previous allocs

                        sfree(temp_buf, PAGE_SIZE);
                        sfree(child_pd, PAGE_SIZE);
                        return NULL;
                    }

                    /* Mark child_pt[j] as writable, copy and then mark
                     * same flags as the parent (in case READ-ONLY) */
                    // Mark as user writable to avoid creating a global TLB entry
                    child_pt[j] |= PE_USER_WRITABLE;

                    log("vm_address:%lx, i:0x%x, j:0x%x, "
                    "parent_pd[i]:0x%lx, child_pd[i]:0x%lx, "
                    "parent_pt[j]:0x%lx, child_pt[j]:0x%lx",
                    vm_address, i, j,
                    parent_pd[i], child_pd[i],
                    parent_pt[j], child_pt[j]);

                    /* Copy parent to temp, change page-directory,
                     * copy child to parent, restore parent page-directory */

                    memcpy(temp_buf, (uint32_t *) vm_address, PAGE_SIZE);
                    vm_set_pd(child_pd);
                    memcpy((uint32_t *) vm_address, temp_buf, PAGE_SIZE);
                    vm_set_pd(parent_pd);


                    // Zero out flags
                    child_pt[j] &= ~(PAGE_SIZE - 1);
                    // Set parent flags
                    child_pt[j] |= parent_pt[j] & (PAGE_SIZE - 1);

                } else {
                    assert(parent_pt[j] == 0);
                }
            }
        } else {
            assert(parent_pd[i] == 0);
        }
    }

    sfree(temp_buf, PAGE_SIZE);
    assert(is_valid_pd(child_pd));
    return child_pd;
}

/** @brief Sets new page table directory and enables paging if necessary.
 *
 *  Paging should only be set once and never disabled.
 *
 *  @param pd Page directory pointer
 *  @return Void.
 */
void
vm_enable_task( void *pd )
{
    affirm_msg(pd, "Page directory must be non-NULL!");
    affirm_msg(PAGE_ALIGNED(pd), "Page directory must be page aligned!");
    affirm_msg((uint32_t) pd < USER_MEM_START,
                "Page directory must in kernel memory!");

    vm_set_pd(pd);
    if (!paging_enabled) {
        enable_paging();

        /* Set PGE flag in cr4 so kernel mappings not flushed on context switch */
        set_cr4(CR4_PGE | get_cr4());
    }
}
```

```
/** @brief Enables write_protect flag in cr0, allowing
 *  kernel to bypass VM's read-only protection. */
void
enable_write_protection( void )
{
    uint32_t current_cr0 = get_cr0();
    set_cr0(current_cr0 | WRITE_PROTECT_FLAG);
}

/** @brief Disables write_protect flag in cr0, stopping
 *  kernel from bypassing VM's read-only protection. */
void
disable_write_protection( void )
{
    uint32_t current_cr0 = get_cr0();
    set_cr0(current_cr0 & (~WRITE_PROTECT_FLAG));
}


/** @brief Checks if a user pointer is valid.
 *  Valid means the pointer is non-NULL, belongs to
 *  user memory and is in an allocated memory region.
 *
 *  @param ptr Pointer to check
 *  @param read_write Whether to check for write permission
 *  @return 1 if valid, 0 if not */
int
is_valid_user_pointer(void *ptr, write_mode_t write_mode)
{
    /* Check not kern memory and non-NULL */
    if ((uint32_t)ptr < USER_MEM_START)
        return 0;

    /* Check if allocated */
    if (!is_user_pointer_allocated(ptr)) {
        return 0;
    }
    /* Check for correct write_mode */
    uint32_t **pd = (uint32_t **)TABLE_ADDRESS(get_cr3());
    uint32_t pd_index = PD_INDEX(ptr);
    uint32_t pt_index = PT_INDEX(ptr);
    uint32_t *pt = (uint32_t *) TABLE_ADDRESS(pd[pd_index]);
    if (write_mode == READ_WRITE && !(pt[pt_index] & RW_FLAG))
        return 0;


    return 1;
}

int
is_user_pointer_allocated( void *ptr )
{
    uint32_t **pd = (uint32_t **)TABLE_ADDRESS(get_cr3());
    uint32_t pd_index = PD_INDEX(ptr);
    uint32_t pt_index = PT_INDEX(ptr);

    /* Not present in page directory */
    if (!(((uint32_t) pd[pd_index]) & PRESENT_FLAG)) {
        return 0;
    }
    /* Not present in page table */
    uint32_t *pt = (uint32_t *) TABLE_ADDRESS(pd[pd_index]);
    if (!(pt[pt_index] & PRESENT_FLAG)) {
        return 0;
    }
    return 1;
}
```

```
/** @brief Checks that the address of every character in the string is a valid
 *         address
 *
 *  //TODO test this function and see if it catches stuff
 *
 *  The maximum permitted string length is USER_STR_LEN, including '\0'
 *  terminating character. Therefore the longest possible user string will
 *  have at most USER_STR_LEN - 1 non-NULL characters.
 *
 *  This does not check for the existence of a user executable with this
 *  name. That is done when we try to fill in the ELF header.
 *         address. Does not check for write permissions!
 *
 *  @param s String to be checked
 *  @param len Length of string
 *  @param null_terminated  Whether the string should be checked
 *                          for null-termination
 *  @return 1 if valid user string, 0 otherwise
 */
static int
is_valid_user_string_helper( char *s, int len, int null_terminated)
{
    /* Check address of every character in s */
    int i;
    for (i = 0; i < len; ++i) {

        if (!is_valid_user_pointer(s + i, READ_ONLY)) {
            log_warn("invalid address %p at index %d of user string %s",
                     s + i, i, s);
            return 0;

        } else {

            /* String has ended within USER_STR_LEN */
            if (s[i] == '\0') {
                break;
            }
        }
    }
    /* Check length of s */
    if (i == len && null_terminated) {
        log_warn("user string of length >= USER_STR_LEN");
        return 0;
    }
    return 1;
}

/** @brief Checks that the address of every character in the string is a valid
 *         address. Does not check for write permissions!
 *
 *  The maximum permitted string length is max_len, including '\0'
 *  terminating character. Therefore the longest possible user string will
 *  have at most max_len - 1 non-NULL characters.
 *
 *  This does not check for the existence of a user executable with this
 *  name. That is done when we try to fill in the ELF header.
 *
 *  @param s String to be checked
 *  @param len Length of string
 *  @return 1 if valid user string, 0 otherwise
 */
int
is_valid_null_terminated_user_string( char *s, int len )
{
    return is_valid_user_string_helper(s, len, 1);
```

```
}

/** @brief Checks that the address of every character in the string is a valid
 *         address. Does not check for write permissions!
 *
 *  @param s String to be checked
 *  @param len Length of string
 *  @return 1 if valid user string, 0 otherwise
 */
int
is_valid_user_string( char *s, int len )
{
    return is_valid_user_string_helper(s, len, 0);
}

/** @brief Checks address of every char * in argvec, argvec has max length
 *         of < NUM_USER_ARGS
 *
 *  //TODO test this function and see if it catches stuff
 *
 *  @param execname Executable name
 *  @param argvec Argument vector
 *  @return Number of user args if valid argvec, 0 otherwise
 */
int
is_valid_user_argvec( char *execname, char **argvec )
{
    /* Check address of every char * in argvec */
    int i;
    for (i = 0; i < NUM_USER_ARGS; ++i) {

        /* Invalid char ** */
        if (!is_valid_user_pointer(argvec + i, READ_ONLY)) {
            log_warn("invalid address %p at index %d of argvec", argvec + i, i);
            return 0;

        /* Valid char **, so check if char * is valid */
        } else {

            /* String has ended within NUM_USER_ARGS */
            if (argvec[i] == NULL) {
                break;
            }
            /* Check if valid string */
            if (!is_valid_null_terminated_user_string(argvec[i],
                        USER_STR_LEN)) {
                log_warn("invalid address user string %s at index %d of argvec",
                         argvec[i], i);
                return 0;
            }
        }
    }
    /* Check length of arg_vec */
    if (i == NUM_USER_ARGS) {
        log_warn("argvec has length >= NUM_USER_ARGS");
        return 0;
    }
    /* Check if argvec[0] == execname */
    if (strcmp(argvec[0],execname) != 0) {
        log_warn("argvec[0]:%s not equal to execname:%s", argvec[0], execname);
        return 0;
    }
    return i;
}

/* ----- HELPER FUNCTIONS ----- */

/** @brief Allocate memory for a new page table and zero all entries
 *
 *  Ensures that the allocated page table has an address that is page
 *  aligned
 *
 *  @return Pointer in kernel VM of page table if successful, 0 otherwise
 */
void *
allocate_new_pt( void )
{
    /* Allocate memory for a new page table */
    void *pt = smemalign(PAGE_SIZE, PAGE_SIZE);
    if (!pt) {
        return 0;
    }
    log("new pt at address %p", pt);
    assert(PAGE_ALIGNED((uint32_t) pt));

    /* Initialize all page table entries as non-present */
    memset(pt, 0, PAGE_SIZE);

    return pt;
}

/** @brief Allocate memory for a new page directory and zero all entries
 *
 *  @return Pointer in kernel VM of page directory if successful, NULL otherwise
 */
static void *
allocate_new_pd( void )
{
    /* Allocate memory for a new page directory */
    void *pd = smemalign(PAGE_SIZE, PAGE_SIZE);
    if (!pd) {
        log_warn("allocate_new_pd(): "
                 "unable to allocate new page directory");
        return NULL;
    }
    log("allocate_new_pd(): "
        "new pd at address %p", pd);
    assert(PAGE_ALIGNED((uint32_t) pd));

    /* Initialize all page table entries as non-present */
    memset(pd, 0, PAGE_SIZE);

    return pd;
}

//done
/** @brief Allocates a new page table and adds it to a page directory entry and
 *         initializes the page directory entry lower 12 bits to correct flags.
 *
 *  @param pd Page directory pointer to add the page table to.
 *  @param virtual_address VM address corresponding to the page
 *         table to be created and added to the page directory.
 *  @return 0 on success, -1 on error.
 */
static int
add_new_pt_to_pd( uint32_t **pd, uint32_t virtual_address )
{
    /* Page directory should be valid */
    assert(is_valid_pd(pd));

    /* pd cannot be NULL */
    if (!pd) {
        log_warn("add_new_pt_to_pd(): "
                 "pd cannot be NULL!");
```

```
        return -1;
    }
    /* Get page directory index */
    uint32_t pd_index = PD_INDEX(virtual_address);

    /* pd entry we're adding the new page table must be NULL */
    if (pd[pd_index] != NULL) {
        log_warn("add_new_pt_to_pd(): "
                 "pd_index:0x%08lx to insert into pd:%p for "
                 "virtual_address:0x%08lx must be NULL!, instead "
                 "pd[pd_index]:%p",
                 pd_index, pd, virtual_address, pd[pd_index]);
        return -1;
    }
    /* Allocate a new empty page table, set it to page table index */
    void *pt = allocate_new_pt();
    if (!pt) {
        log_warn("add_new_pt_to_pd(): "
                 "unable to allocate new page table in pd:%p for "
                 "virtual_address:0x%08lx", pd, virtual_address);
        return -1;
    }
    pd[pd_index] = pt;

    /* Page table should be valid */
    assert(is_valid_pt(pt, pd_index));

    /* Set all page directory entries as kernel writable, determine
     * whether truly writable in page table entry. */
    pd[pd_index] = (uint32_t *)((uint32_t)pd[pd_index] | PE_USER_WRITABLE);

    /* Page directory should be valid */
    assert(is_valid_pd(pd));
    return 0;
}


/** @brief Gets pointer to page table entry in a given page directory.
 *         Allocates page table if necessary.
 *
 *  Invariant violations for page directory or page table will cause
 *  a crash.
 *
 *  @param pd Page table address
 *  @param virtual_address Virtual address corresponding to page table
 *  @return Pointer to a page table entry that can be dereferenced to get
 *          a physical address, NULL on failure.
 */
uint32_t *
get_ptep( const uint32_t **pd, uint32_t virtual_address )
{
    /* Checking if a pd is valid is expensive, hence an assert() */
    assert(is_valid_pd(pd));

