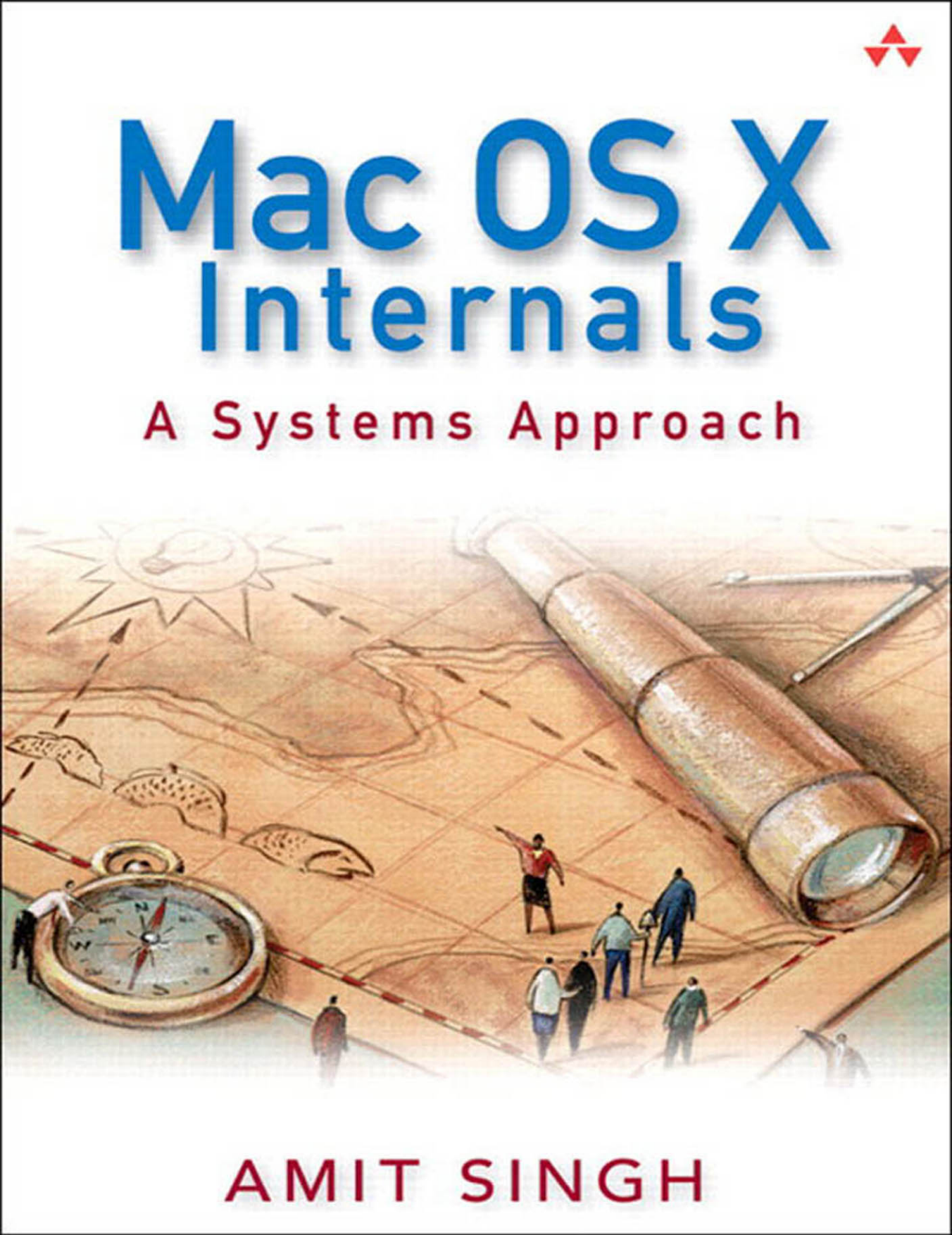
Mac OS X
Internals
A Systems Approach
AMIT SINGH

