IOT Report

Ali BAYDOUN

March 2025

1 First Steps

First of all, i will understand and take notes regarding the initial files provided to use, that way i can use the notes i took as documentation in case i forget what a certain option does in the MakeFile for example.

I don't think i have to do this for the linker (kernel.ld), as it is already documented so i can just read the comments in the code in case needed.

2 The Makefile

The provided Makefile is used to automate the process of compiling, linking, and running our project for an ARM-based system. It defines compilation rules, handles dependencies, and sets up debugging options. Below is a breakdown of its key components.

2.1 Configurable Parameters

At the beginning of the Makefile, several variables are defined :

- `BOARD=versatile`: Specifies the target board. In this case, it is a "Versatile" board, which is an ARM-based development platform. (Documentation)
- `CPU=cortex-a8`: Defines the target CPU architecture, ensuring the correct instruction set is used. (Documentation)
- `TOOLCHAIN=arm-none-eabi`: Indicates the toolchain to be used for cross-compilation. Since the project targets an ARM processor, an appropriate cross-compiler is required.
- `DEBUG?=yes`: Enables or disables debugging features. If set to yes, additional flags are added for debugging.
- `BUILD=build/`: Specifies the directory where compiled output files will be stored.
- `objs= startup.o main.o exception.o uart.o`: Lists the object files that will be generated from their corresponding source files.

2.2 Handling Object Files

The Makefile constructs the list of object files dynamically using:

```
OBJS = $(addprefix $(BUILD), $(objs))
```

This prepends the build directory path to each object file name, ensuring they are placed in the correct location.

2.3 Conditional Compilation Based on Board

The Makefile includes conditional logic to configure compilation options based on the selected board. If the board is "versatile" (which it is in our case), it defines:

- QEMU-related settings:

```
VGA=-nographic
SERIAL=-serial mon:stdio
```

  These settings configure QEMU to run without a graphical interface and redirect serial output to the console.

- Memory settings:

```
MEMSIZE=32
MEMORY="$(MEMSIZE)K"
```

  The project is configured to allocate 32 KB of memory, this can easily be changed by modifying the MEMSIZE in the make file if we every need more memory size for our "kernel".

- QEMU machine type:

```
MACHINE=versatileab
```

2.4 Compiler Flags (`CFLAGS`)

The `CFLAGS` variable contains options passed to the compiler when compiling C source files. The flags used in this Makefile are:

- `-c`: Tells the compiler to generate an object file (`.o`) instead of a complete executable.
- `-mcpu=$(CPU)`: Specifies the target CPU architecture.
- `-nostdlib`: Excludes the standard C library from the build process.
- `-ffreestanding`: Indicates that the code does not depend on a hosted environment (i.e., an operating system). So the software we are running is directly used "above" the hardware.
- `-DCPU=$(CPU)`: Defines a preprocessor macro `CPU` with the value specified in the Makefile.
- `-DMEMORY="($(MEMSIZE)*1024)"`: Defines a preprocessor macro `MEMORY` representing the memory size in bytes. This macro is used in the main.c file of our code to calculate the memory size.

If debugging is enabled (`DEBUG=yes`), the following additional flags are included:

- `-ggdb`: Generates debugging information that can be used by GDB.

2.5 Assembler Flags (`ASFLAGS`)

The `ASFLAGS` variable contains options passed to the assembler when assembling assembly source files:

- `-mcpu=$(CPU)`: Ensures the assembler generates code compatible with the specified ARM processor.

If debugging is enabled, the following flag is also included:

- `-g`: Includes debugging information in the assembled output to allow debugging at the assembly level.

2.6 Linker Flags (LDFLAGS)

The LDFLAGS variable contains options passed to the linker when linking object files into an executable:

- `-T kernel.ld`: Specifies the linker script to use, which will define the memory layout and how sections of the program (such as text, data, and stack) are organized.
- `-nostdlib`: Ensures the linker does not link against the standard C library.
- `-static`: Forces static linking, meaning no dynamic libraries are used.

If debugging is enabled, the following flag is also included:

- `-g`: Embeds debugging information into the final executable, allowing tools like GDB to perform source-level debugging.

