

Chapter 3 **Booting and Initialization**

Design and Implementation of SysPak OS



Abstract

This chapter gives the description of development environment and tool chain used to build SysPak OS. The compilation process is described along with the directory structure of the source code and the resulting kernel. Booting process of SysPak OS is explained next. Then it is explained how the system is initialized, what steps are performed in subsystem initializations and how the shell comes up and start running.

3.1 Development Environment

The SysPak OS is developed on the Red Hat Linux 8.0. Tools used are listed below:

The GNU GCC Compiler (3.0.4): GCC used to stand for the GNU C Compiler, but since the compiler supports several other languages aside from C, it now stands for the GNU Compiler Collection. GCC 3.0.4 has several new optimizations, new targets, new languages and many other new features, relative to GCC 2.95.x. SysPak OS can be compiled on GCC 3.0.4 at any platform that is capable of generating elf binaries.

The GNU as Assembler (2.12.1): GNU `as` is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax. `as` is primarily intended to assemble the output of the GNU C compiler `gcc` for use by the linker `ld`.

The GNU ar (2.12.1): The GNU `ar` program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called member of the archive). The original files' contents, mode (permission), timestamp, owner, and group are preserved in the archive, and may be reconstituted on extraction. `ar` is considered a binary utility because archives of this sort are most often used as libraries holding commonly needed subroutines.

The GNU ld Linker (2.12.1): `ld` combines a number of object and archive files, relocates their data and ties up symbol reference. Often the last step in building a new compiled program to run is call to `ld`. `ld` accepts Linker Command Language files to provide control over the linking process. `ld` can read, combine and write object files in many different formats, for example COFF or a.out. Different formats may be linked together to produce any available kind of object file.

The GNU objcopy(2.12.1): this utility copies the contents of an object file to another. It can write the destination object files in a format different from that of the source object file. The exact behavior of `objcopy` is controlled by command-line options. `objcopy` is used to generate a raw binary file by using an output target of binary (e.g. use `-O binary`). When `objcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocations information will be discarded. The memory dump will start at the virtual address of the lowest section copied into the object file.

The GNU strip (2.12.1): GNU `strip` discards all symbols from the object files. The list of object files may include archives. `strip` modifies the file in its arguments, rather than writing modified copies under different names.

The GNU size (2.12.1): The GNU `size` utility lists the section sizes—and the total size—for each of the object files in its arguments. By default one line of output is generated for each object file or each module in an archive.

3.2 Building the SysPak OS Kernel Image

This section explains the steps taken during compilation of the SysPak OS kernel and the output produced at each stage.

When the user types 'make final' the resulting bootable kernel image is stored as `build/i386/final`. Here is how the image is built:

1. C and assembly source files are compiled into ELF relocatable object format (.o) and some of them are grouped logically into archives (.a) using **ar**.
2. Using **ld**, the above .o and .a are linked into `kernel` which is a statically linked, non-stripped ELF 32-bit LSB 80386 executable file.
3. Irrelevant or uninteresting symbols are gripped out by the **strip** using the configuration file `boot/i386/config.ini` producing output file `build/i386/final.prepre`. ELF sections `.note` and `.comment` are also removed here.
4. In this step `final.prepre` is compressed by the **gzip** by using the command
`gzip -f -9 build/i386/final.prepre` and producing file `final.prepre.gz`
5. The **cat** command concatenates the `stage1` file (the primary boot loader) and `final.pre.gz` to produce the output file `final.pre`.
`cat build/i386/boot/stage1 build/i386/final.prepre.gz >
build/i386/final.pre`
6. Then `bootblock.asm` is assembled producing the output file `bootblock.bin`. This is 512 bytes code that is placed in the boot sector of disk.
7. Then the final image of the SysPak OS is made by the program **makeflop**. This binary combines *the bootblock.bin* with *final.pre* producing the final image of the SysPak OS. The command is

```
boot/i386/makeflop -p 18432 boot/i386/bootblock.bin build/i386/final.pre  
build/i386/final
```

3.3 SysPak OS Source Code Directory Hierarchy

At the top most level, there are only directories:

- ❖ `/boot` - this is the directory that contains boot information and tools to make a bootable kernel
- ❖ `/core` - this is the crown jewels - the "main" code of the kernel
- ❖ `/drivers` - a home of their own
- ❖ `/global` - for any code that spans the above
- ❖ `/glue` - this is the code that must be linked with each executable
- ❖ `/ldscripts` - one wrong move and your kernel is retargetable.
- ❖ `/libc` - all those little functions that developers take for granted
- ❖ `/libm` - all those little functions that pencil pushing math majors understand
- ❖ `/prefs` - for the preferences apps for the kernel

Since everyone is so interested in core, let's talk for a brief moment about each file/directory therein...

- ❖ `/core/addons` - this is where the (surprise) addons go - our kernel is modular, too. Examples include file systems and bus managers
- ❖ `/core/arch` - home of the architecture specific stuff . This is the place for the portability police to start their investigation
- ❖ `/core/fs` - contains the code for the pseudo le systems - vfs, bootfs, etc
- ❖ `/core/vm` - the virtual memory module
- ❖ `cbuf.c` - this is a whole le dedicated to a buffering system. Used in the ports system.

