```
/**

@mainpage 15-410 Project 3

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

Same drill as last time.

Ideally, this is not a place to say which functions appear
in which files, or to narrate what code does step by step.
Please use most of the README to document design decisions
that you made.

*/
```

```
/** @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_

#include "./timer_driver.h"
#include "./keybd_driver.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
 */

.globl call_timer_int_handler

call_timer_int_handler:
  pusha /* Pushes all registers onto the stack */
  call timer_int_handler /* calls timer interrupt handler */
  popa /* Restores all registers onto the stack */
  iret /* Return to procedure before interrupt */

.globl call_keybd_int_handler

call_keybd_int_handler:
  pusha /* Pushes all registers onto the stack */
  call keybd_int_handler /* calls timer interrupt handler */
  popa /* Restores all registers onto the stack */
  iret /* Return to procedure before interrupt */
```

```c
/** @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> /* All console function prototypes */
#include <video_defines.h> /* CONSOLE_HEIGHT, CONSOLE_WIDTH */
#include <string.h> /* memmove () */
#include <assert.h> /* assert(), affirm() */
#include <x86/asm.h> /* outb() */

/* 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

/** @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.
 *
 *  @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 )
{
    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));
        return 0;
    }
    /* cursor location is invalid, do nothing and return -1 */
    assert(onscreen(cursor_row, cursor_col));
    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 )
{
    affirm(row);
    affirm(col);
    *row = cursor_row;
    *col = cursor_col;
}

/** @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 )
{
    /* If row or col out of range, invalid row, no effect. */
    if (!onscreen(row, col)) {
        return;
    }
    /* If background color not supported, invalid color, no effect. */
    if (color & INVALID_COLOR) {
        return;
    }
    /* All arguments valid, draw character with specified color */
    char *chp = (char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col));
    *chp = ch;
    *(chp + 1) = color;
}

/** @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));
}
```

```
/** @file console_driver.c
 *  @brief Contains console driver functions implementint the p1kern interface,
 *         and other helper functions not in the interface as well.
 *
 *  For interface functions as declared in p1kern.h, additional comment blocks
 *  are preceeded and succeeded by '--'.
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs
 */

#include <limits.h> /* INT_MAX */
#include <ctype.h> /* isprint() */
#include <p1kern.h> /* client interface functions */
#include <stddef.h> /* NULL */
#include <assert.h> /* assert() */
#include <x86/asm.h> /* outb() */
#include <string.h> /* memmove () */

/* 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

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

/** @brief Checks if row and col are within console screen dimensions.
 *
 *  @param row Row to check
 *  @param col Col to check
 *  @return value greater than 1 if true, 0 otherwise
 */
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.
 */
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);

  return;
}

/** @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.
 */
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 (size_t 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 to record if there was
 *         a screen scroll of 1 line after we put a byte on the screen.
 *
 *  putbyte() is a wrapper around this function. Useful for readline() in
 *  the keyboard driver.
 *
 *  @param ch The character to print
 *  @param scrolled Must be 0. Set to 1 if scrolled a line, unchanged otherwise
 *  @return The input character
 */
int
_putbyte(char ch, int *scrolled) {
  assert(scrolled != NULL);
  assert(*scrolled == 0);
  assert(onscreen(cursor_row, cursor_col));

  /* Get current cursor position */
  char ogch = ch;
  int row, col;
  get_cursor(&row, &col);

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

      /* Scroll if at screen bottom */
      if (cursor_row + 1 >= CONSOLE_HEIGHT) {
        scroll();
        *scrolled = 1;
      }
      /* Always update the cursor position relative to content */
      set_cursor(cursor_row + 1, 0);
      break;
```

```
    }
    case '\r': {
      set_cursor(cursor_row, 0);
      break;
    }
    case '\b': {

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

        /* Always draw space before moving else cursor will be 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 */
      } else {
        draw_char(cursor_row - 1, CONSOLE_WIDTH - 1, ' ', console_color);
        set_cursor(cursor_row - 1, CONSOLE_WIDTH - 1);
      }
      break;
    }
    default: {

      /* Print the character, unprintable characters return -1 */
      if (!isprint(ch)) return -1;
      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();
          *scrolled = 1;
        }
        /* Start printing below */
        set_cursor(cursor_row + 1, 0);
      } else {
        set_cursor(cursor_row, cursor_col + 1);
      }
      break;
    }
  }
  assert(onscreen(row, col));
  return ogch;
}

/*********************************************************************/
/*                                                                   */
/* Console interface functions                                       */
/*                                                                   */
/*********************************************************************/

/** @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.
 *  --
 *  Unprintable characters return -1.
 *  We move the cursors as we write.