2.7 Compilation Rules

The Makefile defines explicit rules for compiling source files:

- Compiling C files:

```
$(BUILD)%.o: %.c
    $(TOOLCHAIN)-gcc $(CFLAGS) -o $@ $<
```

  This means any C source file (`%.c`) is compiled into an object file (`%.o`) and placed in the `BUILD` directory.

- Assembling assembly files:

```
$(BUILD)%.o: %.s
    $(TOOLCHAIN)-as $(ASFLAGS) -o $@ $<
```

  This rule processes assembly source files in a similar manner.

2.8 Building and Linking the Kernel

To create the final executable, the Makefile defines the `all` target which ensures that:

- The `build` directory is created.
- The kernel is linked into an ELF executable.
- The ELF file is converted into a binary file.

The ELF file is built then converted into a binary format with:

```
$(BUILD)kernel.bin: $(BUILD)kernel.elf
    $(TOOLCHAIN)-objcopy -O binary $(BUILD)kernel.elf $(BUILD)kernel.bin
```

2.9 Creating the Build Directory

Since all compiled files are placed in the `build` directory, it must be created before compilation starts. The `build` target does this:

```
build:
    @mkdir $(BUILD)
```

2.10 Cleaning Up

To remove all compiled files and reset the project, the `clean` target is used:

```
clean:
    rm -rf $(BUILD)
```

This ensures that a fresh compilation is performed the next time the Makefile is run.

2.11 Running the code

```
run: all
    @echo "\n\nBoard: Versatile Board...\n"
    $(QEMU) $(QEMU_ARGS) -device loader,file=$(BUILD)kernel.elf
```

This compiles the project and runs it in QEMU.

2.12 Debugging with GDB

To debug the kernel, the `debug` target is used:

```
debug: all
    @echo "\n\nBoard: Versatile Board...\n"
    $(QEMU) $(QEMU_ARGS) -device loader,file=$(BUILD)kernel.elf -gdb tcp::1234 -S
```

This starts QEMU in debugging mode, waiting for a connection on TCP port 1234. The `-S` option ensures that execution is halted until the debugger is attached.

3 Handling Keyboard Inputs

The first thing I will try to do is receive inputs from the keyboard and display them on the screen.

To achieve this, i spent some time reading about UART, which stands for Universal Asynchronous Receiver/Transmitter. It is a simple, two-wire protocol for exchanging serial data, commonly used for communication between a computer and embedded systems. In this project, i will use UART as a hardware communication protocol to send and receive data between the system and a serial terminal.

I found the base addresses of UART0, UART1, and UART2, which are the three available UART interfaces, in the documentation of the Versatile board here. I used these addresses to define the corresponding base values in the `uart-mmio.h` file.

Additionally, I found detailed information about the PL011 UART registers in the ARM documentation here. This documentation provides the offset, name, type, and detailed descriptions of each register. For now, I will focus on two key registers: `UARTDR` and `UARTFR`, as they are essential for basic input and output operations.

`UARTDR` (UART Data Register) is used for reading received data and writing data to be transmitted. This is the primary register for sending and receiving characters.

`UARTFR` (UART Flag Register) contains various status flags that indicate the current state of the UART. The flags I will be using are:

- **TXFF (Transmit FIFO Full)**: Indicates whether the transmit FIFO buffer is full. If this flag is set, no more data can be written to `UARTDR` until space becomes available.
- **RXFE (Receive FIFO Empty)**: Indicates whether the receive FIFO buffer is empty. If this flag is set, there is no data available to read from `UARTDR`.

- **TXFE (Transmit FIFO Empty)**: Indicates whether the transmit FIFO buffer is empty. This is useful for checking if all outgoing data has been sent.
- **BUSY**: Indicates whether the UART is currently transmitting data. This flag can be used to ensure the UART has finished its operation before sending new data.

