Boot & Build System Documentation

This document explains the core boot process, linker configuration, and build automation of the Novix Operating System.

File: `boot/boot.c`

Function: `boot()`

This is the very first function executed when the system boots. It is marked with `__attribute__((naked))` and placed in the `.text.boot` section so that it is loaded and executed first by the processor.

Purpose The `boot()` function sets up the stack pointer and jumps directly to the `kernel_main()` function. It initializes the C runtime environment by establishing a valid stack before executing any kernel logic.

Parameters

- None — this function does not accept parameters or return values.

Return Value

- None — control is transferred directly to `kernel_main()`.

Detailed Description

1. The assembler moves the value of `__stack_top` into the stack pointer register (`sp`).
2. Execution then jumps to the address of `kernel_main` using a direct `j` (jump) instruction.
3. The use of `__attribute__((naked))` ensures no prologue or epilogue code is generated by the compiler.

This is effectively the bridge between bootloader context and the kernel C environment.

File: `kernel.ld`

Purpose

The linker script (`kernel.ld`) defines the memory layout for the kernel, describing where sections like `.text`, `.data`, `.bss`, and the stack are placed in physical memory.

Key Features

- Entry point: `boot` — execution begins at the `boot()` function.
- Kernel load address: `0x80200000`.
- Separate sections for text, rodata, data, bss, initramfs, stack, free RAM, and heap.
- Predefined global symbols exported for use in C code (e.g., `__stack_top`, `__heap_start`, `__heap_end`).

Detailed Layout

Section	Description	Alignment / Size
`.text.boot`	Boot function (first executed code)	N/A
`.text`	Kernel code and submodules	Default
`.rodata`	Read-only data such as strings	4-byte aligned
`.initramfs`	Embedded initramfs image from user binaries	8-byte aligned
`.data`	Initialized variables	4-byte aligned
`.bss`	Zero-initialized variables	4-byte aligned
`Stack`	128 KB kernel stack	4-byte aligned
`Free RAM`	64 MB reserved for dynamic memory	4096-byte aligned
`Heap`	32 MB dynamic kernel heap	4096-byte aligned

Section	Description	Alignment / Size

Special Symbols

- `__stack_top` / `__stack_bottom`: Define stack boundaries.
- `__free_ram` / `__free_ram_end`: Track available physical RAM.
- `__heap_start` / `__heap_end`: Heap area for dynamic kernel allocations.

This configuration gives the kernel full control over memory layout, ensuring that all parts of the OS are deterministic and easy to debug.

File: `run.sh`

Purpose

This shell script automates the entire build, linking, and QEMU run process for the Novix OS project. It ensures consistent compilation and reproducibility of the environment.

Main Build Steps

1. **Cleanup** — Remove old artifacts (ELF, object files, static libraries, tar archives).
2. **Compile Userland Libraries** — Common C modules such as `printf.c`, `malloc.c`, and `syscall.c` are compiled and archived into `libuser.a`.
3. **Compile Startup Code** — Low-level startup assembly (`crt0.S` and `init_crt0.S`) prepares the stack and global environment for user programs.
4. **Compile User Programs** — Each program (like `shell`, `ls`, `ps`, etc.) is compiled separately.
5. **Link User Programs** — The linker combines startup code, program object files, and `libuser.a` into `.elf` executables.
6. **Create Initramfs** — All user binaries are archived into `initramfs.tar` and converted to an ELF section to be embedded inside the kernel.
7. **Build Kernel** — Compiles and links all kernel sources into `bin/kernel.elf` using `kernel.ld`.
8. **Start QEMU** — Launches the virtual machine with proper RISC-V hardware emulation.

QEMU Parameters

- Machine: `virt`
- Memory: `512M`
- BIOS: OpenSBI firmware
- Devices: virtio-gpu, virtio-net, virtio-serial
- I/O: Serial output (`mon:stdio`), file logging, and host time socket via `socat`

Development Workflow

1. Modify kernel or user source files.
2. Run `./run.sh` — builds everything automatically.
3. QEMU boots directly into the Novix shell environment.