    /* NULL pd, so abort and return NULL */
    if (!pd) {
        log_warn("get_ptep(): "
                 "pd cannot be NULL!");
        return NULL;
    }
    /* Get page directory and page table index */
    uint32_t pd_index = PD_INDEX(virtual_address);
    uint32_t pt_index = PT_INDEX(virtual_address);

    /* Page directory cannot have NULL entry at corresponding page table */
    if (!pd[pd_index]) {
        log_warn("get_ptep(): "
                 "pd:%p, virtual_address:0x%08lx, pd[pd_index] cannot be "
                 "NULL!", pd, virtual_address);
        return NULL;
    }
    /* Page directory must have correct flag bits set */
    affirm_msg(((uint32_t) pd[pd_index] & PE_USER_WRITABLE) == PE_USER_WRITABLE,
               "get_ptep(): "
               "pd[pd_index]:%p does not have PE_USER_WRITABLE bits set!",
                pd[pd_index]);

    /* Page table entry pointer zeroes out bottom 12 bits */
    uint32_t *ptep = (uint32_t *) TABLE_ADDRESS(pd[pd_index]);

    /* Return pointer to appropriate index */
    ptep += pt_index;
    affirm_msg(STACK_ALIGNED(ptep), "ptep:%p not stack aligned!", ptep);
    return ptep;
}

/** @brief Allocate new frame at given virtual memory address.
 *  Allocates page tables on demand.
 *
 *  If memory location already had a frame, checks to see if it has the
 *  same permissions. This is necessary since BSS and DATA occupy the same
 *  page-sized page-size aligned boundary. In fact BSS contiguously comes after
 *  DATA.
 *
 *  TODO CR4 needs to be set to prevent TLB flush
 *
 *  @return 0 on success, -1 on error
 *  */
static int
allocate_frame( uint32_t **pd, uint32_t virtual_address,
                write_mode_t write_mode, uint32_t sys_prog_flag )
{
    if (!is_valid_sys_prog_flag(sys_prog_flag)) {
        return -1;
    }
    /* is_valid_pd() is expensive, hence the assert() */
    assert(is_valid_pd(pd));
    log("allocate frame for vm:%p", (uint32_t *) virtual_address);

    /* pd is NULL, abort with error */
    if (!pd) {
        return -1;
    }
    /* Find page table entry corresponding to virtual address */
    uint32_t *ptep = get_ptep((const uint32_t **) pd, virtual_address);
    if (!ptep) {
        return -1;
    }
    uint32_t pt_entry = *ptep;

    /* If page table entry contains a non-NULL address */
    if (TABLE_ADDRESS(pt_entry)) {
        log_warn("frame already allocated for vm:%p", (uint32_t *) virtual_address)
;
        //return -1;

        /* Must be present else broken invariant */
        affirm_msg(pt_entry & PRESENT_FLAG, "pt_entry must be present");

        /* Ensure it's allocated with same flags. */
        if (write_mode == READ_WRITE) {
            if ((pt_entry & (PAGE_SIZE - 1)) !=
                (PE_USER_WRITABLE | sys_prog_flag)) {
                return -1;
```

```
            }
        } else {
            if ((pt_entry & (PAGE_SIZE - 1)) !=
                (PE_USER_READABLE | sys_prog_flag)) {
                return -1;
            }
        }

        // TODO: memory in this frame is zeroed out

    /* Page table entry contains a NULL address, allocate new physical frame */
    } else {
        uint32_t free_frame = physalloc();
        if (!free_frame) {
            return -1;
        }
        *ptep = free_frame;
    }

    if (write_mode == READ_WRITE) {
        *ptep |= (PE_USER_WRITABLE | sys_prog_flag);
    } else {
        *ptep |= (PE_USER_READABLE | sys_prog_flag);
    }

    return 0;
}

/** @brief Checks if system programmer flags are valid or not as defined in
 *         memory_manager_internal.h
 *
 *  @param sys_prog_flag System programmer flag
 *  @return 1 if valid flag, 0 otherwise
 */
int
is_valid_sys_prog_flag( uint32_t sys_prog_flag )
{
    switch (sys_prog_flag) {

        /* Empty sys_prog_flag is vacuously valid */
        case 0 :
            return 1;
            break;

        case NEW_PAGE_BASE_FLAG :
            return 1;
            break;

        case NEW_PAGE_CONTINUE_FROM_BASE_FLAG :
            return 1;
            break;

        /* Invalid sys programmer flag */
        default :
            return 0;
    }
}


/** @brief Allocates the system wide zero frame.
 *
 *  Currently this is only ever called by new_pages(), and thus
 *  allocated pages will always be writable by user programs. However since
 *  this is a zero frame, we set it to only have readonly access by the user.
 *
 *  When a user attempts to write to this physical frame, we check if the
 *  frame is the system wide zero frame and if it is, allocate it with
 *  user write permissions.
 *
 *  Requires that virtual address is valid
 *
 *  @param pd Page directory pointer
 *  @param virtual_address VM address we are allocating zero frame for
 *  @param sys_prog_flag Bits 9,10,11 to OR page table entry with
 *  @param 0 on success, -1 on error.
 */
int
allocate_user_zero_frame( uint32_t **pd, uint32_t virtual_address,
                          uint32_t sys_prog_flag )
{
    assert(PAGE_ALIGNED(virtual_address));
    if (!is_valid_sys_prog_flag(sys_prog_flag)) {
        log_info("allocate_user_zero_frame(): "
                 "invalid sys_prog_flag:0x%x",
                 sys_prog_flag);
        return -1;
    }
    /* is_valid_pd() is expensive, hence the assert() */
    assert(is_valid_pd(pd));
    log("allocate_user_zero_frame(): "
        "allocate zero frame for vm:%p", (uint32_t *) virtual_address);

    /* pd is NULL, abort with error */
    if (!pd) {
        log_info("allocate_user_zero_frame(): "
                 "allocate_zero_frame pd cannot be NULL!");
        return -1;
    }
    assert(is_valid_pd(pd));

    /* Find page table entry corresponding to virtual address */
    uint32_t *ptep = get_ptep((const uint32_t **) pd, virtual_address);

    /* pd is valid, so !ptep means must add new page table to page directory */
    if (!ptep) {
        uint32_t pd_index = PD_INDEX(virtual_address);
        affirm(pd[pd_index] == NULL);
        add_new_pt_to_pd(pd, virtual_address);
        log_info("allocate_user_zero_frame(): "
                 "adding new pt to pd for virutal_address:0x%08lx",
                 virtual_address);
    }
    ptep = get_ptep((const uint32_t **) pd, virtual_address);
    affirm(ptep);
    uint32_t pt_entry = *ptep;

    /* If page table entry contains a non-NULL address */
    if (TABLE_ADDRESS(pt_entry)) {
        log_info("allocate_user_zero_frame(): "
                 "zero frame already allocated!");
        return -1;
    }
    /* Allocate new physical frame */
    *ptep = sys_zero_frame;

    /* Mark as READ_ONLY for user */
    *ptep |= (sys_prog_flag | PE_USER_READABLE);
    return 0;
}

void
unallocate_user_zero_frame( uint32_t **pd, uint32_t virtual_address)
{
    /* is_valid_pd() is expensive, hence the assert() */
```

```
    assert(is_valid_pd(pd));
    log("unallocate zero frame for vm:%p", (uint32_t *) virtual_address);

    /* pd is NULL, abort with error */
    affirm_msg(pd, "unallocate_zero_frame pd cannot be NULL!");

    /* Find page table entry corresponding to virtual address */
    uint32_t *ptep = get_ptep((const uint32_t **) pd, virtual_address);
    affirm_msg(ptep, "unable to get page table entry");

    uint32_t pt_entry = *ptep;

    /* If page table entry contains a non-NULL address */
    affirm_msg(TABLE_ADDRESS(pt_entry) == sys_zero_frame,
               "should be a zero frame allocated here!");

    /* zero frame should be marked as READ_ONLY for users */
    affirm_msg((pt_entry & PE_USER_READABLE) == PE_USER_READABLE,
               "zero frame should be PE_USER_READABLE");

    /* Unallocate new physical frame */
    *ptep = 0;
}

/** Allocates a memory region in virtual memory.
 *
 *  If there aren't enough physical frames to satisfy allocation
 *  request, region is not allocated and function returns a negative
 *  value.
 *
 *  @param pd    Pointer to page directory
 *  @param start  Virtual memory addess for start of region to be allocated
 *  @param len    Length of region to be allocated
 *  @param write_mode 0 if read-only region, non-zero value if writable
 *
 *  @return 0 on success, negative value on failure.
 *  */
static int
allocate_region( uint32_t **pd, void *start, uint32_t len, write_mode_t write_mode
)
{
    uint32_t pages_to_alloc = (len + PAGE_SIZE - 1) / PAGE_SIZE;

    /* Ensure we have enough free frames to fulfill request */
    if (num_free_phys_frames() < pages_to_alloc) {
        return -1;
    }
    /* FIXME: Do we have any guarantee memory regions are page aligned?
     *        They should be, to some extent. At the very least, 2 memory
     *        regions should not be intersect with the same page, as they
     *        could require distinct permissions. THis might not be the case
     *        for data and bss, though, as both are read-write sections. */
    uint32_t u_start = (uint32_t)start;

    /* Allocate 1 frame at a time. */
    for (int i = 0; i < pages_to_alloc; ++i) {
        uint32_t virtual_address = u_start + PAGE_SIZE * i;
        uint32_t pd_index = PD_INDEX(virtual_address);

        affirm((pd[pd_index] != NULL && TABLE_ADDRESS(pd[pd_index]) != 0)
               || (pd[pd_index] == NULL));

        /* Get a new page table every time page directory entry is NULL */
        if (pd[pd_index] == NULL) {
            if (add_new_pt_to_pd(pd, virtual_address) < 0) {
                log_warn("allocate_region(): "
                         "unable to allocate new page table in pd:%p for "
                         "virtual_address: 0x%08lx", pd, virtual_address);
                return -1;
            }
        }
        int res = allocate_frame((uint32_t **)pd, virtual_address,
                                 write_mode, 0);
        if (res < 0) {
            // TODO CLEAN UP ALL PREVIOUSLY ALLOCATED PHYS FRAMES
            return -1;
        }
    }
    return 0;
}

static int
valid_memory_regions( simple_elf_t *elf )
{
    /* TODO:
     * - Check if memory regions intersect each other. If so, false.
     * - Check if memory regions intersect same page. If so and they
     *   have different read/write permissions, false.
     * - Othws, true
     *  */
    return 1;
}

/** @brief Enables paging mechanism. */
static void
enable_paging( void )
{
    /* Enable paging flag */
    uint32_t current_cr0 = get_cr0();
    set_cr0(current_cr0 | PAGING_FLAG);

    affirm_msg(!paging_enabled, "Paging should be enabled exactly once!");
    paging_enabled = 1;
}

static void
vm_set_pd( void *pd )
{
    uint32_t cr3 = get_cr3();
    /* Unset top 20 bits where new page table will be stored.*/
    cr3 &= PAGE_SIZE - 1;
    cr3 |= (uint32_t)pd;

    set_cr3(cr3);
}



/** @brief Frees a page table along with all physical frames
 *
 *
 *  //TODO do we free the page tables less than USER_MEM_START or do we
 *  keep them global for aliasing?
 *
 *  @param pt Page table to be freed
 */
static void
free_pt_memory( uint32_t *pt, int pd_index ) {

    affirm(is_valid_pt(pt, pd_index));

    for (int i = 0; i < PAGE_SIZE / sizeof(uint32_t); ++i) {

        /* pt holds physical frames for user memory */
```

```
        if (pd_index >= (USER_MEM_START >> PAGE_DIRECTORY_SHIFT)) {
            uint32_t pt_entry = pt[i];
            if (pt_entry & PRESENT_FLAG) {
                affirm_msg(TABLE_ADDRESS(pt_entry) != 0, "pt_entry:0x%08lx",
                           pt_entry);
                uint32_t phys_address = TABLE_ADDRESS(pt_entry);
                physfree(phys_address);