 *  --
 *  @param ch the character to print
 *  @return The input character
 */
int putbyte(char ch) {

  /* Wrapper for _putbyte(). We don't care in putbyte() if we scrolled */
  int scrolled = 0;
  return _putbyte(ch, &scrolled);
}

/** @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 )
{
  if (len < 0 || s == NULL) return;
  for (size_t i = 0; i < len; i++) {
    char ch = s[i];
    putbyte(ch);
  }
}

/** @brief Prints character ch with the specified color
 *         at position (row, col).
 *
 *  If any argument is invalid, the function has no effect.
 *  --
 *  Note that we only draw printable characters. To draw something is to make
 *  it to be seen. If a character cannot be seen/printed it cannot be
 *  drawn and thus we do not draw unprintable characters
 *  --
 *  @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) {

  /* If row out of range, invalid row, no effect. */
  if (!(0 <= row && row < CONSOLE_HEIGHT)) return;

  /* If col out of range, invalid col, no effect. */
  if (!(0 <= col && col < CONSOLE_WIDTH)) return;

  /* If ch not printable, invalid ch, no effect. */
  if (!isprint(ch)) return;

  /* If background color not supported, invalid color, no effect. */
  if (color & INVALID_COLOR) return;
```

```
  /* All arguments valid, draw character */
  *(char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col)) = ch;
  *(char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col) + 1) = 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) {

  /* If out of range, return '\0'. */
  if (!(0 <= row && row < CONSOLE_HEIGHT)
      || !(0 <= col && col < CONSOLE_WIDTH)) {
    return '\0';
  }
  /* Else return char at row, col. */
  return *(char *)(CONSOLE_MEM_BASE + 2*(row * CONSOLE_WIDTH + col));
}

/** @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, 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) {

  /* set logical cursor */
  if (onscreen(row, col)) {
    cursor_row = row;
    cursor_col = col;

    /* If cursor is not hidden, set the hardware cursor */
    if (!cursor_hidden) set_hardware_cursor(row, col);
    return 0;
  }
  /* cursor location is invalid, do nothing and return -1 */
  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
 *  --
 *  @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) {
  if (row != NULL) *row = cursor_row;
  if (col != NULL) *col = cursor_col;
  return;
}

/** @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;
  return;
}

/** @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;
  return;
}

/** @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) {
  if (color != NULL) *color = console_color;
}

/** @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) {

  /* No effect if invalid color passed */
  if (color & INVALID_COLOR) return -1;

  /* Else set console_color */
  console_color = color;
```

```
  return 0;
}

/** @brief Clears the entire console.
 *
 * The cursor is reset to the first row and column
 *
 *  @return Void.
 */
void clear_console(void) {
  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);
}
```

```
/** Context switch facilities */

typedef struct pcb pcb_t;
typedef struct tcb tcb_t;

struct pcb {
    void *ptd_start; // Start of page table directory
    tcb_t *first_thread; // First thread in linked list

};

struct tcb {
    pcb_t *owning_task;
    tcb_t *next_thread; // Embeded linked list of threads from same process

};

/** Switch to a new process. This function takes a while to return.
 *  If thread belongs to currently running process, only registers are
 *  updated. */
void
switch_process( int tid )
{

}
```

```
/** @file console.c
 *  @brief A console driver.
 *
 *  These empty function definitions are provided
 *  so that stdio will build without complaining.
 *  You will need to fill these functions in. This
 *  is the implementation of the console driver.
 *  Important details about its implementation
 *  should go in these comments.
 *
 *  @author Harry Q. Bovik (hqbovik)
 *  @author Fred Hacker (fhacker)
 *  @bug No know bugs.
 */

#include <console.h>

//int putbyte( char ch )
//{
//  (void)ch; // placate compiler
//  return ch;
//}

//void
//putbytes( const char *s, int len )
//{
//  (void)s; // placate compiler
//  (void)len; // placate compiler
//}

//int
//set_term_color( int color )
//{
//  (void)color; // placate compiler
//  return -1;
//}

//void
//get_term_color( int *color )
//{
//  (void)color; // placate compiler
//}

//int
//set_cursor( int row, int col )
//{
//  (void)row; // placate compiler
//  (void)col; // placate compiler
//  return -1;
//}

//void
//get_cursor( int *row, int *col )
//{
//  (void)row; // placate compiler
//  (void)col; // placate compiler
//}

//void
//hide_cursor(void)
//{
//}

//void
//show_cursor(void)
//{
//}

//void
//clear_console(void)
//{
//}

//void
//draw_char( int row, int col, int ch, int color )
//{
//  (void)row; // placate compiler
//  (void)col; // placate compiler
//  (void)ch; // placate compiler
//  (void)color; // placate compiler
//}

//char
//get_char( int row, int col )
//{
//  (void)row; // placate compiler
//  (void)col; // placate compiler
//  return ' ';
//}
```

```
/** @file game.c
 *  @brief A kernel with timer, keyboard, console support which serves mainly
 *         as a big file of test functions.
 *
 *  This file contains the kernel's main() function.