To check if a specific flag is set in the code, I need to use a bitmask operation. Each flag corresponds to a specific bit position in the `UARTFR` register. For example, the `RXFE` flag is located at bit position 4. To check whether this bit is set, I create a mask using `(1 << 4)`, which results in the binary value `0b00010000`. I then perform a bitwise AND operation between this mask and the value of the `UARTFR` register. If the result is nonzero, it means the flag is set; otherwise, it is cleared.

Then, for receiving data, I need to check if the receive FIFO buffer is not empty by examining the `RXFE` flag in the `UARTFR` register. This is done using a bitwise AND operation. If the flag is not set, it means data is available, and I can read the received character from the `UARTDR` register and store it in the pointer passed to the `receive` function.

For sending data, I will check if the transmit FIFO buffer is not full by examining the `TXFF` flag. If the buffer has space available (i.e., `TXFF` is not set), I can write the character to be sent into the `UARTDR` register. This ensures that the data is properly queued for transmission without overwriting any ongoing transmissions.

3.1 Update for Week 2

Regarding the registers mentioned in Section 3, I ultimately used only `TXFF` and `RXFE`, as the other registers were not necessary for my current implementation.

4 Implementing Interrupts

4.1 Setting up the registers

To begin implementing interrupts, I first identified the Vector Interrupt Controller (VIC) used by the board. The Versatile board features the PL190 VIC, as listed in the board's supported peripherals documentation here.

The base address for the VIC is `0xFFFFF000`, according to the VIC programming model. Additionally, I noted the offsets of key registers that may be used:

- `VICIRQSTATUS` (`0x000`) – Indicates which interrupt sources are currently active and enabled for IRQ handling.
- `VICFIQSTATUS` (`0x004`) – Shows which interrupts are active and configured as FIQ (Fast Interrupt Request).
- `VICRAWINTR` (`0x008`) – Displays all active interrupt sources before any masking is applied.
- `VICINTSELECT` (`0x00C`) – Configures each interrupt source as either IRQ or FIQ.
- `VICINTENABLE` (`0x010`) – Enables specific interrupt sources by setting corresponding bits.
- `VICINTENCLEAR` (`0x014`) – Disables specific interrupts by clearing corresponding bits.

I also added the corresponding IRQ addresses to `isr.h`, allowing me to define and manage different interrupt sources. The file now includes the following IRQ definitions:

- **UART Interrupts:** These correspond to the three UART controllers on the board.
  - `UART0_IRQ` (IRQ 12) – Mask: `(1 << UART0_IRQ)`

- `UART1_IRQ` (IRQ 13) – Mask: `(1 << UART1_IRQ)`
- `UART2_IRQ` (IRQ 14) – Mask: `(1 << UART2_IRQ)`

- **Timer Interrupts:** These correspond to the system timers.
  - `TIMER3_IRQ` (IRQ 5) – Mask: `(1 << TIMER3_IRQ)`
  - `TIMER2_IRQ` (IRQ 5) – Mask: `(1 << TIMER2_IRQ)`
  - `TIMER1_IRQ` (IRQ 4) – Mask: `(1 << TIMER1_IRQ)`
  - `TIMER0_IRQ` (IRQ 4) – Mask: `(1 << TIMER0_IRQ)`

4.2 Implementing Interrupt Handling Functions

The process of enabling interrupts in the system was a multi-step approach that involved modifications at both the assembly and C levels, as well as updates to the linker script. Below is are the steps i did in order to achieve it:

4.2.1 Assembly Modifications and Interrupt Vector Setup

To start with, I modified the assembly startup code and exception handler to ensure that interrupts could be properly handled the C code:

- **Enabling IRQs in Startup:** Originally, the startup code disabled IRQ interrupts using a CPSR flag. I removed the disablement of the IRQ (by omitting the `CPSR_VIC_FLAG`) in the following line:

  ```
  msr        cpsr_c,#(CPSR_SYS_MODE | CPSR_FIQ_FLAG),
  ```

  This change allows IRQ interrupts to be enabled from the start, enabling the system to respond to them immediately.