                // Zero the entry as well
                pt[i] = 0;
            }
        /* Currently free < USER_MEM_START */
        /* TODO final version shouldn't need to free kernel memory */
        } else {
            pt[i] = 0;
        }
    }
}

/** brief Walks the page directory and frees the entire page directory,
 *        page tables, and all physical frames
 */
void
free_pd_memory( void *pd )
{
    affirm(is_valid_pd(pd));
    uint32_t **pd_cast = (uint32_t **) pd;

    for (int i = NUM_KERN_PAGE_TABLES; i < PAGE_SIZE / sizeof(uint32_t); ++i) {

        uint32_t *pd_entry = pd_cast[i];

        /* Check page table if entry non-zero */
        if ((uint32_t) pd_entry & PRESENT_FLAG) {
            uint32_t *pt = (uint32_t *) TABLE_ADDRESS(pd_entry);
            free_pt_memory(pt, i);
            sfree(pt, PAGE_SIZE);
        }
    }
}
```

```
/** @file memory_manager_internal.h
 *  @brief Internal macros and functions that memory related functions in
 *         lib_memory_manager/ use.
 */


/* System programmer flags.
 * Bits 9, 10, 11 are used together to offer 8 possible flags that cannot
 * be bit-ORed with one another
 */
#define NEW_PAGE_BASE_FLAG (1 << 9)
#define NEW_PAGE_CONTINUE_FROM_BASE_FLAG (2 << 9)

/* 7 is 111 in binary, and we bitshift << 9 to only keep bits 9, 10, 11 in the
 * address
 */
#define SYS_PROG_FLAG(ADDRESS)\
    (((uint32_t)(ADDRESS)) & (7 << 9))

#define PAGING_FLAG (1 << 31)
#define WRITE_PROTECT_FLAG (1 << 16)

#define PAGE_GLOBAL_ENABLE_FLAG (1 << 7)

#define PAGE_OFFSET 0x00000FFF

/* 16MB = 4 PT */
#define NUM_KERN_PAGE_TABLES 4

/* Flags for page directory and page table entries */
#define PRESENT_FLAG (1 << 0)
#define RW_FLAG      (1 << 1)
#define USER_FLAG    (1 << 2)
#define GLOBAL_FLAG  (1 << 8)

#define PE_USER_READABLE (PRESENT_FLAG | USER_FLAG )
#define PE_USER_WRITABLE (PE_USER_READABLE | RW_FLAG)

/* Set global flag so TLB doesn't flush kernel entries */
#define PE_KERN_READABLE (PRESENT_FLAG | GLOBAL_FLAG)
#define PE_KERN_WRITABLE (PE_KERN_READABLE | RW_FLAG)
#define PE_UNMAPPED 0

#define TABLE_ENTRY_INVARIANT(TABLE_ENTRY)\
    ((((uint32_t)(TABLE_ENTRY) != 0) && (TABLE_ADDRESS(TABLE_ENTRY) != 0))\
    || ((uint32_t)(TABLE_ENTRY) == 0))

uint32_t *get_ptep( const uint32_t **pd, uint32_t virtual_address );
int is_valid_sys_prog_flag( uint32_t sys_prog_flag );
void unallocate_frame( uint32_t **pd, uint32_t virtual_address );
```

```
/** @file panic.c
 *  @brief This file contains the panic() function which is called whenever
 *         a thread crashes.
 *
 *  We note that if "one thread in a multi-threaded application experiences
 *  a fatal exception, the application as a whole is unlikely to continue in a
 *  useful fashion." Because the crashed thread could have been holding on
 *  to locks such as mutexes and on crashing be unable to unlock those locks
 *  for other threads to use. Therefore whenever any child thread panics, the
 *  goal is to cause other threads to vanish as well, and so panic() invokes
 *  task_vanish().
 *
 *  @author edited by Nicklaus Choo (nchoo), acknowledgements below.
 *
 *  ---
 *
 *  Copyright (c) 1996-1995 The University of Utah and
 *  the Computer Systems Laboratory at the University of Utah (CSL).
 *  All rights reserved.
 *
 *  Permission to use, copy, modify and distribute this software is hereby
 *  granted provided that (1) source code retains these copyright, permission,
 *  and disclaimer notices, and (2) redistributions including binaries
 *  reproduce the notices in supporting documentation, and (3) all advertising
 *  materials mentioning features or use of this software display the following
 *  acknowledgement: ``This product includes software developed by the
 *  Computer Systems Laboratory at the University of Utah.''
 *
 *  THE UNIVERSITY OF UTAH AND CSL ALLOW FREE USE OF THIS SOFTWARE IN ITS "AS
 *  IS" CONDITION.  THE UNIVERSITY OF UTAH AND CSL DISCLAIM ANY LIABILITY OF
 *  ANY KIND FOR ANY DAMAGES WHATSOEVER RESULTING FROM THE USE OF THIS SOFTWARE.
 *
 *  CSL requests users of this software to return to csl-dist@cs.utah.edu any
 *  improvements that they make and grant CSL redistribution rights.
 */

#include <stdarg.h> /* va_list(), va_end() */
#include <logger.h> /* log_crit() */
#include <asm.h>    /* disable interrupts */

/** @brief This function is called by the assert() macro defined in assert.h;
 *         it's also a nice simple general-purpose panic function. Ceases
 *         execution of all running threads.
 *
 *  Any thread that calls panic should pass the reason for panic() in the
 *  argument fmt. Observe that va_end() is used after va_start() to prevent
 *  stack corruption.
 *
 *  @param fmt String to print out on panic
 *  @param ... Other arguments put into fmt
 *  @return Void.
 */
void panic( const char *fmt, ... )
{
    /* Print error that occurred */
    va_list args;
    va_start(args, fmt);
    vtprintf(fmt, args, CRITICAL_PRIORITY);
    va_end(args);

#include <simics.h>
    MAGIC_BREAK;

    disable_interrupts();
    while (1) {
        continue;
    }
    // call halt();

    return;
}
```

```c
/** @file scheduler.c
 *  @brief A round-robin scheduler.
 *
 *  Note: As mutexes are implemented by manipulating the schedulers
 *  so that threads waiting on a lock are not executed, the scheduler
 *  itself uses disable/enable interrupts to protect critical sections.
 *  All critical sections protected this way must be short. */

#include <scheduler.h>
#include <task_manager.h>   /* tcb_t */
#include <task_manager_internal.h> /* To use Q MACROS on tcb */
#include <variable_queue.h> /* Q_NEW_LINK() */
#include <context_switch.h> /* context_switch() */
#include <assert.h>         /* affirm() */
#include <malloc.h>         /* smalloc(), sfree() */
#include <stdint.h>         /* uint32_t */
#include <cr.h>             /* get_esp0() */
#include <logger.h>         /* log_warn() */
#include <asm.h>            /* enable/disable_interrupts() */

#include <simics.h>

/* Timer interrupts every ms, we want to swap every 2 ms. */
#define WAIT_TICKS 2

/* Whether scheduler has been initialized */
static int scheduler_init = 0;

/* Thread queues.*/
static queue_t runnable_q;
static tcb_t *running_thread = NULL; // Currently running thread

static void swap_running_thread( tcb_t *to_run, status_t store_status,
        void (*callback)(tcb_t *, void *), void *data );
static void switch_threads(tcb_t *running, tcb_t *to_run);

/** @brief Whether the scheduler is initialized
 *
 *  @return 1 if initialized, 0 if not */
int
is_scheduler_init( void )
{
    return scheduler_init;
}

void
print_status(status_t status)
{
    switch (status) {
        case RUNNING:
            log_info("Status is RUNNING");
            break;
        case RUNNABLE:
            log_info("Status is RUNNABLE");
            break;
        case DESCHEDULED:
            log_info("Status is DESCHEDULED");
            break;
        case BLOCKED:
            log_info("Status is BLOCKED");
            break;
        case DEAD:
            log_info("Status is DEAD");
            break;
        case UNINITIALIZED:
            log_info("Status is UNINITIALIZED");
            break;
        default:
            log_info("Status is UNKNOWN");
            break;
    }
}

/** @brief Yield execution of current thread, storing it at
 *         the runnable queue if store_status is RUNNABLE.
 *
 *  @param store_status Status which currently running thread will take
 *  @param tid          Id of thread to yield to, -1 if any
 *  @param callback     Function to be called atomically with tcb of
 *                      current thread. MUST BE SHORT
 *  @return 0 on success, negative value on error */
int
yield_execution( status_t store_status, int tid,
        void (*callback)(tcb_t *, void *), void *data )
{
    affirm(scheduler_init);
    if (!scheduler_init) {
        log_warn("Attempting to call yield but scheduler is not initialized");
        return -1;
    }

    if (tid == get_running_tid()) {
        affirm_msg(store_status == RUNNABLE, "Trying to yield to self"
                " but with non-RUNNABLE status");
        return 0; // yielding to self
    }

    /* Get tcb to swap to */
    tcb_t *tcb;
    if (tid == -1) {
        disable_interrupts();
        tcb = Q_GET_FRONT(&runnable_q);

        /* If we are coincidentally at the front of the queue,
         * we only want to yield to ourselves if we want to remain
         * runnable. */
        if (tcb && tcb->tid == get_running_tid()) {
            /* If store_status is not runnable find someone else. */
            if (store_status != RUNNABLE) {
                Q_REMOVE(&runnable_q, tcb, scheduler_queue);
                tcb = Q_GET_FRONT(&runnable_q);
            }
        }

        if (!tcb) {
            if (store_status == RUNNABLE) {
                enable_interrupts();
                return 0; /* Yield to self*/
            } else {
                print_status(store_status);
                panic("DEADLOCK, scheduler has no one to run!");
            }
        }

        /* Any thread in runnable queue is either running (and also
         * in runnable queue because of a yield), or is just runnable. */
        assert(get_tcb_status(tcb) == RUNNABLE ||
                get_tcb_status(tcb) == RUNNING);

    } else {
        /* find_tcb() is already guarded by a mutex */
        tcb = find_tcb(tid); // FIXME: Could this cause an issue? Recursion or smth
        if (!tcb) {
            log_warn("Trying to yield_execution to non-existent"
```

```
                    " thread with tid %d", tid);
            return -1; /* Thread not found */
        }
        disable_interrupts();
        if (get_tcb_status(tcb) != RUNNABLE) {
            log_warn("Trying to yield_execution to non-runnable"
                    " thread with tid %d", tid);
            enable_interrupts();
            return -1;
        }
    }

    /* If this thread is to be made not-runnable, ensure it is not
     * on the runnable queue. The currently running thread can be
     * on the runnable queue if it was yielded to previously. */
    if (store_status != RUNNABLE &&
            Q_IN_SOME_QUEUE(running_thread, scheduler_queue)) {
        /* tcb must be in scheduler_q, therefore we can safely remove it */
        Q_REMOVE(&runnable_q, running_thread, scheduler_queue);
    }

    swap_running_thread(tcb, store_status, callback, data);
    return 0;
}

/** @brief Gets tid of currently active thread.
 *
 *  @return Id of running thread */
int
get_running_tid( void )
{
    /* If running_thread is NULL, we have a single thread with tid 0. */
    if (!running_thread) {
        return 0;
    }
    return running_thread->tid;
}

/** @brief Gets currently active thread.
 *
 *  @return Running thread, NULL if no such thread */
tcb_t *
get_running_thread( void )
{
    return running_thread;
}

/** @brief Gets pointer to PCB that currently running thread belongs to
 *
 *  @return non-NULL pointer of owning task of currently running thread if
 *          currently running thread is non-NULL, NULL otherwise
 */
pcb_t *
get_running_task( void )
{
    if (!running_thread) {
        affirm(!scheduler_init);
        return NULL;
    } else {
        affirm(running_thread->owning_task);
        return running_thread->owning_task;
    }
}


/** @brief Initializes scheduler and registers its first thread.
 *
 *  @return 0 on success, -1 on error
 */
static int
init_scheduler( void )
{
    /* Initialize once and only once */
    affirm(!scheduler_init);

