System Boot and Startup

The first step in starting the operating system is that the kernel is loaded into memory by a **boot** loader.

When the FreeBSD kernel is loaded into memory, the kernel goes through several stages of hardware and software initialization.

- Initialize CPU state including run-time stack and virtual-memory mapping.
- Machine-dependent initializations.
  - → Setting up mutexes and page tables.
  - → Configuring I/O devices.
- Machine-independent initializations.
  - → Mounting root filesystem.
  - → Initializing system data structures.
- Create the first user-mode process.
- Start user-mode system processes

Bootstrapping

- When a PC computer is powered up control is transferred to a startup routine in the BIOS (Basic Input/Output System), stored in nonvolatile storage.
  - → The BIOS runs some hardware diagnostics.
  - → A short boot program is loaded from the boot sector (sector 0) of the boot disk.
  - → Which disk to use as boot disk is selectable in the BIOS setup menu.
- If the standard PC boot loader is used, it will load another boot loader from the boot block in the partition marked as boot partition in the MBR (Master Boot Record).
- This second level boot loader may be the FreeBSD boot loader or a general purpose boot loader such as GRUB.
- The FreeBSD boot loader loads the kernel into memory from a standard location in the root file system (or it can be given commands to load from a nonstandard location).

Boot Programs

- A boot program is a *standalone program* that operates without assistance of an operating system.
- A standalone program is usually linked against a standalone I/O library, that supports a variety of hardware devices.
- Some boot programs use BIOS I/O routines to read from the disk.
  - → Such boot programs often create problems if not booting from the first partition on the disk, because many BIOS:es have severe limitations on the number of disk sectors they can access.

Kernel Initialization

- Ultimately the FreeBSD kernel is loaded into memory by a boot loader.
- When the FreeBSD kernel is started by the boot program it will begin execution with several initialization stages:

1. Run assembly code to do basic hardware initializations.
2. Initialize the kernel modules that implement all the kernel's internal services.
3. Start up the user mode system processes and run the user level startup scripts.

Assembly Startup

When the boot program starts the FreeBSD kernel:

1. All interrupts are disabled.
2. The hardware address-translation is disabled so all memory references are to physical memory locations.

- The first initializations are highly machine dependent and are carried out by assembly language code:
  - ➝ Setting up the runtime stack.
  - ➝ Identifying the type of CPU.
  - ➝ Calculating the amount of physical memory.
  - ➝ Initializing and enabling virtual memory address translation.

Kernel Initialization

- The FreeBSD kernel is structured into a number of subsystems.
- Most of the subsystems depend on data structures and possibly hardware that must be initialized before the subsystem can be used.
- Because many of the subsystems are dependent on each other, the initialization of the data structures must be done in the correct order.
- Each subsystem uses the SYSINIT macro to register an initialization subroutine.

```
SYSINIT(name, subsystem, order, function, identifier
```

- The *function* argument is the name of the initialization routine that will be called at system startup.
- The *subsystem* argument is a global subsystem identifier defined in the file **kernel.h**.
- The *subsystem* identifier is defined as a numeric constant that determines the order in which the initialization subroutines will be called.

Kernel Initialization

- The FreeBSD kernel supports *kernel loadable modules*, that can be loaded and unloaded at run-time.
- The *kernel loadable modules* are typically used for device drivers, but may also be used for other parts of the kernel.
- The kernel subsystems are sometimes called *permanent kernel modules*.
- After the assembly level code has completed, the C language *mi_startup()* routine is called.
- The *mi_startup()* routine sorts the list of routines that need to be started, and calls the *function* routine for each.
- The list of subsystem identifiers is defined so that some basic system services are initialized first (Table 14.2).

Kernel subsystems

Examples of initialization routines for basic system services:

vm_mem_init() Activates the memory management unit. After this all memory allocations in the kernel are for virtual addresses.

eventhandler_init() Initializes an event-handler service that allows kernel routines to register functions to be called when an event occurs.

module_init() Creates the basic data structures for module loading and unloading.

apic_init() Initializes the APIC (Advanced Programmable Interrupt Controller)

mp_start() Activates the extra processors on an SMP. The extra processors are now active, but will not be scheduled until the entire startup sequence is complete.

Kernel Process Initialization

- The kernel has several processes that are set up at boot time and needed whenever the kernel is running (Table 14.3).
- The *swapper, init*, and *idle* processes are all created early in the startup process, but are not scheduled to run until activated by the scheduler as the last step in the kernel initialization.
- The *proc0_init()* routine initializes the process control blocks for process 0, the *swapper*.
  - ➝ It also creates a prototype virtual-memory map that will be the prototype for the other processes that will be created.
- After the swapper process is initialized, *create_init()* is called to create the **init** process.
  - ➝ Creating the init process immediately after process 0, will give it PID 1.
  - ➝ The init process is created by first calling *fork1()* to create a copy of process 0.
  - ➝ The start address of the new init process is modified to the *start_init()* kernel routine.
  - ➝ When **init** finally is started by the scheduler it will start running *start_init()* that will do execve("/sbin/init").
- An *idle* process is created for each CPU in the system.

Device Initialization

- When the kernel's basic services and basic processes have been created, the rest of the devices can be initialized (Table 14.4).
- First some more helper systems are initialized:

  mbuf_init() Initialize the memory allocation routines for the network mbufs.

  start_softintr() Initialize the softclock *callout* subsystem.
- The devices are initialized by the *autoconfiguration* system.
- The *initclocks()* routine is called to start the hardware clocks.

Start Kernel Threads

The last steps in the kernel initialization are:

- Setup of kernel threads like *pagedaemon* and *bufdaemon* (table 14.6).
- Start the extra processors on an SMP (The Application Processors in Intel speak).
- Finally the *scheduler()* routine is called and never returns.
  - → It starts scheduling the kernel threads and then the user level processes (init).

User-Level Initialization

- When the **init** process is started by the scheduler it execs **/sbin/init**
- When booted into multiuser mode, init first spawns a shell that executes the commands in the **/etc/rc** script file.
- All user level initializations are started from **/etc/rc.**
- If the **/etc/rc** script completes successfully, init forks a copy of itself for each terminal marked active in the **/etc/ttys** file.
- These copies of init execs **getty** to set up the terminal line.
- Finally getty execs **login**.
- The login program waits for a user to log in, and if successful execs a user shell.