 *
 *  It sets up the drivers and starts the game. It contains test code to
 *  exercise the kernel
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */
#include <assert.h>
#include "../spec/p1kern.h"

/* libc includes. */
#include <stdio.h>
#include <simics.h>                 /* lprintf() */
#include <malloc.h>

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

/* memory includes. */
#include <lmm.h>                    /* lmm_remove_free() */

/* x86 specific includes */
#include <x86/seg.h>                /* install_user_segs() */
#include <x86/interrupt_defines.h>  /* interrupt_setup() */
#include <x86/asm.h>                /* enable_interrupts() */

#include <string.h>

volatile static int __kernel_all_done = 0;

/* Think about where this declaration
 * should be... probably not here!
 */
void tick(unsigned int numTicks) {
  //lprintf("numTicks: %d\n", numTicks);
}

void test_scroll(void) {
  for (size_t i = 0; i < CONSOLE_HEIGHT + 1; i++) {
    printf("%d\n", i);
  }
}
void test_putbyte(void) {
  for (size_t i = 0; i < CONSOLE_HEIGHT; i++) {
    draw_char(i, 1, (i % 10) + '0', FGND_RED);
  }
  printf("a");
}

/** @brief Runs a few assert statements to test draw_char() and
 *         get_char() functions
 *
 *
 */
void test_draw_char_get_char(void) {
  lprintf("Testing draw_char() and get_char()");
  draw_char(0,0, 'A', FGND_RED | BGND_BLACK);
  assert(get_char(0,0) == 'A');

  draw_char(0, CONSOLE_WIDTH - 1, 'B', FGND_WHITE | BGND_BLUE);
  assert(get_char(0, CONSOLE_WIDTH - 1) == 'B');

  draw_char(CONSOLE_HEIGHT - 1, CONSOLE_WIDTH - 1, 'C', FGND_RED | BGND_GREEN);
  assert(get_char(CONSOLE_HEIGHT - 1, CONSOLE_WIDTH - 1) == 'C');

  draw_char(CONSOLE_HEIGHT - 1, 0, 'D', FGND_YLLW | BGND_CYAN);
  assert(get_char(CONSOLE_HEIGHT - 1, 0) == 'D');

  /* Offscreen row drawing has no effect */
  draw_char(CONSOLE_HEIGHT, 0, 'E', FGND_YLLW | BGND_CYAN);
  assert(get_char(CONSOLE_HEIGHT, 0) == '\0');

  /* Offscreen col drawing has no effect */
  draw_char(CONSOLE_HEIGHT - 1, CONSOLE_WIDTH, 'E', FGND_YLLW | BGND_CYAN);
  assert(get_char(CONSOLE_HEIGHT - 1, CONSOLE_WIDTH) == '\0');

  /* Invalid background color drawing has no effect */
  draw_char(CONSOLE_HEIGHT - 1, 0, 'F', FGND_YLLW | 0x190);
  assert(get_char(CONSOLE_HEIGHT - 1, 0) == 'D');

  /* Invalid unprintable character drawing has becomes ? */
  draw_char(CONSOLE_HEIGHT - 1, 0, '\6', FGND_YLLW | BGND_RED);
  assert(get_char(CONSOLE_HEIGHT - 1, 0) == 'D');

  lprintf("Passed draw_char() and get_char()");
  return;
}

void test_cursor(void) {

  lprintf("Testing: set_cursor()");
  /* Out of bounds checks */
  assert(set_cursor(CONSOLE_HEIGHT, 0) == -1);
  assert(set_cursor(-1, 0) == -1);
  assert(set_cursor(0, CONSOLE_WIDTH) == -1);
  assert(set_cursor(0, -1) == -1);

  /* In bounds checks */
  assert(set_cursor(0, 0) == 0);
  assert(set_cursor(0, CONSOLE_WIDTH -1) == 0);
  assert(set_cursor(CONSOLE_HEIGHT - 1, CONSOLE_WIDTH - 1)
         == 0);
  assert(set_cursor(CONSOLE_HEIGHT - 1, 0)
         == 0);
  assert(set_cursor(0, 0) == 0);
  lprintf("Passed: set_cursor()");
}


/** @brief Kernel entrypoint.
 *
 *  This is the entrypoint for the kernel.  It simply sets up the
 *  drivers and passes control off to game_run().
 *
 * @return Does not return
 */
int kernel_main(mbinfo_t *mbinfo, int argc, char **argv, char **envp)
{
    /*
     * Initialize device-driver library.
     */
    int res = handler_install(tick);
    lprintf("res of handler_install: %d", res);

    /*
     * When kernel_main() begins, interrupts are DISABLED.
     * You should delete this comment, and enable them --
     * when you are ready.