    Q_INIT_HEAD(&runnable_q);

    scheduler_init = 1;

    return 0;
}

static int
make_thread_runnable_helper( uint32_t tid, int switch_safe )
{
    log("Making thread %d runnable", tid);

    tcb_t *tcbp = find_tcb(tid);
    if (!tcbp)
        return -1;

    /* Add tcb to runnable queue, as any thread starts as runnable */
    disable_interrupts();
    if (tcbp->status == RUNNABLE || tcbp->status == RUNNING) {
        log_warn("Trying to make runnable thread %d runnable again", tid);
        if (!switch_safe)
            enable_interrupts();
        return -1;
    }

    if (!scheduler_init) {
        init_scheduler();
        tcbp->status = RUNNING;
        running_thread = tcbp;
    } else {
        if (tcbp->status == UNINITIALIZED || switch_safe) {
            tcbp->status = RUNNABLE;
            Q_INSERT_TAIL(&runnable_q, tcbp, scheduler_queue);
        } else {
            /* "Improve" preemptibility by immediately swapping to thread
             * being made runnable. To avoid doing so for newly registered
             * threads, only swap immediately if status != UNINITIALIZED. */
            swap_running_thread(tcbp, RUNNABLE, NULL, NULL);
        }
    }

    if (!switch_safe)
        enable_interrupts();

    return 0;
}

int
switch_safe_make_thread_runnable( uint32_t tid )
{
    return make_thread_runnable_helper(tid, 1);
}

/** @brief Registers thread with scheduler. After this call,
 *         the thread may be executed by the scheduler.
 *
 *  @param tid Id of thread to register
 *
 *  @return 0 on success, negative value on error */
```

```
/* TODO: Think of synchronization here*/
int
make_thread_runnable( uint32_t tid )
{
    return make_thread_runnable_helper(tid, 0);
}

void
scheduler_on_tick( unsigned int num_ticks )
{
    if (!scheduler_init)
        return;

    if (num_ticks % WAIT_TICKS == 0) {
        disable_interrupts();

        tcb_t *to_run;
        /* Do nothing if there's no thread waiting to be run */
        if (!(to_run = Q_GET_FRONT(&runnable_q))) {
            enable_interrupts();
            return;
        }
        Q_REMOVE(&runnable_q, to_run, scheduler_queue);

        swap_running_thread(to_run, RUNNABLE, NULL, NULL);
    }
}

/* ------- HELPER FUNCTIONS -------- */

/** @brief Swaps the running thread to to_run.
 *
 *  @pre Interrupts disabled when called.
 *
 *  @param to_run Thread to run next
 *  @param store_at Queue in which to store old thread in case store_status
 *                  is blocked. For any other store status scheduler determines
 *                  queue to store thread into.
 *  @param store_status Status with which to store old thread.
 *  @return Void.
 */
static void
swap_running_thread( tcb_t *to_run, status_t store_status,
        void (*callback)(tcb_t *, void *), void *data )
{
    assert(to_run);
    affirm_msg(scheduler_init, "Scheduler has to be initialized before calling "
               "swap_running_thread");

    /* yield_execution will not remove the thread it yields to from
     * the runnable queue. Therefore, we give it more CPU cycles without
     * an explicit context switch by just returning.*/

    /* No-op if we swap with ourselves */
    if (to_run->tid == running_thread->tid) {
        affirm(store_status == RUNNABLE);
        enable_interrupts();
        return;
    }

    /* Validate store status */
    switch (store_status) {
        case RUNNING:
            panic("Trying to store thread with status RUNNING!");
            break;
        case RUNNABLE:
            break;
        case DESCHEDULED:
            break;
        case BLOCKED:
            break;
        case DEAD:
            break;
        case UNINITIALIZED:
            panic("Trying to store thread with status UNINITIALIZED!");
            break;
        default:
            panic("Trying to store thread with unknown status!");
            break;
    }

    assert(running_thread->status == RUNNING);

    tcb_t *running = running_thread;
    running->status = store_status;

    /* Data structure for other statuses are managed by their own components,
     * scheduler is only responsible for managing runnable/running threads. */
    /* If running thread is already in runnable queue, don't insert again */
    if (store_status == RUNNABLE && !Q_IN_SOME_QUEUE(running, scheduler_queue))
        Q_INSERT_TAIL(&runnable_q, running, scheduler_queue);
    else if (callback) {
        /* FIXME: What happens if the callback calls a yield_execution? */
        callback(running, data);
    }

    /* FIXME when a thread calls _vanish() on itself, it needs to remain
     * RUNNING until it is done free-ing itself, which is called via
     * the callback set to free_tcb_callback, so we update running->status
     * to DEAD only after free_tcb_callback() has completed */


    /* Update running thread after, since callback expects to be called by
     * original running thread. */
    to_run->status = RUNNING;
    running_thread = to_run;

    /* Interrupts are enabled inside context switch, once it's safe to do so. */
    switch_threads(running, to_run);

    /* Nothing which must run should be placed after switch_threads as,
     * when we context switch to a new thread for the first time, the stack
     * will be setup for it to return directly to the call_fork asm wrapper. */
}

static void
switch_threads(tcb_t *running, tcb_t *to_run)
{
    assert(running && to_run);
    assert(to_run->tid != running->tid);

    /* Let thread know where to come back to on USER->KERN mode switch */
    set_esp0((uint32_t)to_run->kernel_stack_hi);

    context_switch((void **)&(running->kernel_esp), to_run->kernel_esp);
}
```

```
#ifndef SCHEDULER_H_
#define SCHEDULER_H_

#include <task_manager.h>   /* status_t */
#include <stdint.h>         /* uint32_t */
#include <variable_queue.h> /* Q_NEW_HEAD() */

enum status { RUNNING, RUNNABLE, DESCHEDULED, BLOCKED, DEAD, UNINITIALIZED };
typedef enum status status_t;

/* Queue head definition */
Q_NEW_HEAD(queue_t, tcb);

int is_scheduler_init( void );
tcb_t *get_running_thread( void );
pcb_t *get_running_task( void );

int get_running_tid( void );
void scheduler_on_tick( unsigned int num_ticks );
int make_thread_runnable( uint32_t tid );
int switch_safe_make_thread_runnable( uint32_t tid );
int yield_execution( status_t store_status, int tid,
        void (*callback)(tcb_t *, void *), void *data );

#endif /* SCHEDULER_H_ */
```

```
/** @brief Module for management of tasks.
 *  Includes context switch facilities. */

// TODO maybe a function such as is_legal_pcb, is_legal_tcb for invariant checks?
//
#include <task_manager.h>
#include <task_manager_internal.h>
#include <assert.h> /* affirm() */

#include <scheduler.h>  /* add_tcb_to_run_queue() */
#include <eflags.h> /* get_eflags*/
#include <seg.h>    /* SEGSEL_... */
#include <stdint.h> /* uint32_t, UINT32_MAX */
#include <stddef.h> /* NULL */
#include <malloc.h> /* malloc, smemalign, free, sfree */
#include <elf_410.h>    /* simple_elf_t */
#include <page.h>   /* PAGE_SIZE */
#include <cr.h>     /* set_esp0 */
#include <string.h> /* memset */
#include <assert.h> /* affirm, assert */
#include <simics.h> /* sim_reg_process */
#include <logger.h> /* log */
#include <iret_travel.h>    /* iret_travel */
#include <atomic_utils.h>   /* add_one_atomic */
#include <memory_manager.h> /* get_new_page_table, vm_enable_task */
#include <variable_queue.h> /* Q_INSERT_TAIL */
#include <lib_thread_management/hashmap.h>  /* map_* functions */
#include <lib_thread_management/mutex.h>    /* mutex_t */

#define ELF_IF (1 << 9);

static uint32_t get_unique_tid( void );
static uint32_t get_unique_pid( void );
static uint32_t get_user_eflags( void );

Q_NEW_HEAD(pcb_list_t, pcb);
static pcb_list_t pcb_list;


static mutex_t pcb_list_mux;
static mutex_t tcb_map_mux;

/** @brief Next pid to be assigned. Only to be updated by get_unique_pid */
static uint32_t next_pid = 0;
/** @brief Next tid to be assigned. Only to be updated by get_unique_tid */
static uint32_t next_tid = 0;

void *
get_tcb_pd(tcb_t *tcb)
{
    return tcb->owning_task->pd;
}

uint32_t
get_tcb_tid(tcb_t *tcb)
{
    return tcb->tid;
}

void
set_task_exit_status( int status )
{
    tcb_t *tcb = get_running_thread();
    affirm(tcb);
    affirm(tcb->owning_task);
    pcb_t *owning_task = tcb->owning_task;
    /* Atomically update exit status of owning task */
    mutex_lock(&owning_task->set_status_vanish_wait_mux);
    tcb->owning_task->exit_status = status;
    mutex_unlock(&owning_task->set_status_vanish_wait_mux);
}


/** @brief Initializes task manager's resources
 *
 *   @return Void
 */
void
task_manager_init ( void )
{
    map_init();
    mutex_init(&pcb_list_mux);
    mutex_init(&tcb_map_mux);
    Q_INIT_HEAD(&pcb_list);
}

/** @brief Gets the status of a tcb
 *
 *  Needs to be guarded by synchronization from invoking function
 *
 *  @param tcb TCB from which to get status from
 *  @return Status of a thread i.e. RUNNING, RUNNABLE, DESCHEDULED, ...
 */
status_t
get_tcb_status( tcb_t *tcb )
{
    return tcb->status;
}

/** @brief Changes the page directory in the PCB to new_pd
 *
 *  @param new_pd New page directory pointer to change to
 *  @return the old page directory.
 */
void *
swap_task_pd( void *new_pd )
{
    assert(is_valid_pd(new_pd));

    /* Find PCB to swap its stored page directory */
    uint32_t pid = get_pid();
    pcb_t *pcb = find_pcb(pid);
    affirm(pcb);

    /* Swap page directories */
    void *old_pd = pcb->pd;
    pcb->pd = new_pd;

    /* Check and return the old page directory */
    affirm(is_valid_pd(old_pd));
    return old_pd;
}

/** @brief Creates a task
 *
 *  @param pid Pointer where task id for new task is stored
 *  @param tid Pointer where thread id for new thread is stored
 *  @param elf Elf header for use in allocating new task's memory
 *
 *  @return 0 on success, negative value on failure.
 * */
int
create_task( uint32_t *pid, uint32_t *tid, simple_elf_t *elf )
```

```
{
    if (!pid) return -1;
    if (!tid) return -1;
    // TODO: Think about preconditions for this.
    // Paging fine, how about making it a critical section?