This provides a full end-to-end workflow for OS development.

Summary

Component	Purpose
`boot/boot.c`	Initializes stack and jumps to the kernel.
`kernel.ld`	Defines the memory layout and section placement.
`run.sh`	Automates compilation, linking, packaging, and execution.

Together, these three files form the **foundation of the Novix Operating System build pipeline** — from bootloader handoff to a running virtualized kernel environment.

File: kernel/kernel.c

Module: Core Kernel Initialization and UART Output

Overview

This file defines the **main entry point** for the Novix OS kernel (`kernel_main`) and implements a minimal **UART output driver** for early boot communication.
It provides basic I/O routines (`putc`, `puts`) used throughout the kernel before more complex subsystems are initialized.

The functions here are fundamental for **debugging, logging, and kernel startup messages**.

Function: `static inline int uart_is_writable(void)`

Purpose

Checks whether the UART (Universal Asynchronous Receiver/Transmitter) hardware is ready to accept a new character for transmission.

Parameters

- *(none)*

Return Value

- Returns **non-zero (true)** if the UART transmit register is ready (i.e., the transmitter holding register is empty).
- Returns **0 (false)** if not ready.

Detailed Description

- This function reads from the **Line Status Register (LSR)** of the UART at offset `0x05`.
- Bit `5` (mask `0x20`) of this register indicates if the transmitter holding register (THR) is empty.
- When this bit is set, the UART can accept a new byte for transmission.

```
return UART0_LSR & 0x20;
```

This simple check allows the kernel to send data over serial safely without overwriting data that is still being transmitted.

Function: `void putc(char c)`

Purpose

Sends a single character through the UART serial port.

Parameters

- `char c` — The character to be transmitted.

Return Value

- *(void)* — no return value.

Detailed Description

1. The function waits until the UART is ready to accept a new byte:

   ```
   while (!uart_is_writable());
   ```

   This creates a blocking loop that pauses execution until the UART's transmitter buffer is empty.

2. Once writable, the character is written directly to the **Transmit Holding Register (THR)**:

   ```
   UART0_THR = c;
   ```

This is a **low-level blocking transmission** method used for early boot output where timing and concurrency are not yet managed.

Function: `void puts(const char *s)`

Purpose

Outputs a null-terminated string over UART.

Parameters

- `const char *s` — Pointer to the string to send.

Return Value

- *(void)* — no return value.

Detailed Description

1. The function loops through the input string, sending each character via `putc()`.
2. For each newline character (`'\n'`), it automatically sends a carriage return (`'\r'`) before it — ensuring proper line formatting on terminals that expect CR+LF line endings:

   ```
   if (*s == '\n') putc('\r');
   putc(*s++);
   ```

3. The loop terminates once a null terminator (`'\0'`) is reached.

This ensures consistent output formatting across various terminal emulators and UART drivers.

Function: `void kernel_main(void)`

Purpose

Serves as the **entry point** for the kernel once the bootloader (`boot.c`) has set up the stack and transferred control to the kernel.

Parameters

- *(none)*

Return Value

- *(void)* — this function never returns.

Detailed Description

1. Prints the message `"hello world!\n"` using `puts()` to verify UART functionality:

   ```
   puts("hello world!\n");
   ```

2. Enters an infinite loop (`for (;;);`), halting further execution.

At this stage, the kernel is operational in a minimal state with UART verified — this forms the **base milestone** before integrating memory management, interrupts, or process control.

Summary

Function	Description
`uart_is_writable()`	Checks UART status before writing.
`putc(char c)`	Sends a single character to UART.
`puts(const char *s)`	Sends a string to UART, formatting line endings.
`kernel_main()`	Main kernel entry point; outputs a test message and halts.

Key Concepts Introduced

- UART register-level programming
- Early kernel debugging output
- Boot-to-kernel control handoff
- Minimal system initialization structure