     *
```

```
     */
    printf("h");

    lprintf( "Hello from a brand new kernel!" );


    char * badguy = (char *) 0xdeadd00d;
    char * s = "hello";
    lprintf("badguy: 0x%08lx, s: 0x%08lx", (long) badguy, (long) s);
    //putbytes((char *), 3);


    test_draw_char_get_char();
    //test_putbyte();

    test_cursor();

    //clear_console();

    //printf("Hello Mom!");
    //
    test_scroll();

    clear_console();

    putbytes("carriage return should bring the cursor to the front of this line.\r\
n",67);

    int tohide = 0;
    while (!__kernel_all_done) {
      if (tohide) hide_cursor();
      else show_cursor();
       int n =  CONSOLE_HEIGHT * CONSOLE_WIDTH;
       char s[n];
       int res = readline(s, n);
       lprintf("characters read: %d, '%s'",res, s);
       putbytes("how many characters to read next",33);
       res = readline(s, n);
       lprintf("characters read: %d, '%s'",res, s);


       tohide = !tohide;

      continue;
    }

    return 0;
}
```

```
/** @ file console_driver.h
 *  @ brief Contains helpful internal functions for other drivers to use.
 *          Mainly the keyboard driver
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */
#ifndef _P1_CONSOLE_DRIVER_H_
#define _P1_CONSOLE_DRIVER_H_

int _putbyte(char ch, int *scrolled);

#endif
```

```
/*
 *
 *  #     #
 *  ##    #   ####    #####     #     ####   ######
 *  # #   #  #    #     #       #    #    #  #
 *  #  #  #  #    #     #       #    #       #####
 *  #   # #  #    #     #       #    #       #
 *  #    ##  #    #     #       #    #    #  #
 *  #     #   ####      #       #     ####   ######
 *
 * 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>

/** @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 _INSTALL_HANDLER_H_
#define _INSTALL_HANDLER_H_
int handler_install(void (*tickback)(unsigned int));
#endif
```

```
/** @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>

void init_keybd(void);
void keybd_int_handler(void);
int readline(char *buf, int len);

typedef int aug_char;
typedef uint8_t raw_byte;

/** @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
```

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

#ifndef _LOADER_H
#define _LOADER_H


/* --- Prototypes --- */

int getbytes( const char *filename, int offset, int size, char *buf );
int execute_user_program( const char *fname );
/*
 * Declare your loader prototypes here.
 */

#endif /* _LOADER_H */
```

```
/** @ file mem_manager.h
 *  @ brief Contains helpful internal functions for other drivers to use.
 *          Mainly the keyboard driver
 *
 *  @author Nicklaus Choo (nchoo)
 *  @bug No known bugs.
 */
#ifndef _P1_MEM_MANAGER_H_
#define _P1_MEM_MANAGER_H_

void paging_on( void );
void paging_off( void );

int vm_init( void );


#endif
```

```
/** @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));

#endif
```

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

#include <x86/asm.h> /* idt_base() */
#include <assert.h> /* assert() */
#include <x86/interrupt_defines.h>
#include <x86/timer_defines.h> /* TIMER_IDT_ENTRY */
#include <x86/seg.h> /* SEGSEL_KERNEL_CS */
#include <x86/keyhelp.h>
#include <stddef.h> /* NULL */
#include <simics.h> /* lprintf */
#include "./asm_interrupt_handler.h" /* call_timer_int_handler(),
                                        call_keybd_int_handler() */
#include <timer_driver.h> /* init_timer() */
#include <keybd_driver.h> /* init_keybd() */
#include <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
#define D16                 0x00000700
#define D32                 0x00000F00
#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);

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

/** @brief Installs an interrupt handler at idt_entry for timer and keyboard
 *         interrupt handlers.
 *
 *  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 handler_install_in_idt(int idt_entry, asm_wrapper_t *asm_wrapper,
                           void (*tickback)(unsigned int)) {

  if (asm_wrapper == NULL) return -1;

  /* Only when installing the timer handler do we have a non-NULL tickback */
  if (tickback != NULL) {
    init_timer(tickback);
  } else {
    init_keybd();
  }
  /* Get address of trap gate for timer */
  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_0 | D32;

  /* 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;
}

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

/** @brief The driver-library initialization function
 *
 *   Installs the timer and keyboard interrupt handler.
 *   NOTE: handler_install should ONLY install and activate the
 *   handlers; any application-specific initialization should
 *   take place elsewhere.
 *   --
 *   After installing both the timer and keyboard handler successfully,
 *   interrupts are enabled.