- **Integrating the ISR Call:** I modified the exception handler in `exception.s` to call the C-level `isr()` function. The updated IRQ handler looks like:

  ```
  _isr_handler:
      sub lr, lr, #4
      stmfd sp!, {r0-r12, lr}
      bl isr
      ldmfd sp!, {r0-r12, pc}
  ```

  This sequence ensures that when an IRQ occurs, the processor saves the necessary registers, calls `isr()` to dispatch the interrupt to the appropriate handler, and then restores the registers before resuming execution.

4.2.2 Linker Script Updates

I also updated the linker script (`kernel.ld`) to clearly define the memory layout and reserve dedicated memory for the interrupt stack. Key changes include:

- Defining the interrupt vector table within the `.text` section to include the exception handlers.
- Reserving a specific memory region for the IRQ stack:

  ```
  . = ALIGN(8);
  . = . + 0x1000; /* 4KB of stack memory */
  irq_stack_top = .;
  ```

This organization ensures proper alignment and prevents stack overflows while allowing the CPU to switch stacks upon an interrupt.

4.2.3 C-Level Interrupt Handling Functions

Once the assembly and linker modifications were in place, I implemented the interrupt handling functions in C:

- `isr()`: This is the main Interrupt Service Routine (ISR) called from assembly when an interrupt occurs. It:
  1. Disables interrupts to prevent nested calls.
  2. Reads the active interrupts from the VIC using `vic_load_irqs()`.
  3. Iterates over all potential interrupts, calling the registered callback for any active interrupt.
  4. Re-enables interrupts after processing.
- `vic_setup_irqs()`: This function initializes the VIC by disabling all interrupts and then configuring the interrupt system via `_irqs_setup()`.
- `vic_enable_irq()` **and** `vic_disable_irq()`: These functions manage individual interrupt lines. `vic_enable_irq()` registers a callback (with associated data) and enables the interrupt by setting the corresponding bit in the `VICINTENABLE` register. Conversely, `vic_disable_irq()` clears the handler and disables the interrupt.

4.2.4 Testing VIC Interrupt Enabling

To verify that the VIC-level interrupt enabling was working correctly:

- I added the following line to `main.c`:

  ```
  vic_enable_irq(UART0_IRQ, uart_receive, (UART0, &c));
  ```

- Running the code in debug mode (with a breakpoint at the start of `_start()`), I checked the initial state of the `VICINTENABLE` register:

  ```
  x/1xw 0x10140000+0x10
  ```

  The output was:

  ```
  0x10140010: 0x00000000
  ```

  This confirmed that all interrupts were initially disabled.

- After executing `vic_enable_irq()`, running the same command yielded:

  ```
  0x10140010: 0x00001000
  ```

  Converting `0x00001000` to binary confirms that the 12th bit is set to `1` (as expected for `UART0_IRQ`). This verified that the function successfully enabled the correct interrupt.

4.2.5 Enabling UART Interrupts

To allow UART0 to generate interrupts upon receiving data, I completed the UART enable and disable functions:

- The `uart_enable()` function writes to the `UART_IMSC` register to enable the `RXIM` bit.
- The `uart_disable()` function clears the `UART_IMSC` register.

To test the UART interrupt functionality:

- I ensured that UARTs were initialized with `uarts_init()`.
- Then, I invoked `uart_enable(UART0)` in `main.c`.
- Using GDB, I first checked the state of the `UART_IMSC` register:

x/1xw 0x101F1000+0x038

which initially returned:

0x101f1038: 0x00000000

- After calling `uart_enable(UART0)`, the same command yielded:

  0x101f1038: 0x00000010

  Converting `0x00000010` to binary shows that the 4th bit is set, corresponding to the `RXIM` bit being enabled. This confirmed that UART0 receive interrupts is successfully activated.

4.2.6 Putting It All Together in `main.c`

In the final integration, I updated `main.c` to tie all the previous modifications together. In this file, I initialized the UARTs and the IRQ system, enabled the UART interrupts, and set up the IRQ parameters structure. These changes were made so that the system could properly handle incoming data. When the system starts, the following occurs:

- **Stack and System Checks:** The `check_stacks()` function verifies that both the C and IRQ stacks are within the allocated memory bounds.
- **UART Initialization:** The `setup_uarts()` function initializes the UARTs by setting their base addresses and enabling UART0. A confirmation message is sent to indicate that the UART setup is complete.
- **IRQ Setup:** The `setup_irqs()` function initializes the VIC and enables the system-wide interrupts, including the UART interrupt. This is achieved by calling `vic_setup_irqs()` followed by `vic_enable_irqs()`, which registers the UART interrupt callback.
- **Runtime Behavior:** Once all the initialization is complete, the system prints a "The system is now running..." message. From this point on, the main loop halts the CPU with `core_halt()` until an interrupt occurs.