Mac OS X Internals

5.8 Launching the First User-Space Program

As BSD initialization concludes, `load_init_program()` [`bsd/kern/kern_exec.c`] is called to launch the first user program, which is traditionally `/sbin/init` on Unix systems but is another init program on Mac OS X.[38] The function first attempts to execute `/sbin/launchd`. If that fails, it attempts `/sbin/mach_init`. If that too fails, it prompts the user for a pathname to the program to run. The kernel uses `getchar()` [`bsd/dev/ppc/machdep.c`] to read the name character by character, echoing each character read. `getchar()` uses `cngetc()` and `cnputc()` [`osfmk/console/ppc/serial_console.c`], which are wrappers around the now established console I/O operations.

`load_init_program()` allocates a page of memory in the current task's map. It populates the page with a null-terminated list of arguments that have been collected so far in a string variable. `argv[0]` contains the init program's null-terminated name (e.g., `/sbin/launchd`), `argv[1]` contains an argument string that has a maximum size of 128 bytes (including the terminating `NUL` character), and `argv[2]` is `NULL`. Examples of arguments passed to the init program include those indicating safe (`-x`), single-user (`-s`), and verbose (`-v`) booting modes. An `execve_args` structure [`bsd/sys/exec.h`] is populated so that `execve()` can be called from within the kernel, while pretending as if it were called from user space. Consequently, these arguments are first copied out to user space, since the `execve()` system call expects its arguments to be there.

```
// bsd/kern/kern_exec.c

static char *init_program_name[128] = "/sbin/launchd";
static const char *other_init = "/sbin/mach_init";

char init_args[128] = "";

struct execve_args init_exec_args;
int init_attempts = 0;

void
load_init_program(struct proc *p)
{
    vm_offset_t  init_addr;
    char        *argv[3];
    int          error;
    register_t   retval[2];
    error = 0;
```

38. `/sbin/launchd` is the default init program beginning with Mac OS X 10.4.

```
    do {
        ...

        // struct execve_args {
        //     char *fname;
        //     char **argp;
        //     char **envp;
        // };
        init_exec_args.fname = /* user space init_program_name */
        init_exec_args.argp  = /* user space init arguments */
        init_exec.args.envp  = /* user space NULL */

        // need init to run with uid and gid 0
        set_security_token(p);

        error = execve(p, &init_exec_args, retval);
    } while (error);
}
```

Finally, the first user-space program begins to execute.

5.9 Slave Processors

Before we discuss user-level startup, let us look at the `ppc_init_cpu()` [`osfmk/ppc/ppc_init.c`] function. Recall from Figure 5–2 that at boot time, a slave processor calls `ppc_init_cpu()` instead of `ppc_init()`. The execution journey of a slave processor is much shorter than that of the master processor. Figure 5–22 shows the execution path of `ppc_init_cpu()`.

`ppc_init_cpu()` clears the `SleepState` bit in the `cpu_flags` field of the processor's `per_proc_info` structure. On 64-bit hardware, `ppc_init_cpu()` checks whether the `wcte` global variable is set to `0`; if so, it disables the noncacheable unit's store gather timer through an SCOM command. The value of the `wcte` variable can be set through the `wcte` boot argument (see Table 4–12).

Next, `ppc_init_cpu()` calls `cpu_init()` [`osfmk/ppc/cpu.c`], which we came across earlier in this chapter. `cpu_init()` restores the Timebase Register from values saved in the CPU's `per_proc_info` structure. It also sets the values of some fields in the `per_proc_info` structure. Finally, `ppc_init_cpu()` calls `slave_main()` [`osfmk/kern/startup.c`], which never returns.