 *   --
 *   @param tickback Pointer to clock-tick callback function
 *   @return A negative error code on error, or 0 on success
```

```
 **/
int handler_install(void (*tickback)(unsigned int)) {

  /* While interrupt handlers are not set up, disable interrupts */
  disable_interrupts();

  /* Initialize and install timer handler */
  int res = handler_install_in_idt(TIMER_IDT_ENTRY, call_timer_int_handler,
                                   tickback);
  if (res < 0) return E_NO_INSTALL_TIMER_HANDLER;

  /* Initialize and install keyboard handler */
  res = handler_install_in_idt(KEY_IDT_ENTRY, call_keybd_int_handler, NULL);
  if (res < 0) return E_NO_INSTALL_KEYBOARD_HANDLER;

  /* Interrupt handlers successfully installed, enable interrupts */
  enable_interrupts();
  return 0;
}
```

```
/** @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 <install_handler.h> /* handler_install() */
#include <common_kern.h>

/* libc includes. */
#include <stdio.h>
#include <simics.h>                 /* lprintf() */

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

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

#include <x86/cr.h> /* get_cr3() */

#include <console.h> /* clear_console(), putbytes() */
#include <keybd_driver.h> /* readline() */
#include <loader.h> /* execute_user_program() */
#include <mem_manager.h> /* vm_init() */

volatile static int __kernel_all_done = 0;

/* Think about where this declaration
 * should be... probably not here!
 */
void tick(unsigned int numTicks) {
  //lprintf("numTicks: %d\n", numTicks);
}

/** @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)
{
    // placate compiler
    (void)mbinfo;
    (void)argc;
    (void)argv;
    (void)envp;

    /* initialize device-driver library */
    int res = handler_install(tick);
    lprintf("res of handler_install: %d", res);



    clear_console();

    /*
     * When kernel_main() begins, interrupts are DISABLED.
     * You should delete this comment, and enable them --
     * when you are ready.
     */

    lprintf( "Hello from a brand new kernel!" );
    putbytes("executable user programs:\n", 26);
    putbytes("loader_test1\n", 13);
    putbytes("loader_test2\n", 13);
    putbytes("getpid_test1\n", 13);

    /* On kernel_main() entry, all control registers are 0 */
    lprintf("cr1: %p", (void *) get_cr3());
    lprintf("cr2: %p", (void *) get_cr3());
    lprintf("cr3: %p", (void *) get_cr3());
    lprintf("cr4: %p", (void *) get_cr3());
    char * nullp = 0;
    lprintf("garbage at address 0x0:%d", *nullp);
    lprintf("&nullp:%p", &nullp);
    vm_init();


    while (!__kernel_all_done) {
        int n =  CONSOLE_HEIGHT * CONSOLE_WIDTH;
        char s[n];

        /* Display prompt */
        putbytes("pebbles>",8);
        int res = readline(s, n);
        lprintf("read %d bytes: \"%s\"", res, s);
        res = execute_user_program(s);
    }

    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 <x86/keyhelp.h> /* process_scancode() */
#include <x86/video_defines.h> /* CONSOLE_HEIGHT, CONSOLE_WIDTH */
#include <malloc.h> /* calloc */
#include <stddef.h> /* NULL */
#include <assert.h> /* assert() */
#include <console.h> /* putbyte() */
#include <string.h> /* memcpy() */
#include <x86/asm.h> /* process_scancode() */
#include <x86/interrupt_defines.h> /* INT_CTL_PORT */
#include <ctype.h> /* isprint() */
#include "./console_driver.h" /* _putebyte() */
#include "./keybd_driver.h" /* uba */

/* Keyboard buffer */
static uba *key_buf = NULL;

int readchar(void);

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

/** @brief Checks invariants for unbounded arrays
 *
 *  The invariants here are implemented as asserts so in the even that
 *  an assertion fails, the developer knows exactly which assertion
 *  fails. There are many invariants since on top of being an
 *  unbounded array, it is also a circular array.
 *
 *  @param arr Pointer to uba to be checked
 *  @return 1 if valid uba pointer, 0 otherwise
 */
int is_uba(uba *arr) {

  /* non-NULL check */
  assert(arr != NULL);

  /* size check */
  assert(0 <= arr->size && arr->size < arr->limit);

  /* limit check */
  assert(0 < arr->limit);

  /* first and last check */
  if (arr->first <= arr->last) {
    assert(arr->size == arr->last - arr->first);
  } else {
    assert(arr->size == arr->limit - arr->first + arr->last);
  }
  /* first and last in bounds check */
  assert(0 <= arr->first && arr->first < arr->limit);
  assert(0 <= arr->last && arr->last < arr->limit);

  /* data non-NULL check */
  assert(arr->data != NULL);
  return 1;
}

/** @brief Initializes a uba and returns a pointer to it
 *
 *  Fatal errors are only thrown if size == 0 or size == 1 and yet
 *  init_uba() returns NULL (out of heap space)
 *
 *  @param type Element type in the uba
 *  @param limit Actual size of the uba.