    /* Allocates physical memory to a new page table and enables VM */
    /* Ensure alignment of page table directory */
    /* Create new task. Stack is defined here to be the last PAGE_SIZE bytes. */
    void *pd = new_pd_from_elf(elf, UINT32_MAX - PAGE_SIZE + 1, PAGE_SIZE);
    if (!pd) {
        return -1;
    }

    if (create_pcb(pid, pd, NULL) < 0) {
        sfree(pd, PAGE_SIZE);
        return -1;
    }
    pcb_t *pcb = find_pcb(*pid);
    affirm(pcb);

    if (create_tcb(*pid, tid) < 0) {
        // TODO: Delete pcb, and return -1
        sfree(pd, PAGE_SIZE);
        return -1;
    }
    return 0;
}

/** NOTE: Not to be used in context-switch, only when running task
 *  for the first time
 *
 *  Enables virtual memory of task. Use this before transplanting data
 *  into task's memory.
 *  */
int
activate_task_memory( uint32_t pid )
{
    /* Likely messing up direct mapping of kernel memory, and
     * some instruction after task_prepare is being seen as invalid?*/
    pcb_t *pcb;
    if ((pcb = find_pcb(pid)) == NULL)
        return -1;

    /* Update the page directory and enable VM if necessary */
    vm_enable_task(pcb->pd);

    return 0;
}

/** NOTE: Not to be used in context-switch, only when running task
 *  for the first time
 *
 *  Should only ever be called once, and after task has been initialized
 *  after a call to new_task.
 *  The caller is supposed to install memory on the new task before
 *  calling this function. Stack pointer should be appropriately set
 *  if any arguments have been loaded on stack.
 *
 *  @param tid Id of thread to run
 *  @param esp Stack pointer
 *  @param entry_point First program instruction
 *
 *  @return Never returns.
 *  */
void
task_set_active( uint32_t tid )
{
    tcb_t *tcb;
    affirm((tcb = find_tcb(tid)) != NULL);

    /* Let scheduler know it can now run this thread */
    /* Let scheduler know it can now run this thread if it doesn't know */
    /* TODO in a bit of a pickle because we need to call disable_interrupts()
     * to check this?
     */
    if (tcb->status == UNINITIALIZED) {
        make_thread_runnable(tid);
    }
}

void
task_start( uint32_t tid, uint32_t esp, uint32_t entry_point )
{
    tcb_t *tcb;
    affirm((tcb = find_tcb(tid)) != NULL);

    /* Before going to user mode, update esp0, so we know where to go back to */
    set_esp0((uint32_t)tcb->kernel_stack_hi);

    /* We're currently going directly to entry point. In the future,
     * however, we should go to some "receiver" function which appropriately
     * sets user registers and segment selectors, and lastly RETs to
     * the entry_point. */
    affirm(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));
    iret_travel(entry_point, SEGSEL_USER_CS, get_user_eflags(),
        esp, SEGSEL_USER_DS);

    /* NOTREACHED */
    panic("iret_travel should not return");
}

/** Looks for pcb with given pid.
 *
 *  @param pid Task id to look for
 *
 *  @return Pointer to pcb on success, NULL on failure */
pcb_t *
find_pcb( uint32_t pid )
{
    mutex_lock(&pcb_list_mux);
    pcb_t *res = Q_GET_FRONT(&pcb_list);
    while (res && res->pid != pid)
        res = Q_GET_NEXT(res, task_link);
    mutex_unlock(&pcb_list_mux);
    return res;
}

/** Looks for tcb with given tid.
 *
 *  @param tid Thread id to look for
 *
 *  @return Pointer to tcb on success, NULL on failure */
tcb_t *
find_tcb( uint32_t tid )
{
    mutex_lock(&tcb_map_mux);
    tcb_t *res = (tcb_t *)map_get(tid);
    mutex_unlock(&tcb_map_mux);
    return res;
}

/** @brief Initializes new pcb, and corresponding tcb.
 *
```

```
 *  @param pid Pointer to where pid should be stored
 *  @param pd  Pointer to page directory for new task
 *  @return 0 on success, negative value on error
 */
int
create_pcb( uint32_t *pid, void *pd, pcb_t *parent_pcb)
{
    pcb_t *pcb = smalloc(sizeof(pcb_t));
    if (!pcb)
        return -1;

    //mutex_init(&thread_list_mux); TODO: Enable this
    *pid = get_unique_pid();
    pcb->pid = *pid;
    pcb->pd = pd;
    pcb->prepared = 0;

    /* Initialize thread queue */
    Q_INIT_HEAD(&(pcb->owned_threads));
    pcb->num_threads = 0;

    /* Initialize set_status_vanish_wait_mux and other structures for
     * set_status(), vanish(), wait() */
    if (mutex_init(&(pcb->set_status_vanish_wait_mux)) < 0) {
        return -1;
    }
    Q_INIT_HEAD(&(pcb->vanished_child_tasks_list));
    Q_INIT_HEAD(&(pcb->waiting_threads_list));

    if (parent_pcb) {
        pcb->parent_pcb = parent_pcb;
    }
    Q_INIT_ELEM(pcb, vanished_child_tasks_link);



    /* Add to pcb linked list*/
    mutex_lock(&pcb_list_mux);
    Q_INIT_ELEM(pcb, task_link);
    Q_INSERT_TAIL(&pcb_list, pcb, task_link);
    mutex_unlock(&pcb_list_mux);

    return 0;
}

/** @brief Initializes new tcb. Does not add thread to scheduler.
 *     This should be done by whoever creates this thread.
 *
 *  @param pid Id of owning task
 *  @param tid Pointer to where id of new thread will be stored
 *  @return 0 on success, negative value on failure
 * */
int
create_tcb( uint32_t pid, uint32_t *tid )
{
    pcb_t *owning_task;
    if ((owning_task = find_pcb(pid)) == NULL)
        return -1;

    tcb_t *tcb = smalloc(sizeof(tcb_t));
    if (!tcb) {
        return -1;
    }

    *tid = get_unique_tid();
    tcb->tid = *tid;

    /* If debug mode is set, we let each kernel stack be PAGE_SIZE of usable
     * memory, followed by PAGE_SIZE of unusable memory to prevent kernel
     * stacks from overlapping onto each other during execution.
     */
#ifdef DEBUG
    tcb->kernel_stack_lo = smemalign(PAGE_SIZE, 2 * PAGE_SIZE);
    if (!tcb->kernel_stack_lo) {
        sfree(tcb, sizeof(tcb_t));
        return -1;
    }
    uint32_t **parent_pd = owning_task->pd;
    uint32_t pd_index = PD_INDEX(tcb->kernel_stack_lo);
    uint32_t *parent_pt = (uint32_t *) TABLE_ADDRESS(parent_pd[pd_index]);
    uint32_t pt_index = PT_INDEX(tcb->kernel_stack_lo);
    parent_pt[pt_index] = 0x0;
    tcb->kernel_stack_lo += PAGE_SIZE / sizeof(uint32_t);
#else
    tcb->kernel_stack_lo = smalloc(PAGE_SIZE);
    if (!tcb->kernel_stack_lo) {
        sfree(tcb, sizeof(tcb_t));
        return -1;
    }
#endif

    tcb->status = UNINITIALIZED;

    /* Add to owning task's list of threads */
    tcb->owning_task = owning_task;

    /* TODO: Add mutex to pcb struct and lock it here.
     *       For now, this just checks that we're not
     *       adding a second thread to an existing task. */
    affirm(!Q_GET_FRONT(&owning_task->owned_threads));

    /* Link for when this thread calls wait() */
    Q_INIT_ELEM(tcb, waiting_threads_link);

    /* Add to owning task's list of threads, increment num_threads not DEAD */
    //mutex_lock(&owning_task->thread_list_mux);
    Q_INIT_ELEM(tcb, scheduler_queue);
    Q_INIT_ELEM(tcb, tid2tcb_queue);
    Q_INIT_ELEM(tcb, owning_task_thread_list);
    Q_INSERT_TAIL(&(owning_task->owned_threads), tcb, owning_task_thread_list);
    ++(owning_task->num_threads);
    //mutex_unlock(&owning_task->thread_list_mux);

    log("Inserting thread with tid %lu", tcb->tid);
    mutex_lock(&tcb_map_mux);
    map_insert(tcb);
    mutex_unlock(&tcb_map_mux);

    /* memset the whole thing, TODO delete this in future, only good for
     * debugging when printing the whole stack
     */
    memset(tcb->kernel_stack_lo, 0, PAGE_SIZE);

    log("create_tcb(): tcb->stack_lo:%p", tcb->kernel_stack_lo);
    // FIXME faults here

    tcb->kernel_esp = tcb->kernel_stack_lo;
    tcb->kernel_esp = (uint32_t *)(((uint32_t)tcb->kernel_esp) +
              PAGE_SIZE - sizeof(uint32_t));
    tcb->kernel_stack_hi = tcb->kernel_esp;
    log("create_tcb(): tcb->kernel_stack_hi:%p", tcb->kernel_stack_hi);


    // set the canary
```

```
    *(tcb->kernel_stack_hi) = 0xcafebabe;

    *(tcb->kernel_stack_lo) = 0xdeadbeef;
    return 0;
    return 0;
}

/** @brief Returns number of threads in the owning task
 *
 *  The return value cannot be zero. The moment a tcb_t is initialize it
 *  is immediately added to its owning_task's owned_threads field, which is
 *  a list of threads owned by that task.
 *
 *  @param tcbp Pointer to tcb
 *  @return Number of threads in owning task
 */
int
get_num_threads_in_owning_task( tcb_t *tcbp )
{
    /* Argument checks */
    affirm_msg(tcbp, "Given tcb pointer cannot be NULL!");
    affirm_msg(tcbp->owning_task, "Tcb pointer to owning task cannot be NULL!");

    /* Check that we have a legal number of threads and return */
    uint32_t num_threads = tcbp->owning_task->num_threads;
    affirm_msg(num_threads >= 0, "Owning task must have non-negative threads!");
    return num_threads;
}

/** @brief Gets the highest writable address of the kernel stack for thread
 *     that corresponds to supplied TCB
 *
 *  Requires that tcbp is non-NULL, and that its kernel_stack_hi field is
 *  stack aligned.
 *
 *  @param tcbp Pointer to TCB
 *  @return Kernel stack highest writable address
 */
void *
get_kern_stack_hi( tcb_t *tcbp )
{
    /* Argument checks */
    affirm_msg(tcbp, "tcbp cannot be NULL!");

    /* Invariant checks to ensure returned value is legal */
    affirm_msg(tcbp->kernel_stack_hi, "tcbp->kernel_stack_hi cannot be NULL!");
    affirm_msg(STACK_ALIGNED(tcbp->kernel_stack_hi), "tcbp->kernel_stack_hi "
           "must be stack aligned!");
    return tcbp->kernel_stack_hi;
}

void *
get_kern_stack_lo( tcb_t *tcbp )
{
    /* Argument checks */
    affirm_msg(tcbp, "tcbp cannot be NULL!");

    /* Invariant checks to ensure returned value is legal */
    affirm_msg(tcbp->kernel_stack_lo, "tcbp->kernel_stack_lo cannot be NULL!");
    affirm_msg(STACK_ALIGNED(tcbp->kernel_stack_lo), "tcbp->kernel_stack_lo "
           "must be stack aligned!");
    return tcbp->kernel_stack_lo;
}

/** @brief Sets the kernel_esp field in supplied TCB
 *
 *  Requires that tcbp is non-NULL and that kernel_esp is stack aligned and also
 *  non-NULL
 *
 *  @param tcbp Pointer to TCB
 *  @param kernel_esp Kernel esp the next time this thread goes into kernel
 *     mode either through a context switch or mode switch
 *  @return Void.
 */
void
set_kern_esp( tcb_t *tcbp, uint32_t *kernel_esp )
{
    /* Argument checks */
    affirm_msg(tcbp, "tcbp cannot be NULL!");
    affirm_msg(kernel_esp, "kernel_esp cannot be NULL!");
    affirm_msg(STACK_ALIGNED(kernel_esp), "kernel_esp must be stack aligned!");

    tcbp->kernel_esp = kernel_esp;
}

/* ------ HELPER FUNCTIONS ------ */

/** @brief Returns eflags with PL altered to 3 */
static uint32_t
get_user_eflags( void )
{
    uint32_t eflags = get_eflags();

    /* Any IOPL | EFL_IOPL_RING3 == EFL_IOPL_RING3 */
    eflags |= EFL_IOPL_RING0; /* Set privilege level to user */
    eflags |= EFL_RESV1;      /* Maitain reserved as 1 */
    eflags &= ~(EFL_AC);      /* Disable alignment-checking */
    eflags |= EFL_IF;         /* Enable hardware interrupts */

    return eflags;
}

/** @brief Returns a unique pid. */
static uint32_t
get_unique_pid( void )
{
    return add_one_atomic(&next_pid);
}

/** @brief Returns a unique tid. */
static uint32_t
get_unique_tid( void )
{
    return add_one_atomic(&next_tid);
}

/** @brief Returns the pid of the currently running thread
 *
 *  @return Void.
 */
uint32_t
get_pid( void )
{
    /* Get TCB */
    tcb_t *tcb = get_running_thread();
    assert(tcb);

    /* Get PCB to get and return pid */
    pcb_t *pcb = tcb->owning_task;
    assert(pcb);
    uint32_t pid = pcb->pid;
    return pid;
}
```