Recall how the `kernel_bootstrap()` function, while running on the master processor, arranged—through `load_context()`—for `kernel_bootstrap_thread()` to start executing. Similarly, `slave_main()` arranges for `processor_`

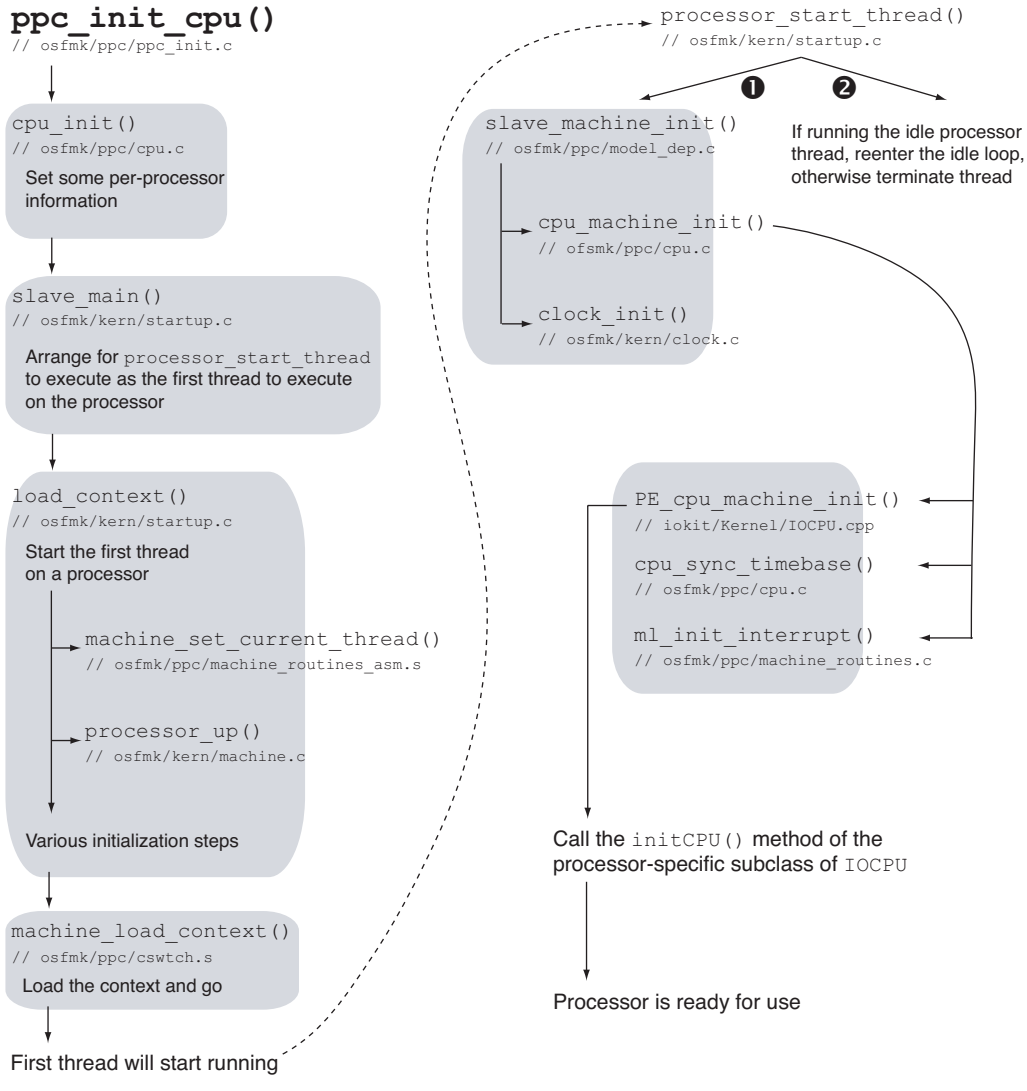

FIGURE 5–22 Slave processor initialization

start_thread() [osfmk/kern/startup.c] to start executing. processor_start_thread() calls slave_machine_init() [osfmk/ppc/model_dep.c].

slave_machine_init() initializes the processor by calling cpu_machine_init() [osfmk/ppc/cpu.c] and the clock by calling clock_