With these modifications, when a button is pressed (or data is received via UART), the hardware triggers an IRQ. The interrupt is caught by the updated assembly handler, which calls the C-level `isr()` function. The `isr()` function then dispatches the interrupt to the registered callback (in this case, `uart_interrupt()`), which processes the incoming data by reading from the UART data register and echoing the character back to the screen.

4.2.7 Taking It Further: Implementing a Minimal Shell

To extend the use of UART interrupts beyond simple input and output, I implemented a minimal shell that processes user commands. This shell uses the interrupt-driven UART input to collect characters, allowing users to enter commands. The commands I added are:

- `echo <message>`**:** Prints the given message back to the console.
- `clear`**:** Clears the terminal screen using ANSI escape sequences.
- `curs <on|off>`**:** Enables or disables the terminal cursor.
- `help`**:** Displays a list of available commands.

I also implemented basic character processing features, such as handling the backspace key (`\b` or `0x7F`) to allow users to correct their input and processing the enter key (`\r` or `\n`) to execute commands.

One issue i'm encountering during development is handling arrow key inputs. When pressing an arrow key, the system receives a sequence of three characters corresponding to ANSI escape codes (e.g., `/033[D` for the left arrow). While I was able to detect these sequences correctly, an unexpected behavior occurres: after receiving the three characters, the system would trigger an additional interrupt that causes a call to `_reset_handler`, which resets the entire system by invoking `_start`.

5 Update for Week 3

With guidance from the professor, I was able to resolve the issue related to arrow key inputs. The root of the problem was a misunderstanding on my part regarding how interrupt enabling and disabling works. Initially, I believed I needed to manually disable and re-enable interrupts within the ISR. However, the professor clarified that the processor handles this automatically when an interrupt is triggered.

Specifically, when an interrupt occurs, the processor saves the current state and disables further interrupts to safely execute the handler. At the end of the handler—more precisely, when the registers are popped back using the assembly instruction with the ^ character—the processor automatically restores the previous mode and re-enables interrupts.

By removing my manual handling of interrupt enabling and disabling, the system no longer resets when an arrow key is pressed. The interrupt is now handled properly, and the shell continues functioning without unexpected resets.

6 Shell Improvements for the Fun of It

With basic input and output working through interrupts, I decided to extend the shell for a more interactive and user-friendly experience. Now that I can detect arrow key escape sequences, I added support for left and right arrows to move the cursor within the input line. This allowed the user to edit commands at any position, not just at the end of the line. To make this work, I implemented logic that adjusts the input buffer based on the current cursor position—characters typed in the middle of a command will shift the rest of the line forward, just like in a typical shell.

Additionally, I implemented a command history feature. The shell now keeps track of the most recent commands entered by the user. Using the up and down arrow keys, it is possible to cycle through past commands. This works similarly to standard terminals: pressing the up arrow retrieves the previous command, while the down arrow moves forward in the history, eventually returning to a blank prompt. Internally, the shell maintains a circular history buffer and updates the shell state accordingly when navigating through it.

These features, although not strictly necessary for the TP, make the shell feel significantly more responsive and complete.

7 Adding Rings - Simple Version

In class, we learned that using ring buffers is a common and efficient strategy in systems programming, especially when dealing with asynchronous data such as UART input. This structure allows us to decouple the producer and consumer: the interrupt handler can quickly store incoming characters without having to immediately process them, while the main loop can read and handle them at its own pace. This avoids complex synchronization or the risk of losing characters when processing takes too long.