 *  @return Pointer to valid uba if successful, NULL if malloc() or
 *          calloc() fails or if limit < 0
 */
uba *uba_new(int limit) {
  if (limit <= 0 ) return NULL;

  /* Convert limit to unsigned */
  uint32_t _limit = (uint32_t) limit;
  assert(0 <= _limit);

  /* Allocate memory for uba struct */
  uba *new_ubap = malloc(sizeof(uba));
  assert(new_ubap != NULL);
  if (new_ubap == NULL) return NULL;
```

```
  /* Allocate memory for the array */
  void *data = NULL;
  data = calloc(_limit, sizeof(uint8_t));
  assert(data != NULL);
  if (data == NULL) return NULL;

  /* Set fields of unbounded array uba */
  new_ubap->size = 0;
  new_ubap->limit = _limit;
  new_ubap->first = 0;
  new_ubap->last = 0;
  new_ubap->data = data;

  return new_ubap;
}

/** @brief Resizes the unbounded array
 *
 *  @param arr Pointer to a uba
 *  @param Pointer to the original uba if no resize needed, bigger uba o/w.
 */
uba *uba_resize(uba *arr) {
  assert(is_uba(arr));
  if (arr->size == arr->limit) {

    /* Only resize if sufficient */
    if (arr->size < UINT32_MAX / 2) {
      uba *new_arr = uba_new(arr->size * 2);
      assert(is_uba(new_arr));

      /* Copy elements over */
      while(arr->size > 0) {
        uint8_t elem = uba_rem(arr);
        uba_add(new_arr, elem);
      }
      /* Free the old array */
      free(arr->data);
      free(arr);

      /* Return new array */
      return new_arr;
    }
    /* at limit but cannot resize, OK for now but error will be thrown
     * if want to add a new element to arr
     */
  }
  return arr;
}

/** @brief Adds new element to the array and resizes if needed
 *
 *  @param arr Unbounded array pointer we want to add to
 *  @param elem Element to add to the unbounded array
 *  @return Void.
 */
void uba_add(uba *arr, uint8_t elem) {
  assert(is_uba(arr));
  assert(arr->size < arr->limit); /* Total memory is 256 MiB and since
                                     each array element is 1 byte we
                                     will never reach max possible limit
                                     of 2^32 */

  /* Insert at index one after last element in array */
  arr->data[arr->last] = elem;

  /* Update last index and increment size*/
  arr->last = (arr->last + 1) % arr->limit;
  arr->size += 1;
  assert(is_uba(arr));

  /* Resize if necessary */
  arr = uba_resize(arr);
  assert(is_uba(arr));
}

/** @brief Removes first character of the uba
 *  @param arr Pointer to uba
 *  @return First character of the uba
 */
uint8_t uba_rem(uba *arr) {
  assert(is_uba(arr));

  /* Get first element and 'remove' from the array, decrement size */
  uint8_t elem = arr->data[arr->first];
  arr->first = (arr->first + 1) % arr->limit;
  arr->size -= 1;
  assert(is_uba(arr));

  /* Return 'popped off' element */
  return elem;
}

/** @brief Checks if uba is empty
 *  @param arr Pointer to uba
 *  @return 1 if empty, 0 otherwise.
 */
int uba_empty(uba *arr) {
  assert(is_uba(arr));
  return arr->size == 0;
}

/** @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);
  uba_add(key_buf, raw_byte);

  /* Acknowledge interrupt and return */
  outb(INT_CTL_PORT, INT_ACK_CURRENT);
}

/** @brief Initialize the keyboard interrupt handler and associated data
 *         structures
 *
 *  Memory for keybd_buf is allocated here.
 *
 *  @return Void.
 */
void init_keybd(void) {

  /* Intialize the raw_byte buffer */
  key_buf = uba_new(CONSOLE_HEIGHT * CONSOLE_WIDTH);
  assert(key_buf != NULL);
}

/** @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                                     */
/*                                                               */
/*****************************************************************/

/** @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) {

  assert(key_buf != NULL);

  /* 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.
   */
  disable_interrupts();
  if (uba_empty(key_buf)) {
    enable_interrupts();
    return -1;
  }
  raw_byte next_byte = uba_rem(key_buf);
  enable_interrupts();

  /* Get augmented character */
  aug_char next_char = process_scancode(next_byte);

  /* Get simplified character */
  if (KH_HASDATA(next_char)) {
    if (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 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;

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

  /* Allocate space 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;
}
```

```
/**
 * 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.