`init()` [`osfmk/kern/clock.c`]. We earlier noted the operation of `clock_init()`, which calls the initialization functions of all available clock devices. `cpu_machine_init()` calls `PE_cpu_machine_init()` [`iokit/Kernel/IOCPU.cpp`], synchronizes the Timebase Register with the master processor, and enables interrupts.

```
// iokit/Kernel/IOCPU.cpp

void
PE_cpu_machine_init(cpu_id_t target, boolean_t boot)
{
    IOCPU *targetCPU = OSDynamicCast(IOCPU, (OSObject *)target);

    if (targetCPU)
        targetCPU->initCPU(boot);
}
```

5.10 User-Level Startup

As Section 5.8 described, user-level startup is initiated when the kernel executes `/sbin/launchd` as the first user process. We will now look at the implementation and operation of `launchd`.

5.10.1 `launchd`

`launchd` is the master bootstrap daemon beginning with Mac OS X 10.4. It subsumes the functionality of the traditional init program and the erstwhile Mac OS X `mach_init` program. The following are notable features of `launchd`.

- It manages both system-wide *daemons* and per-user *agents*. An agent is a type of daemon that runs while a user is logged in. Unless the distinction is necessary, we will use the term *daemons* in this discussion to refer to both daemons and agents.
- As the first user process, it performs user-level system bootstrap.
- It handles both single-user and multiuser booting modes. In a multiuser boot, it runs the traditional BSD-style command script (`/etc/rc`) and sets up daemons whose configuration files are located in designated directories such as `/System/Library/LaunchDaemons/`, `/Library/LaunchDaemons/`,

`/System/Library/LaunchAgents/`, `/Library/LaunchAgents/`, and `~/Library/LaunchAgents/`.

- It supports daemons that are designed to run under the `inetd` superserver on Unix systems.
- It can run jobs periodically. A `launchd` job is an abstraction that consists of a runnable entity (a program) along with the entity's configuration.
- It allows several aspects of a daemon to be configured through a property list file, rather than the daemon having to programmatically configure itself.
- It can start daemons on demand, based on a variety of conditions.

`launchd` simplifies the configuration, management, and, in many cases, even creation of daemons.

5.10.1.1 Daemon Configuration and Management

`launchd` provides a set of predefined keys that can be used in a daemon's property list file to specify various runtime aspects of the daemon. The following are examples of such aspects:

- User and group names (or identifiers)
- Root and working directories
- Umask value
- Environment variables
- Standard error and standard output redirections
- Soft and hard resource limits
- Scheduling priority alterations
- I/O priority alterations

An important ability of `launchd` is that it can launch daemons when they are needed, rather than having "always on" processes. Such on-demand launching can be based on criteria such as the following:

- A given periodic interval
- An incoming connection request on a given TCP port number
- An incoming connection request on a given `AF_UNIX` path
- Modification of a given file system path
- Appearance or modification of file system entities in a given queue directory

The `launchd` configuration file for a daemon is an XML property list file. Let us look at some examples. Figure 5–23 shows the configuration file for the SSH daemon.

FIGURE 5–23 A `launchd` configuration file

```
$ ls -1 /System/Library/LaunchDaemons
bootps.plist
com.apple.KernelEventAgent.plist
com.apple.atrun.plist
com.apple.mDNSResponder.plist
...
ssh.plist
swat.plist
telnet.plist
tftp.plist
$ cat /System/Library/LaunchDaemons/ssh.plist
...
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.openssh.sshd</string>
    <key>Program</key>
    <string>/usr/libexec/sshd-keygen-wrapper</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/sbin/sshd</string>
        <string>-i</string>
    </array>
    <key>Sockets</key>
    <dict>
        <key>Listeners</key>
        <dict>
            <key>SockServiceName</key>
            <string>ssh</string>
            <key>Bonjour</key>
            <array>
                <string>ssh</string>
                <string>sftp-ssh</string>
            </array>
        </dict>
    </dict>
    <key>inetdCompatibility</key>
    <dict>
        <key>Wait</key>
        <false/>
    </dict>
    <key>SessionCreate</key>
```

(continues)