Initially, in the UART interrupt, I would directly read the character from the UART Data Register and immediately pass it to the shell's character handler. However, this tight coupling meant that any delay or complexity in `shell_process_char` could cause problems in the interrupt context, and that the uart and the shell are tightly coupled, so i can't use the interrupt for anything other than the shell. To improve this, I changed the UART interrupt routine so that it now writes incoming characters into a ring buffer. The buffer acts as a queue that stores characters safely until the shell is ready to handle them.

On the shell side, the main loop checks the ring after each interrupt. If the ring is not empty, it reads characters one by one using `ring_get()` and processes them with the existing shell logic.

8 Introducing Processes and Dynamic Memory Allocation

Before implementing the full version of ring buffers with read/write listeners, I wanted to introduce the concept of *processes* in my kernel. The idea was to group various attributes, such as a process's ring buffer and its associated listeners, into a single structure. To prepare for that, I started by building a minimal process abstraction that can launch functions through an entry point. My goal was to allow each process to own its own resources and have cleaner separation of concerns.

8.1 Adding Dynamic Memory Allocation

While designing the process system, I quickly felt the need for dynamic memory allocation. Unlike in userland programming where `malloc` is readily available, in kernel development we must implement our own allocator. So I decided to add a simple heap allocator to my system.

The first step was to reserve a region of memory for the heap in the linker script. I added the following to the `kernel.ld` file:

```
/* Heap region for dynamic memory allocation */
  . = ALIGN(4);
  __heap_start__ = .;
  . = . + 0x2000; /* 8KB for the heap */
  __heap_end__ = .;
```

This reserves 8KB of memory for dynamic allocation. The symbols `__heap_start__` and `__heap_end__` mark the boundaries of the heap and are used by the allocator to manage this region.

To implement allocation and freeing, I created a basic free list allocator. It tracks available blocks in a linked list and can split or merge blocks as needed. This part was heavily inspired by the CSEPC (Conception de système d'exploitation et programmation concurrente) course we had in the first year of the master's degree, where we had to implement our own memory allocation functions `mem_alloc` and `mem_free` (Link to that repository). The allocator supports basic `h_alloc` (malloc) and `h_free` (free) functions:

- `init_heap()` initializes the free list using the heap region.
- `h_alloc()` searches the free list for a sufficiently large block, splits it if possible, and returns a pointer to the usable memory.
- `h_free()` reinserts a block into the free list, coalescing with adjacent free blocks when possible to reduce fragmentation.

This simple allocator should be enough to support dynamic creation and destruction of process structures, and paves the way for more complex use cases.

8.2 Defining Processes and Launching the Shell

Once memory allocation was in place, I designed a basic process abstraction. A process in my system is represented by a `process_t` structure, which includes:

- a process ID (`pid`),
- a state (e.g. `CREATED`, `RUNNING`, `TERMINATED`),
- and a function pointer to the process's entry point.

Processes are created dynamically using `h_alloc()` and registered in a global `process_table`. Here's the core functionality:

- `process_create()` allocates memory and initializes the process structure.

- `process_add()` stores the process in the global table.
- `process_start_all()` starts all registered processes that are in the `CREATED` state.
- `process_start()` marks a process as `RUNNING`, invokes its entry point, and marks it `TERMINATED` afterward. Once done, its memory is freed.

This model mimics a real-world system where processes have distinct lifecycles. For now, my kernel runs processes sequentially, but the infrastructure could later support multitasking with a scheduler and a time quantum for each process.

To test this system, I updated the kernel entry point to create and start the shell as a process:

```
void _start(void) {
    sys_init();
    process_t* p_shell = process_create(shell_start);
    process_start(p_shell);
    panic();
}
```

After running the code, the shell behaves as expected, confirming that the process infrastructure works correctly. I can now use processes within my kernel.

P.S: I moved the loop that I had in the kernel entry point function to the `shell_start` function instead:

```
void shell_start() {
    shell_init();
    while (1) {
        shell_process();

        if (exit_shell) break;

        core_disable_irqs();
        if (ring_empty()) {
            core_halt();
        }
        core_enable_irqs();
    }
}
```

8.3 Testing Dynamic Allocation and Memory Leaks

To verify that my dynamic memory allocator (`h_alloc` and `h_free`) behaves correctly (at least for one process being allocated and freed), I used `gdb` to trace changes to the heap's internal state — specifically, the `free_list` pointer — before and after memory operations.