- ❖ console.c - implements kernel versions of printf.
- ❖ cpu.c - contains the atomic* and test and set system call implementations
- ❖ debug.c - contains the kernel debugger
- ❖ elf.c - functions to load and interpret ELF files (executables)
- ❖ faults.c - handling for processor faults (like divide by 0, etc)
- ❖ gdb.c - code to allow the kernel to be remote debugged by gdb
- ❖ heap.c - malloc and free for the kernel
- ❖ int.c - interrupt handler code
- ❖ khash.C - yes, that's right, some almost C++ code. A generic hash table implementation
- ❖ lock.c - recursive locking and mutex implementation
- ❖ main.c - doesn't everyone need one of these? Initialized the system. The only allowable place for while(1).
- ❖ misc.c - some checksum functionality
- ❖ module.c - in order to have a modular kernel, someone has to load the modules...
- ❖ port.c - implements ports, so messaging can work
- ❖ queue.c - very simple queue
- ❖ sem.c - semaphore implementation
- ❖ smp.c - for those lucky dogs with more than one processor, we need Symmetrical MultiProcessing
- ❖ syscalls.c - because user land wouldn't be the same without it
- ❖ thread.c - processes and threads are implemented here. Also the scheduler.
- ❖ timer.c - All timer stuff , including task switching, goes through here

3.4 Booting Process

Booting process can be separated into the following six logical stages:

1. BIOS selects the boot device.
2. BIOS loads the bootsector from the boot device.
3. Bootsector loads setup, decompression routines and compressed kernel image.
4. The kernel is uncompressed in protected mode.
5. Low-level initialisation is performed.

3.4.1 BIOS POST

1. The power supply starts the clock generator and asserts #POWERGOOD signal on the bus.
2. CPU #RESET line is asserted (CPU now in real 8086 mode).
3. %ds=%es=%fs=%gs=%ss=0, %cs=0xFFFF0000,%eip = 0x0000FFF0 (ROM BIOS POST code).
4. All POST checks are performed with interrupts disabled.
5. IVT (Interrupt Vector Table) initialised at address 0.
6. The BIOS Bootstrap Loader function is invoked via **int 0x19**, with %dl containing the boot device 'drive number'. This loads track 0, sector 1 at physical address 0x7C00 (0x07C0:0000).

3.4.2 Processor Startup Sequence

The processors in Intel's x86 clan begin life by performing a hardware reset, initializing all the little bits of cache, registers, and buffers to a known value. It goes into [real mode](#) shortly. The EIP register is initialized to 0000FFF0H, and the CS register is a segment selector (value F000H) which points to a base address of FFFF0000H. Thus, execution begins at address FFFFFFF0H, sixteen bytes from the top of physical memory, in an EPROM. The EPROM is usually located at a much lower physical address, but is being remapped to a high address by the system chipset (e.g. Intel 430HX). Note that the selector/base correspondence here is not

the usual relationship when programming in real mode. Typically, in a PC this EPROM will set up a real-mode IDT and jump to the BIOS.

3.4.3 Bootstrap Sequence

1. The BIOS bootstrap routine generates an int 0x19 which usually loads the first [sector](#) of the floppy or hard disk (0:0:1 in CHS [disk addressing](#) format) in memory at [segment address](#) 0000:7C00H. The first sector (in this example) is the [primary bootstrap loader](#).
The BIOS checks that the last two bytes of that sector are AA55H. If they are not, you will probably get a BIOS-dependent message (maybe Non-System or Non-Bootable Disk) or a hang.
2. This primary or *first stage* bootstrap loader mainly has the job of loading the [secondary bootstrap loader](#). The code for the primary bootstrap is in `/boot/i386/stage2`. This is generated from code in a directory, in this case `/boot/i386/`.
3. Secondary bootstrap loader starts high level initialisation.

3.5 High level initialization

By "high-level initialization" we consider anything which is not directly related to bootstrap, even though parts of the code to perform this are written in asm, namely `arch/i386/kernel/stage2.c` which is the head of the uncompressed kernel. The following steps are performed:

1. Initialise segment values
2. Check CPU type using EFLAGS and, if possible, *cpuid*, able to detect 386 and higher.
3. Initialise page tables.
4. Enable paging by setting PG bit in `%cr0`.
5. Copy the *ka* parameters (kernel_arguments).
6. The first CPU (Boot Processor) calls `start()`, in `/kernel/main.c`.

The `/kernel/main.c: start()` is written in C and does the following:

1. Take a global kernel lock (it is needed so that only one CPU goes through initialization).
2. Perform arch-specific setup (memory layout analysis, copying kernel arguments etc.).
3. Initialise traps.
4. Initialise irqs.
5. Initialise data required for scheduler.
6. Initialise time keeping data.
7. Initialise softirq subsystem.
8. Parse boot commandline options.
9. Initialise console.
10. If module support was compiled into the kernel, initialise dynamical module loading facility.
11. Go into the idle loop, this is an idle thread with `pid=0`.

3.7 Bringing Up the Shell

After high level initialization, shell is brought up. The `init` process is created which creates the process, `shell`. Shell then take over and executes all the commands, given to it at command line.