```
/** @brief Frees a TCB
 *
 *  @pre TCB must not be in any list/queue
 *  @param tcb Pointer to TCB to be freed
 *  @return Void.
 */
void
free_tcb(tcb_t *tcb)
{
    affirm(tcb);
    affirm(!(Q_IN_SOME_QUEUE(tcb, waiting_threads_link)));
    affirm(!(Q_IN_SOME_QUEUE(tcb, scheduler_queue)));
    affirm(!(Q_IN_SOME_QUEUE(tcb, tid2tcb_queue)));
    affirm(!(Q_IN_SOME_QUEUE(tcb, owning_task_thread_list)));
    sfree(tcb, sizeof(tcb));
}
```

```
/** @file task_manager_internal.h
 *  @brief Internal struct definitions not published in the interface
 *
 *  @author Andre Nascimento (anascime)
 *  @author Nicklaus Choo (nchoo)
 *
 */

#ifndef TASK_MANAGER_INTERNAL_H_
#define TASK_MANAGER_INTERNAL_H_

#include <variable_queue.h> /* Q_NEW_LINK */
#include <scheduler.h> /* status_t */
#include <lib_thread_management/mutex.h> /* mutex_t */

/* PCB owned threads queue definition */
Q_NEW_HEAD(owned_threads_queue_t, tcb);

typedef struct pcb pcb_t;
typedef struct tcb tcb_t;

Q_NEW_HEAD(vanished_child_tasks_list_t, pcb);
Q_NEW_HEAD(waiting_threads_list_t, tcb);

/** @brief Task control block */
struct pcb {
    /* For set_status(), vanish(), wait() */
    mutex_t set_status_vanish_wait_mux;
    vanished_child_tasks_list_t vanished_child_tasks_list;
    waiting_threads_list_t waiting_threads_list;
    pcb_t *parent_pcb;

    /* When the last thread of this task has vanished, this link is used
     * to put the PCB on its parent task's vanished_child_tasks_list */
    Q_NEW_LINK(pcb) vanished_child_tasks_link;

    //mutex_t thread_list_mux; // TODO enable mutex
    void *pd; /* page directory */
    owned_threads_queue_t owned_threads; /* list of owned threads */
    uint32_t num_threads; /* number of threads not DEAD */
    Q_NEW_LINK(pcb) task_link; // Embedded list of tasks
    uint32_t pid; /* Task/process ID */
    int prepared; /* Whether this task's VM has been initialized */
    int exit_status; /* Task exit status */
};

/** @brief Thread control block */
struct tcb {
    /* When this thread calls wait(), this link is used to put the TCB on its
     * owning_task's waiting_threads_list */
    Q_NEW_LINK(tcb) waiting_threads_link;

    Q_NEW_LINK(tcb) scheduler_queue; /* Link for queues in scheduler */
    Q_NEW_LINK(tcb) tid2tcb_queue; /* Link for hashmap of TCB queues */
    Q_NEW_LINK(tcb) owning_task_thread_list; /* Link for TCB queue in PCB */
    status_t status; /* Thread's status */
    pcb_t *owning_task; /* PCB of process that owns this thread */
    uint32_t tid; /* Thread ID */

    /* Stack info. Needed for resuming execution.
    * General purpose registers, program counter
    * are stored on stack pointed to by esp. */
    uint32_t *kernel_esp; /* needed for context switch */
    uint32_t *kernel_stack_hi; /* Highest _writable_ address in kernel stack */
                               /* that is stack aligned */
    uint32_t *kernel_stack_lo; /* Lowest _writable_ kernel stack address */
                               /* that is stack aligned */

    /* Info for syscalls */
    uint32_t sleep_expiry_date;
};
#endif /* TASK_MANAGER_INTERNAL_H_ */
```

```
/** @file tests.c
 *  Set of tests for user/kern interactions */

#include <tests.h>

#include <asm.h>        /* outb() */
#include <assert.h>
#include <simics.h>
#include <logger.h>
#include <physalloc.h>  /* physalloc_test() */
#include <interrupt_defines.h>  /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <lib_thread_management/mutex.h>
#include <memory_manager.h>
#include <x86/cr.h>
#include <x86/page.h>

/* These definitions have to match the ones in user/progs/test_suite.c */
#define MULT_FORK_TEST  0
#define MUTEX_TEST      1
#define PHYSALLOC_TEST  2
#define PD_CONSISTENCY  3

static volatile int total_sum_fork = 0;
static volatile int total_sum_mux = 0;
static mutex_t mux;

/* Init is run once, before any syscalls can be made */
void
init_tests( void )
{
    mutex_init(&mux);
}

int
mult_fork_test()
{
    log_info("Running mult_fork_test");

    /* Give threads enough time to context switch */
    for (int i=0; i < 1 << 24; ++i)
        total_sum_fork++;

    log_info("SUCCESS, mult_fork_test");
    return 0;
}

int
mutex_test()
{
    log_info("Running mutex_test");

    mutex_lock(&mux);
    int old_total_sum = total_sum_mux;
    for (int i=0; i < 1 << 24; ++i)
        total_sum_mux++;
    if (total_sum_mux != old_total_sum + (1 << 24)) {
        log_info("FAIL, mutex_test.");
        return -1;
    }
    mutex_unlock(&mux);

    log_info("SUCCESS, mutex_test");
    return 0;
}

void
test_pd_consistency( void )
{
    lprintf("testing pd_consistency");
    affirm(is_valid_pd((void *)TABLE_ADDRESS(get_cr3())));
}


int
test_int_handler( int test_num )
{
    /* Acknowledge interrupt and return */
    outb(INT_CTL_PORT, INT_ACK_CURRENT);

    switch (test_num) {
        case MULT_FORK_TEST:
            return mult_fork_test();
            break;
        case MUTEX_TEST:
            return mutex_test();
            break;
        case PHYSALLOC_TEST:
            test_physalloc();
            return 0;
        case PD_CONSISTENCY:
            test_pd_consistency();
            return 0;
    }

    return 0;
}

/** @brief Installs the test() interrupt handler
 */
int
install_test_handler( int idt_entry, asm_wrapper_t *asm_wrapper )
{
    if (!asm_wrapper) {
        return -1;
    }
    init_tests();
    int res = install_handler_in_idt(idt_entry, asm_wrapper, DPL_3, D32_TRAP);
    return res;
}
```

```
#ifndef TESTS_H_
#define TESTS_H_

#include <stdint.h> /* uint32_t */
#include <install_handler.h> /* asm_wrapper_t */

int test_int_handler( int test_num );
int install_test_handler( int idt_entry, asm_wrapper_t *asm_wrapper );
void call_test_int_handler( void );

#endif /* TESTS_H_ */
```

```c
/** @file timer_driver.c
 *  @brief Contains functions that implement the timer driver
 *
 *  The PC timer rate is 1193182 Hz. The timer is configured to generate
 *  interrupts every 10 ms. and so we round off to an interrupt every
 *  11932 clock cycles. (which is more accurate then rounding down to
 *  11931 clock cycles.
 *
 *  @author Nicklaus Choo (nchoo)
 */

#include <interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <asm.h> /* outb() */
#include <assert.h> /* assert() */
#include <stddef.h> /* NULL */
#include <timer_defines.h> /* TIMER_SQUARE_WAVE */

/** @brief Get 8 least significant bits in x. */
#define LSB(x) ((long unsigned int)x & 0xFF)

/** @brief Get 8 most significant bits in x. */
#define MSB(x) (((long unsigned int)x >> 8) & 0xFF)

/* 1 khz. */
#define DESIRED_TIMER_RATE 1000

/* Initialize tick to NULL */
static void (*application_tickback) (unsigned int) = NULL;

/* Total ticks caught */
static unsigned int total_ticks = 0;

unsigned int
get_total_ticks( void )
{
    return total_ticks;
}

/**********************************************************************/
/*                                                                    */
/* Internal helper functions                                          */
/*                                                                    */
/**********************************************************************/

/** @brief Update total number of timer interrupts received and call the
 *         application provided timer callback function.
 *
 *  The interface for the application provided timer callback as written in the
 *  handout says that 'Timer callbacks should run "quickly"', and so
 *  timer_int_handler() waits for the callback application_tickback() to
 *  return quickly before sending an ACK to the relevant I/O port and returning.
 *
 *  @return Void.
 */
void timer_int_handler(void) {

    /* Acknowledging the interrupt before any context switch can take place */
    uint32_t current_total_ticks = ++total_ticks;
    outb(INT_CTL_PORT, INT_ACK_CURRENT);
    application_tickback(current_total_ticks);
    return;
}

/** @brief Initializes the timer driver
 *
 *  TIMER_RATE is the clock cycles per second. To generate interrupts once
 *  every 10 ms, we generate 1 interrupt every 10/1000 s which is 1/100 s.
 *  Therefore the number of timer cycles between interrupts is:
 *
 *  TIMER_RATE cycles     1 s
 *  ----------------- x ------ = TIMER_RATE / 100 cycles
 *         s             100
 *
 *  @param tickback Application provided function for callbacks triggered by
 *         timer interrupts.
 *  @return Void.
 */
void init_timer(void (*tickback)(unsigned int)) {

  /* Set application provided tickback function */
  assert(tickback != NULL);
  application_tickback = tickback;

  outb(TIMER_MODE_IO_PORT, TIMER_SQUARE_WAVE);
  uint16_t cycles_between_interrupts = (uint16_t)(TIMER_RATE / DESIRED_TIMER_RATE);

  outb(TIMER_PERIOD_IO_PORT, LSB(cycles_between_interrupts)); // Send lsb first
  outb(TIMER_PERIOD_IO_PORT, MSB(cycles_between_interrupts)); // then msb.

  return;
}
```

```
/** @file variable_buffer.h
 *  @brief Generalized circular buffer module for a consumer producer model
 *         where producers can add to the buffer and consumers can remove from
 *         the buffer.
 *
 *  TODO add mutex support
 *
 *  In order to use the other variable buffer functions on a particular
 *  buffer instance, the first thing one needs to do is to define the buffer
 *  type with new_buf() and then initialize that particular buffer via a pointer
 *  to it supplied to init_buf().
 *
 *  For variable buffer functions that take in pointers as arguments, due to
 *  macros being expanded inline, to avoid compiler complaints of
 *  &variable_name always evaluating to 'true', the affirm() statements will
 *  affirm(ptr_name != NULL)
 *
 *  a pointer to variable_name, or else the compiler will complain that the
 *  NULL-pointer check is unecessary. On the contrary these checks are indeed
 *  necessary because variable functions to do not know if pointers passed to
 *  them are valid or not.
 *
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef _VARIABLE_BUFFER_H_
#define _VARIABLE_BUFFER_H_

#include <string.h> /* memset() */
#include <stdint.h> /* uint32_t */
#include <stddef.h> /* NULL */

/** @def new_buf
 *  @brief Generates a new structure of type BUF_TYPE, with an array of
 *         BUF_ELEM_TYPE with size BUF_LIMIT. The array is a circular array.
 *
 *  Usage: new_buf(BUF_TYPE, BUF_ELEM_TYPE, BUF_LIMIT); //create the type
 *         BUF_TYPE buffer_name; //instantiate buffer of the given type
 *
 *  @param BUF_TYPE The type of the newly generated structure
 *  @param BUF_ELEM_TYPE The type of elements in the buffer
 *  @param BUF_LIMIT Maximum capacity of the buffer
 */
#define new_buf(BUF_TYPE, BUF_ELEM_TYPE, BUF_LIMIT)\
typedef struct {\
    uint32_t limit; /* 0 < limit */\
    uint32_t size; /* 0 <= size && size < limit */\
    uint32_t first; /* if first <= last so size == last - first */\
    uint32_t last; /* else last < first so size == limit - first + last */\
    BUF_ELEM_TYPE buffer[BUF_LIMIT];\
} BUF_TYPE