The goal of this test was to ensure that:

- The heap is correctly initialized.
- Allocations reduce the available free space and update `free_list` accordingly.
- Freeing a block restores the heap to its previous state and coalesces memory if possible.

To do this, I set watchpoints and breakpoints in `gdb` and stepped through the memory management lifecycle of the shell. The test procedure was as follows:

1. Set a watchpoint on the global `free_list` variable.
2. Set breakpoints at the entry of `h_alloc` and `h_free`.

3. Run the system and observe the state of `free_list` at each critical point.

Here is the actual `gdb` session that illustrates the process:

```
(gdb) target remote :1234
Remote debugging using :1234
(gdb) watch free_list
Hardware watchpoint 1: free_list
(gdb) break h_alloc
Breakpoint 1 at 0x174: file src/allocator.c, line 22.
(gdb) break h_free
Breakpoint 2 at 0x2d0: file src/allocator.c, line 64.
(gdb) continue
```

When the system initialized the heap:

```
Hardware watchpoint 1: free_list

Old value = (block_t *) 0x0
New value = (block_t *) 0x2d84
init_heap () at src/allocator.c:16
```

The `free_list` was set to the start of the heap at address `0x2d84`.

Later, when a process (the shell) was created and allocated:

```
Breakpoint 1, h_alloc (size=12) at src/allocator.c:22
(gdb) continue

Hardware watchpoint 1: free_list

Old value = (block_t *) 0x2d84
New value = (block_t *) 0x2d98
h_alloc (size=12) at src/allocator.c:42
```

Here, we see that `h_alloc` split off a block from the head of the heap, and `free_list` advanced to `0x2d98`, reflecting the remaining space.

Once the shell exited and the memory was freed:

```
Breakpoint 2, h_free (ptr=0x2d8c) at src/allocator.c:64
(gdb) continue

Hardware watchpoint 1: free_list

Old value = (block_t *) 0x2d98
New value = (block_t *) 0x2d84
h_free (ptr=0x2d8c) at src/allocator.c:93
```

The `free_list` returned to its original value (`0x2d84`), confirming that the allocator successfully coalesced the freed block with the remaining heap. This indicates that no memory was leaked in this round-trip allocation and free.

9 Adding Rings – Improved Version with Listeners

To further enhance the system's interactivity and modularity, I extended the ring buffer implementation to support listeners. The idea is to decouple the hardware interrupt handling from the processing of input data by associating a listener callback with each process. In practice, each process now has its

own ring buffer that stores incoming UART data, and when new data arrives, the registered listener is invoked to process it.

In this improved design, the process creation API was updated to include listener functions. For instance, the modified `_start` function now creates a shell process with a read listener as follows:

```
void _start(void) {
    sys_init();
    process_t* p_shell = process_create(shell_start, NULL,
        shell_read_listener, NULL);
    process_start(p_shell);
    sys_exit(0, "End of _start entry point");
}
```

Here, the shell process is created with an entry point (`shell_start`), and a read listener (`shell_read_listener`). The process structure was extended to include a context that holds a ring buffer and pointers to both read and write listener functions. This context allows the process to maintain its own input buffer and to trigger callback functions when new data is available.

Within the UART interrupt routine, instead of directly passing the read character to the shell, the character is now placed into the active process's ring buffer:

- The UART interrupt reads available characters from the UART data register.
- For each character, it checks if the active process exists and whether its ring buffer is full.
- If there is room, the character is enqueued into the ring.
- Once all characters are read, the registered read listener for the active process is called to process the buffered input.

On the shell side, the `shell_read_listener` function repeatedly reads characters from its ring buffer (using the `ring_get` function) and passes them to `shell_process_char`.

For robust error handling, I created a dedicated error handler file. The `sys_exit` function prints a message (including error codes for non-zero exits) and then calls `panic`, which disables interrupts and halts the system. This ensures that in case of critical failures (e.g., if the ring buffer becomes full), the system exits gracefully without undefined behavior.