 */
/*@{*/

/* --- Includes --- */
#include <string.h> /* memset() */
#include <stdio.h>
#include <malloc.h>
#include <exec2obj.h>
#include <loader.h>
#include <elf_410.h>
#include <assert.h> /* assert() */
#include <simics.h> /* lprintf() */
/* --- Local function prototypes --- */

///* Format of entries in the table of contents. */
//typedef struct {
//  const char execname[MAX_EXECNAME_LEN];
//  const char* execbytes;
//  int execlen;
//} exec2obj_userapp_TOC_entry;
//
///* The number of user executables in the table of contents. */
//extern const int exec2obj_userapp_count;
//
///* The table of contents. */
//extern const exec2obj_userapp_TOC_entry exec2obj_userapp_TOC[MAX_NUM_APP_ENTRIES]
;
//


/**
 * 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 returns the number of bytes copied on succes; -1 on failure
 */
int getbytes( const char *filename, int offset, int size, char *buf )
{
    if (!filename) {
        return -1;
    }
    if (!buf) {
        return -1;
    }
    const char *current = filename;
    current += offset;
    int i = 0;
    while (i < size) {
        lprintf("i:%d", i);
        assert(*(current + i));
        *(buf + i) = *(current + i);
        i++;
    }
    /* Quick test for correct copying */
    for (int j = 0; j < size; j++) {
        assert(*(filename + offset + j) == *(buf + j));
    }
    return i;
}

int
copy2physical( simple_elf_t *se_hdr )
{
    /* copy text segment into physical memory */
    lprintf("e_entry:%lx, e_txtstart:%lx, e_datstart:%lx, e_rodatstart:%lx, "
            "e_bssstart:%lx", se_hdr->e_entry, se_hdr->e_txtstart,
            se_hdr->e_datstart, se_hdr->e_rodatstart, se_hdr->e_bssstart);

    /* I feel like buf is physical memory. No buff is virtual memory */
    getbytes(se_hdr->e_fname,
             (unsigned int) se_hdr->e_txtoff,
             (unsigned int) se_hdr->e_txtlen,
             (char *) se_hdr->e_txtstart);

    getbytes(se_hdr->e_fname,
             (unsigned int) se_hdr->e_datoff,
             (unsigned int) se_hdr->e_datlen,
             (char *) se_hdr->e_datstart);

    getbytes(se_hdr->e_fname,
             (unsigned int) se_hdr->e_rodatoff,
             (unsigned int) se_hdr->e_rodatlen,
             (char *) se_hdr->e_rodatstart);

    return 0;
}


/** @brief Given user program name such as './getpid', loads the program and
  *        runs it
  */

int
execute_user_program( const char *fname )
{
    /* Load user program into helper struct */
    simple_elf_t se_hdr;
    memset(&se_hdr, 0, sizeof(simple_elf_t));
    int res = elf_load_helper(&se_hdr, fname);
    if(res == ELF_SUCCESS) lprintf("ELF_SUCCESS");
    if (res == ELF_NOTELF) lprintf("ELF_NOTELF");

    /* we try to use the physical addresses */
    copy2physical(&se_hdr);
    return 0;
}


/* we try to use the physical addresses */

/*@}*/
```

```
#include <stddef.h>
#include <malloc.h>
#include <malloc_internal.h> /* _malloc family of functions */

/* safe versions of malloc functions */
void *malloc(size_t size)
{
  return _malloc(size);
}

void *memalign(size_t alignment, size_t size)
{

  return _memalign(alignment, size);
}

void *calloc(size_t nelt, size_t eltsize)
{
  return  _calloc(nelt, eltsize);
}

void *realloc(void *buf, size_t new_size)
{
    return _realloc(buf, new_size);
}

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

void *smalloc(size_t size)
{
  return _smalloc(size);
}

void *smemalign(size_t alignment, size_t size)
{
  return smemalign(alignment, size);
}

void sfree(void *buf, size_t size)
{
    sfree(buf, size);
}
```

```
/**
 * Virtual memory manager
 *
 *
 * */
#include <simics.h> /* lprintf() */
#include <stdint.h>
#include <stddef.h>
#include <malloc.h>
#include <elf/elf_410.h>
#include <x86/cr.h> /* set_cr0() */
#include <common_kern.h> /* machine_phys_frame() */
#include <string.h> /* memset() */

#define PAGE_ENTRY_SIZE 64
#define PAGE_TABLE_SIZE (1024 * PAGE_ENTRY_SIZE)

#define PAGE_DIRECTORY_INDEX 0xFFC00000
#define PAGE_TABLE_INDEX 0x003FF000
#define PAGE_OFFSET 0x00000FFF

int num_page_frames = 0;

#define NUM_ENTRIES 1024
/* This is 4KByte = 4 * 1024 Byte total */
void *PAGE_TABLE[NUM_ENTRIES];

/* Page directory and page table coincidentally have
 * the same size. */
#define PAGE_DIRECTORY_SIZE PAGE_TABLE_SIZE

/** lmm and malloc are responsible for managing kernel virtual
 *  memory. However, that virtual memory must still be associated
 *  to some physical memory (which we must direct map)
 *
 *  For user memory, however, we must manage physical pages starting
 *  at USER_MEM_START up to USER_MEM_START + machine_phys_frames() * PAGE_SIZE.
 *  TODO: Currently no free, but in the future some data structure should manage
 *  this.
 *
 * To ensure page tables/directories are aligned, always use
 *  smemalign() and sfree() * */

/* FIXME: Temporary variable for enabling allocation of physical frames.