/** @def init_buf(BUF_NAME, BUF_ELEM_TYPE, BUF_LIMIT)
 *  @brief Initializes the buffer by zeroing out the buffer elements and setting
 *         struct fields to correct initial values
 *
 *  Usage: init_buf(BUF_NAME, BUF_ELEM_TYPE, BUF_LIMIT);
 *
 *  @param BUF_NAME Pointer to the buffer to initialize
 *  @param BUF_ELEM_TYPE The type of elements in the buffer
 *  @param BUF_LIMIT Maximum capacity of the buffer
 */
#define init_buf(BUF_NAME, BUF_ELEM_TYPE, BUF_LIMIT) do\
{\
    /* Pointer argument checks */\
    affirm_msg((BUF_NAME) != NULL, "Variable buffer pointer cannot be NULL!");\
\
    /* Set all fields to correct values */\
    ((BUF_NAME)->limit) = BUF_LIMIT;\
    ((BUF_NAME)->size) = 0;\
    ((BUF_NAME)->first) = 0;\
    ((BUF_NAME)->last) = 0;\
\
    /* Zero out the buffer */\
    memset(((BUF_NAME)->buffer),0,((BUF_NAME)->size) * sizeof(BUF_ELEM_TYPE));\
} while (0)

/** @def buf_insert(BUF_NAME, BUF_ELEM)
 *  @brief Inserts an element into the end of the buffer
 *
 *  Usage: buf_insert(BUF_NAME, BUF_ELEM);
 *
 *  @param BUF_NAME Pointer to the buffer to insert to
 *  @param BUF_ELEM Element to insert into the buffer. Requires that this
 *         element be of correct type, else behavior is undefined
 */
#define buf_insert(BUF_NAME, BUF_ELEM) do\
{\
    /* Pointer argument checks */\
    affirm_msg((BUF_NAME) != NULL, "Variable buffer pointer cannot be NULL!");\
\
    /* Only insert into buffer if there is space to insert it */\
    if (((BUF_NAME)->size) < ((BUF_NAME)->limit)) {\
\
        /* Insert at index one after last element in array */\
        ((BUF_NAME)->buffer[(BUF_NAME)->last]) = (BUF_ELEM);\
\
        /* Update last index and increment size */\
        ((BUF_NAME)->last) = (((BUF_NAME)->last) + 1) % ((BUF_NAME)->limit);\
        ++((BUF_NAME)->size);\
    }\
} while (0)

/** @def buf_empty(BUF_NAME)
 *  @brief Simple boolean for whether buffer is empty or not
 *
 *  Usage: buf_empty(BUF_NAME);
 *
 *  Since we are trying to do the equivalent of an affirm() statement to check
 *  for invalid NULL pointer arguments in a C macro that "returns" a value,
 *  we use the comma operator, which allows us to have multiple expressions
 *  that evaluate to the last expression.
 *
 *  In our case, the comma separated expressions have type (void, bool) and
 *  since both expression sequences need matching types, we add dummy
 *  expressions '(void) (BUF_NAME)', and simply '0' to our ternary operator
 *  'return' expressions.
 *
 *  @param BUF_NAME Pointer to the variable buffer
 */
#define buf_empty(BUF_NAME)\
(\
    /* BUF_NAME non-NULL check */\
    ((BUF_NAME) != NULL) ?\
        ((void) (BUF_NAME), (BUF_NAME)->size == 0) :\
        (panic("Variable buffer pointer cannot be NULL"), 0)\
)

/** @def buf_remove(BUF_NAME, ELEM_PTR)
 *  @brief Removes the first element (element added earliest compared to all
 *         other elements) in the buffer
 *
 *  Usage: buf_remove(BUF_NAME, ELEM_PTR);
 *
```

```
 *  @param BUF_NAME Variable buffer pointer
 *  @param ELEM_PTR Pointer to store buffer element
 */
#define buf_remove(BUF_NAME, ELEM_PTR) do\
{\
    /* Pointer argument checks */\
    affirm_msg((BUF_NAME) != NULL, "Variable buffer pointer cannot be NULL");\
    affirm_msg((ELEM_PTR) != NULL, "Element pointer cannot be NULL");\
\
    /* Only remove if size is non-zero */\
    if (((BUF_NAME)->size > 0)) {\
        *(ELEM_PTR) = ((BUF_NAME)->buffer[(BUF_NAME)->first]);\
        ((BUF_NAME)->first) = ((BUF_NAME)->first + 1) % ((BUF_NAME)->limit);\
        --((BUF_NAME)->size);\
    }\
} while (0)

/** @define is_buf(BUF_NAME)
 *  @brief Invariant checks for a variable buffer that throw assertion errors
 *         on invariant violations. Does nothing if assert()'s are off.
 *
 *  Usage: is_buf(BUF_NAME);
 *
 *  @param BUF_NAME Variable buffer pointer
 */
#define is_buf(BUF_NAME) do\
{\
    /* non NULL */\
    affirm_msg((BUF_NAME) != NULL, "Variable buffer pointer cannot be NULL!");\
\
    /* size check */\
    assert(0 <= ((BUF_NAME)->size) && ((BUF_NAME)->size) <\
           ((BUF_NAME)->limit));\
\
    /* limit check */\
    assert(0 < ((BUF_NAME)->limit));\
\
    /* first and last check */\
    if (((BUF_NAME)->first) <= ((BUF_NAME)->last)) {\
        assert(((BUF_NAME)->size) == ((BUF_NAME)->last) -\
               ((BUF_NAME)->first));\
    } else {\
        assert(((BUF_NAME)->size) == ((BUF_NAME)->limit) -\
               ((BUF_NAME)->first) + ((BUF_NAME)->last));\
    }\
    /* first and last in bounds check */\
    assert(0 <= ((BUF_NAME)->first) && ((BUF_NAME)->first) <\
           ((BUF_NAME)->limit));\
    assert(0 <= ((BUF_NAME)->last) && ((BUF_NAME)->last) <\
           ((BUF_NAME)->limit));\
} while (0)

#endif /* _VARIABLE_BUFFER_H_ */
```

```
/** @file variable_queue.h
 *
 *  @brief Generalized queue module for data collection.
 *
 *  This header file is not placed in inc/ since it is private to the kernel's
 *  own modules.
 *
 *  @author Nicklaus Choo (nchoo)
 */

#ifndef _VARIABLE_QUEUE_H_
#define _VARIABLE_QUEUE_H_

#include <stddef.h> /* NULL */
#include <assert.h> /* panic, affirm_msg */

/** @def Q_NEW_HEAD(Q_HEAD_TYPE, Q_ELEM_TYPE)
 *
 *  @brief Generates a new structure of type Q_HEAD_TYPE representing the head
 *  of a queue of elements of type Q_ELEM_TYPE.
 *
 *  Usage: Q_NEW_HEAD(Q_HEAD_TYPE, Q_ELEM_TYPE); //create the type <br>
 *         Q_HEAD_TYPE headName; //instantiate a head of the given type
 *
 *  This queue must only be manipulated by one thread at any one time
 *
 *  @param Q_HEAD_TYPE the type you wish the newly-generated structure to have.
 *
 *  @param Q_ELEM_TYPE the type of elements stored in the queue.
 *         Q_ELEM_TYPE must be a structure.
 *
 **/
#define Q_NEW_HEAD(Q_HEAD_TYPE, Q_ELEM_TYPE) \
typedef struct {\
    struct Q_ELEM_TYPE *front;\
    struct Q_ELEM_TYPE *tail;\
} Q_HEAD_TYPE;

/** @def Q_NEW_LINK(Q_ELEM_TYPE)
 *
 *  @brief Instantiates a link within a structure, allowing that structure to be
 *         collected into a queue created with Q_NEW_HEAD.
 *
 *  Usage: <br>
 *  typedef struct Q_ELEM_TYPE {<br>
 *  Q_NEW_LINK(Q_ELEM_TYPE) LINK_NAME; //instantiate the link <br>
 *  } Q_ELEM_TYPE; <br>
 *
 *  A structure can have more than one link defined within it, as long as they
 *  have different names. This allows the structure to be placed in more than
 *  one queue simultanteously.
 *
 *  index 0 is the previous pointer, index 1 is the next pointer
 *
 *  @param Q_ELEM_TYPE the type of the structure containing the link
 **/
#define Q_NEW_LINK(Q_ELEM_TYPE) \
struct {\
    struct Q_ELEM_TYPE *prev;\
    struct Q_ELEM_TYPE *next;\
}

/** @def Q_INIT_HEAD(Q_HEAD)
 *
 *  @brief Initializes the head of a queue so that the queue head can be used
 *         properly.
 *  @param Q_HEAD Pointer to queue head to initialize
 **/
#define Q_INIT_HEAD(Q_HEAD) do\
{\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    (Q_HEAD)->front = NULL;\
    (Q_HEAD)->tail = NULL;\
} while(0)

/** @def Q_INIT_ELEM(Q_ELEM, LINK_NAME)
 *
 *  @brief Initializes the link named LINK_NAME in an instance of the structure
 *         Q_ELEM.
 *
 *  Once initialized, the link can be used to organized elements in a queue.
 *
 *  @param Q_ELEM Pointer to the structure instance containing the link
 *  @param LINK_NAME The name of the link to initialize
 **/
#define Q_INIT_ELEM(Q_ELEM, LINK_NAME) do\
{\
    affirm_msg((Q_ELEM) != NULL,\
               "Variable queue element pointer cannot be NULL!");\
    ((Q_ELEM)->LINK_NAME).prev = NULL;\
    ((Q_ELEM)->LINK_NAME).next = NULL;\
} while(0)

/** @def Q_INSERT_FRONT(Q_HEAD, Q_ELEM, LINK_NAME)
 *
 *  @brief Inserts the queue element pointed to by Q_ELEM at the front of the
 *         queue headed by the structure Q_HEAD.
 *
 *  The link identified by LINK_NAME will be used to organize the element and
 *  record its location in the queue. Requires that Q_INIT_ELEM() be called on
 *  Q_ELEM prior to insertion.
 *
 *  @param Q_HEAD Pointer to the head of the queue into which Q_ELEM will be
 *         inserted
 *  @param Q_ELEM Pointer to the element to insert into the queue
 *  @param LINK_NAME Name of the link used to organize the queue
 *  @return Void (you may change this if your implementation calls for a
 *                return value)
 **/
#define Q_INSERT_FRONT(Q_HEAD, Q_ELEM, LINK_NAME) do\
{\
    /* Pointer argument checks */\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    affirm_msg((Q_ELEM) != NULL,\
               "Variable queue element pointer cannot be NULL!");\
    affirm_msg(((Q_ELEM)->LINK_NAME).next == NULL,\
               "Variable queue element pointer next must be NULL!");\
    affirm_msg(((Q_ELEM)->LINK_NAME).prev == NULL,\
               "Variable queue element pointer prev must be NULL!");\
\
    /* Q is currently empty, insert tail as well */\
    if ((Q_HEAD)->front == NULL) {\
        affirm_msg((Q_HEAD)->tail == NULL,\
                   "Empty variable queue head and tail must be both NULL");\
        (Q_HEAD)->front = (Q_ELEM);\
        (Q_HEAD)->tail = (Q_ELEM);\
\
    /* Q not empty, update front */\
    } else {\
        (((Q_HEAD)->front)->LINK_NAME).prev = (Q_ELEM);\
        ((Q_ELEM)->LINK_NAME).next = (Q_HEAD)->front;\
        (Q_HEAD)->front = (Q_ELEM);\
```

```
    }\
} while(0)