9.1 Reflections on the Current Implementation

In the current implementation, I haven't used the `write_listener` because I'm not sure how it would behave in the context of the shell. Currently, I write directly using the `uart_send_string` function. This approach introduces a strong coupling between the UART protocol and the shell program, meaning that if we want to use a different protocol to communicate with the keyboard, the current implementation wouldn't support it.

One limitation of the current implementation is that shell execution occurs in interrupt mode, which can block other interrupts if the shell execution takes too long. However, the advantage is that since the UART calls the `read_listener` callback of the shell, the shell doesn't need to know or care who is handling the data retrieval. It simply reads from the buffer and processes the data. This means that, if I decide to switch from UART to a different protocol (e.g., SPI, I2C, or Bluetooth) to handle keyboard input, the rest of the code will still work. The new protocol would just need to collect the data, add it to the shell's ring buffer, and then trigger the `read_listener` callback, at which point the shell would process the data as needed.

That said, this flexibility doesn't currently extend to the data transmission part. The UART and shell are tightly coupled because I directly call `uart_send` within the shell. This is something I plan to address in the future to improve modularity and achieve better separation of concerns.

10 Transitioning to an Event-Based System with Tasks

In this version, I transformed the system from a traditional process model into an event-based architecture driven by tasks and event queues. This approach helps decouple task execution from direct interrupt handling and improves modularity and responsiveness.

Task Abstraction and Initialization

Instead of creating full processes, I now define tasks. Each task is represented by a structure that includes its context (which contains an ID, a ring buffer, and pointers to read/write listener functions) and an entry point function. The task is created and initialized with a call to `task_create`:

```
task_t* t_shell = task_create(shell_init, NULL,
    shell_read_listener, shell_write_listener, UART0);
```

During task initialization (`task_init`), the system:

- Assigns a unique task ID and sets the initial state.
- Initializes the task's ring buffer by setting both the head and tail pointers to zero and clearing the buffer contents.
- Registers listener functions for reading and writing.
- Associates a specific UART (via its number) with the task by calling `uart_grab`, so that subsequent UART inputs are directed to this task's ring buffer.

Task Activation and Execution

Once a task is created, it is activated via `task_activate`, which calls the task's entry point function (here, `shell_init`) with its context and any initial cookie. The task then begins its execution. In the new design, the task is not responsible for continuous polling but reacts to events.

Event Handling Loop

After task activation, the main function enters an infinite loop that continually processes events:

```
while (1) {
    const event_t* event = event_pop();
    if (event != NULL) {
        event->callback(event->cookie);
    } else {
        core_halt();
    }
}
```

- `event_pop()` checks the event queue (specifically, the "ready" queue) for pending events.
- If an event is available, its associated callback is invoked with the given cookie.
- If no events are present, the system halts (using `core_halt()`) until the next interrupt.

This loop ensures that events are processed asynchronously, allowing the kernel to react promptly to external stimuli (for example, user input via UART).

Modified UART Interrupt Handling with Listeners

The UART interrupt routine has been refactored to work with the event-driven architecture. Instead of processing the input data directly within the interrupt context, the routine now places the received characters into the ring buffer associated with the task linked to the UART. Once all characters are enqueued, an event is created and posted to trigger the task's read listener. This decoupling ensures minimal work is done in the interrupt handler and allows the main event loop to handle the processing asynchronously.

Key aspects of this approach include:

- **Task Association:** The routine checks that the UART is linked to a task context (stored in `t_context`). If no task is associated, the system exits with an error.
- **Buffered Input:** Characters received from the UART are enqueued into the task's ring buffer. This buffering allows for asynchronous processing of input data.
- **Event Posting:** Once all characters have been enqueued, an event is created with the task's read listener as its callback. This event is posted immediately (since `eta` is 0), signaling the event loop to process the input.

The corresponding read listener (for the shell in our example) continuously reads characters from the ring buffer and passes them to the shell command processor, thereby decoupling the low-level UART interrupts from the higher-level command handling logic.