 *        Only to be used for user memory. Starts at USER_MEM_START, where
 *        the first phys frames are available. */
static void *next_free_phys_frame;

#define PAGING_OFF 0x7FFFFFFF
#define PAGING_ON 0x80000000
void
paging_on( void )
{
    uint32_t current_cr0 = get_cr0();
    set_cr0(current_cr0 | PAGING_ON);
}

void
paging_off( void )
{
    uint32_t current_cr0 = get_cr0();
    set_cr0(current_cr0 & PAGING_OFF);
}

/** Initialize virtual memory */
int
vm_init( void )
{
    next_free_phys_frame = (void *) (uint32_t) USER_MEM_START;
    num_page_frames = machine_phys_frames();
    memset(PAGE_TABLE, 0, sizeof(void *) * NUM_ENTRIES);
    lprintf("PAGE_TABLE address:%p", &PAGE_TABLE);

    /* Need to have a page directory here */
    return 0;
}

/** Allocate new pages in a given process' virtual memory. */
/* Currently only 1 task supported, so no need to index based on tid */
int
vm_new_pages ( void *ptd_start, void *base, int len )
{
    return -1;
}

/** Allocate memory for new task at given page table directory.
 *
 *  TODO: Create a subroutine suitable for cloning. */
int
vm_new_task ( void *ptd_start, simple_elf_t *elf )
{
    /* Allocate phys frames and assign to virtual memory */

    /* Allocate rodata, text with read-only permissions */

    /* Allocate code with execute permissions */

    /* Allocate data with read-write permissions */

    return -1;
}

/** Translate logical address into physical address.
 *
 *  Note: This can be used by loader to "transplant"
 *  data from elf binary into a new task's virtual
 *  memory. */
void *
translate( void *logical )
{
    // TODO: logical -> linear -> physical
    return NULL;
}
```

```
/** Context switch facilities */

#include <stdint.h>
#include <stddef.h>
#include <malloc.h>

// TODO: Have this be defined in an internal .h file
// and be shared with mem_manager.c
#define PAGE_ENTRY_SIZE 64
#define PAGE_TABLE_SIZE (1024 * PAGE_ENTRY_SIZE)

/* Page directory and page table coincidentally have
 * the same size. */
#define PAGE_DIRECTORY_SIZE PAGE_TABLE_SIZE

typedef struct pcb pcb_t;
typedef struct tcb tcb_t;

struct pcb {
    void *ptd_start; // Start of page table directory
    tcb_t *first_thread; // First thread in linked list

};

struct tcb {
    pcb_t *owning_task;
    tcb_t *next_thread; // Embeded linked list of threads from same process

};

pcb_t pcb_list_start;

/* Create new process (memory + pcb and tcb) */
int
new_task( simple_elf_t *elf )
{
    // TODO: Affirm kernel_mode

    /* TODO: Paging might be enabled, check for that and disable it. */

    // TODO: Add to pcb_list, for now support 1 task (just set pcb_list_start)

    /*-- Checkpoint 1 impl start --*/
    pcb_list_start->ptd_start = malloc(
}
```

```
/** @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 <x86/interrupt_defines.h> /* INT_CTL_PORT, INT_ACK_CURRENT */
#include <x86/asm.h> /* outb() */
#include <assert.h> /* assert() */
#include "stddef.h" /* NULL */
#include <x86/timer_defines.h> /* TIMER_SQUARE_WAVE */

#define INTERRUPT 100
#define SHORT_LSB_MASK 0x00FF
#define SHORT_MSB_MASK 0xFF00

/* Initialize tick to NULL */
static void (*application_tickback) (unsigned int) = NULL;

/* Total ticks caught */
static unsigned int total_ticks = 0;

/**********************************************************************/
/*                                                                    */
/* 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) {

  /* Update total ticks */
  total_ticks += 1;

  /* Pass total ticks to application callback which should run quickly */
  application_tickback(total_ticks);

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

  assert(tickback != NULL);

  outb(TIMER_MODE_IO_PORT, TIMER_SQUARE_WAVE);
  short cycles_between_interrupts = (short)(TIMER_RATE / INTERRUPT);

  /* Round off */
  if ((TIMER_RATE % INTERRUPT) > (INTERRUPT / 2)) {
    cycles_between_interrupts += 1;

    assert(((cycles_between_interrupts - 1) * INTERRUPT) +
        (TIMER_RATE % INTERRUPT) == TIMER_RATE);
  } else {
    assert((cycles_between_interrupts * INTERRUPT) +
        (TIMER_RATE % INTERRUPT) == TIMER_RATE);
  }
  /* Send the least significant byte */
  short lsb =  cycles_between_interrupts & SHORT_LSB_MASK;
  outb(TIMER_PERIOD_IO_PORT, lsb);

  /* Send the most significant byte */
  short msb = (cycles_between_interrupts & SHORT_MSB_MASK) >> 8;
  outb(TIMER_PERIOD_IO_PORT, msb);

  /* Set application provided tickback function */
  application_tickback = tickback;

  return;
}
```