/** @def Q_INSERT_TAIL(Q_HEAD, Q_ELEM, LINK_NAME)
 *  @brief Inserts the queue element pointed to by Q_ELEM at the end of the
 *         queue headed by the structure pointed to by Q_HEAD.
 *
 *  The link identified by LINK_NAME will be used to organize the element and
 *  record its location in the queue.
 *
 *  @param Q_HEAD Pointer to the head of the queue into which Q_ELEM will be
 *         inserted
 *  @param Q_ELEM Pointer to the element to insert into the queue
 *  @param LINK_NAME Name of the link used to organize the queue
 *
 *  @return Void (you may change this if your implementation calls for a
 *                return value)
 **/
#define Q_INSERT_TAIL(Q_HEAD, Q_ELEM, LINK_NAME) do\
{\
    /* Pointer argument checks */\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    affirm_msg((Q_ELEM) != NULL,\
               "Variable queue element pointer cannot be NULL!");\
    affirm_msg(((Q_ELEM)->LINK_NAME).next == NULL,\
               "Variable queue element pointer next must be NULL!");\
    affirm_msg(((Q_ELEM)->LINK_NAME).prev == NULL,\
               "Variable queue element pointer prev must be NULL!");\
\
    /* Q is currently empty, insert front as well */\
    if ((Q_HEAD)->tail == NULL) {\
        affirm_msg((Q_HEAD)->front == NULL,\
                   "Empty variable queue head and tail must be both NULL");\
        (Q_HEAD)->front = (Q_ELEM);\
        (Q_HEAD)->tail = (Q_ELEM);\
\
    /* Q not empty, update tail */\
    } else {\
        (((Q_HEAD)->tail)->LINK_NAME).next = (Q_ELEM);\
        ((Q_ELEM)->LINK_NAME).prev = (Q_HEAD)->tail;\
        (Q_HEAD)->tail = (Q_ELEM);\
    }\
} while(0)


/** @def Q_GET_FRONT(Q_HEAD)
 *
 *  @brief Returns a pointer to the first element in the queue, or NULL
 *  (memory address 0) if the queue is empty.
 *
 *  @param Q_HEAD Pointer to the head of the queue
 *  @return Pointer to the first element in the queue, or NULL if the queue
 *          is empty
 **/
#define Q_GET_FRONT(Q_HEAD)\
(\
    /* Q_HEAD non-NULL check */\
    ((Q_HEAD) != NULL) ?\
        ((void) (Q_HEAD), (Q_HEAD)->front) :\
        (panic("Variable queue head pointer cannot be NULL"), NULL)\
)

/** @def Q_GET_TAIL(Q_HEAD)
 *
 *  @brief Returns a pointer to the last element in the queue, or NULL
 *  (memory address 0) if the queue is empty.
 *
 *  @param Q_HEAD Pointer to the head of the queue
 *  @return Pointer to the last element in the queue, or NULL if the queue
 *          is empty
 **/
#define Q_GET_TAIL(Q_HEAD)\
(\
    /* Q_HEAD non-NULL check */\
    ((Q_HEAD) != NULL) ?\
        ((void) (Q_HEAD), (Q_HEAD)->tail) :\
        (panic("Variable queue head pointer cannot be NULL"), NULL)\
)

/** @def Q_GET_NEXT(Q_ELEM, LINK_NAME)
 *
 *  @brief Returns a pointer to the next element in the queue, as linked to by
 *         the link specified with LINK_NAME.
 *
 *  If Q_ELEM is not in a queue or is the last element in the queue,
 *  Q_GET_NEXT should return NULL.
 *
 *  @param Q_ELEM Pointer to the queue element before the desired element
 *  @param LINK_NAME Name of the link organizing the queue
 *
 *  @return The element after Q_ELEM, or NULL if there is no next element
 **/
#define Q_GET_NEXT(Q_ELEM, LINK_NAME)\
(\
    /* Q_ELEM non-NULL check */\
    ((Q_ELEM) != NULL) ?\
        ((void) (Q_ELEM), ((Q_ELEM)->LINK_NAME).next) :\
        (panic("Variable queue element pointer cannot be NULL"), NULL)\
)

/** @def Q_GET_PREV(Q_ELEM, LINK_NAME)
 *
 *  @brief Returns a pointer to the previous element in the queue, as linked to
 *         by the link specified with LINK_NAME.
 *
 *  If Q_ELEM is not in a queue or is the first element in the queue,
 *  Q_GET_NEXT should return NULL.
 *
 *  @param Q_ELEM Pointer to the queue element after the desired element
 *  @param LINK_NAME Name of the link organizing the queue
 *
 *  @return The element before Q_ELEM, or NULL if there is no next element
 **/
#define Q_GET_PREV(Q_ELEM, LINK_NAME) \
(\
    /* Q_ELEM non-NULL check */\
    ((Q_ELEM) != NULL) ?\
        ((void) (Q_ELEM), ((Q_ELEM)->LINK_NAME).prev) :\
        (panic("Variable queue element pointer cannot be NULL"), NULL)\
)

/** @def Q_INSERT_AFTER(Q_HEAD, Q_INQ, Q_TOINSERT, LINK_NAME)
 *
 *  @brief Inserts the queue element Q_TOINSERT after the element Q_INQ
 *         in the queue.
 *
 *  Inserts an element into a queue after a given element. If the given
 *  element is the last element, Q_HEAD should be updated appropriately
 *  (so that Q_TOINSERT becomes the tail element)
 *
 *  @param Q_HEAD head of the queue into which Q_TOINSERT will be inserted
 *  @param Q_INQ  Element already in the queue
 *  @param Q_TOINSERT Element to insert into queue
```

```
 *  @param LINK_NAME  Name of link field used to organize the queue
 **/

#define Q_INSERT_AFTER(Q_HEAD,Q_INQ,Q_TOINSERT,LINK_NAME) do\
{\
    /* Pointer argument checks */\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    affirm_msg((Q_INQ) != NULL,\
               "Variable queue element pointer in queue cannot be NULL!")\
    affirm_msg((Q_TOINSERT) != NULL,\
               "Variable queue element pointer to insert cannot be NULL!")\
\
    ((Q_TOINSERT)->LINK_NAME).prev = Q_INQ;\
    ((Q_TOINSERT)->LINK_NAME).next = ((Q_INQ)->LINK_NAME).next;\
    ((Q_INQ)->LINK_NAME).next = Q_TOINSERT;\
\
    /* Insert at tail of queue */\
    if ((Q_INQ) == (Q_HEAD)->tail) {\
        (Q_HEAD)->tail = Q_TOINSERT;\
\
    /* Not at tail of queue, update child's prev */\
    } else {\
        affirm_msg((Q_TOINSERT)->(LINK_NAME).next != NULL,\
                   "Variable queue element pointer's next cannot be NULL!")\
        ((((Q_TOINSERT)->LINK_NAME).next)->LINK_NAME).prev = Q_TOINSERT;\
    }\
} while(0)


/** @def Q_INSERT_BEFORE(Q_HEAD, Q_INQ, Q_TOINSERT, LINK_NAME)
 *
 *  @brief Inserts the queue element Q_TOINSERT before the element Q_INQ
 *         in the queue.
 *
 *  Inserts an element into a queue before a given element. If the given
 *  element is the first element, Q_HEAD should be updated appropriately
 *  (so that Q_TOINSERT becomes the front element)
 *
 *  @param Q_HEAD head of the queue into which Q_TOINSERT will be inserted
 *  @param Q_INQ  Element already in the queue
 *  @param Q_TOINSERT Element to insert into queue
 *  @param LINK_NAME  Name of link field used to organize the queue
 **/

#define Q_INSERT_BEFORE(Q_HEAD,Q_INQ,Q_TOINSERT,LINK_NAME) do\
{\
    /* Pointer argument checks */\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    affirm_msg((Q_INQ) != NULL,\
               "Variable queue element pointer in queue cannot be NULL!")\
    affirm_msg((Q_TOINSERT) != NULL,\
               "Variable queue element pointer to insert cannot be NULL!")\
\
    ((Q_TOINSERT)->LINK_NAME).next = Q_INQ;\
    ((Q_TOINSERT)->LINK_NAME).prev = ((Q_INQ)->LINK_NAME).prev;\
    ((Q_INQ)->LINK_NAME).prev = Q_TOINSERT;\
\
    /* Insert at front of queue */\
    if ((Q_INQ) == (Q_HEAD)->front) {\
        (Q_HEAD)->front = Q_TOINSERT;\
\
    /* Not at front of queue, update ancestor's next */\
    } else {\
        affirm_msg((Q_TOINSERT)->(LINK_NAME).prev != NULL,\
                   "Variable queue element pointer's prev cannot be NULL!")\
        ((((Q_TOINSERT)->LINK_NAME).prev)->LINK_NAME).next = Q_TOINSERT;\
    }\
} while(0)

/** @def Q_IN_SOME_QUEUE(Q_ELEM,LINK_NAME)
 *
 *  @brief Checks if Q_ELEM is in some queue for a given link, LINK_NAME.
 *
 *  @param Q_ELEM Pointer to the element to remove from the queue organized by
 *         LINK_NAME.
 *  @param LINK_NAME Link name for link to check for queue.
 *  */
#define Q_IN_SOME_QUEUE(Q_ELEM,LINK_NAME)\
(\
    /* Q_HEAD non-NULL check */\
    ((Q_ELEM) != NULL) ?\
        ((void) (Q_ELEM), ((Q_ELEM)->LINK_NAME).next || ((Q_ELEM)->LINK_NAME).prev)
 :\
        (panic("Q_IN_SOME_QUEUE: Q_ELEM parameter cannot be NULL"), 0)\
)

/** @def Q_REMOVE(Q_HEAD,Q_ELEM,LINK_NAME)
 *
 *  @brief Detaches the element Q_ELEM from the queue organized by LINK_NAME,
 *         and returns a pointer to the element.
 *
 *  If Q_HEAD does not use the link named LINK_NAME to organize its elements or
 *  if Q_ELEM is not a member of Q_HEAD's queue, the behavior of this macro
 *  is undefined.
 *
 *  @param Q_HEAD Pointer to the head of the queue containing Q_ELEM. If
 *         Q_REMOVE removes the first, last, or only element in the queue,
 *         Q_HEAD should be updated appropriately.
 *  @param Q_ELEM Pointer to the element to remove from the queue headed by
 *         Q_HEAD.
 *  @param LINK_NAME The name of the link used to organize Q_HEAD's queue
 *
 *  @return Void (if you would like to return a value, you may change this
 *                specification)
 **/
#define Q_REMOVE(Q_HEAD,Q_ELEM,LINK_NAME) do\
{\
    /* Pointer argument checks */\
    affirm_msg((Q_HEAD) != NULL,\
               "Variable queue head pointer cannot be NULL!");\
    affirm_msg((Q_ELEM) != NULL,\
               "Variable queue element pointer cannot be NULL!");\
\
    /* If Q_ELEM is only element */\
    if (((Q_ELEM) == (Q_HEAD)->front) && ((Q_ELEM) == (Q_HEAD)->tail)) {\
        (Q_HEAD)->front = NULL;\
        (Q_HEAD)->tail = NULL;\
\
    /* If Q_ELEM is at the front */\
    } else if ((Q_ELEM) == (Q_HEAD)->front) {\
        (Q_HEAD)->front = ((Q_ELEM)->LINK_NAME).next;\
        affirm_msg((Q_HEAD)->front != NULL,\
                   "Variable queue front pointer cannot be NULL!");\
        (((Q_HEAD)->front)->LINK_NAME).prev = NULL;\
\
    /* If Q_ELEM is at the tail */\
    } else if ((Q_ELEM) == (Q_HEAD)->tail) {\
        (Q_HEAD)->tail = ((Q_ELEM)->LINK_NAME).prev;\
        affirm_msg((Q_HEAD)->tail != NULL,\
                   "Variable queue tail pointer cannot be NULL!");\
        (((Q_HEAD)->tail)->LINK_NAME).next = NULL;\
\
```

```
    /* Q_ELEM is somewhere in the middle */\
    } else {\
        affirm_msg((((Q_ELEM)->LINK_NAME).next) != NULL,\
                   "Variable queue element pointer next cannot be NULL");\
        ((((Q_ELEM)->LINK_NAME).next)->LINK_NAME).prev = \
            (((Q_ELEM)->LINK_NAME).prev);\
\
        affirm_msg((((Q_ELEM)->LINK_NAME).prev) != NULL,\
                   "Variable queue element pointer prev cannot be NULL");\
        ((((Q_ELEM)->LINK_NAME).prev)->LINK_NAME).next = \
            (((Q_ELEM)->LINK_NAME).next);\
    }\
    /* Make sure to remove aliases of next and prev */\
    ((Q_ELEM)->LINK_NAME).next = NULL;\
    ((Q_ELEM)->LINK_NAME).prev = NULL;\
} while(0)

#endif /* _VARIABLE_QUEUE_H_ */
```