D Systemstart

Wie jedes Programm durchläuft der Kern eine Lade- und Initialisierungsphase, bevor er seine eigentlichen Aufgaben übernehmen kann. Während diese erste Phase bei normalen Anwendungen üblicherweise nicht sonderlich spannend ist, wird der Kern als zentrale Schicht des Systems mit einigen ungewöhnliche Problemen konfrontiert, die gelöst werden müssen. Die Bootphase selbst lässt sich in drei Abschnitte untergliedern:

- Laden des Kerns in den RAM-Speicher und Aufsetzen einer minimalen Ablaufumgebung
- Sprung in den (plattform*ab*hängigen) Maschinencode des Kerns und System-spezifische Initialisierung der elementaren Systemfunktionen in Assembler.
- Sprung in den (plattform*un*abhängig) in C geschriebenen Teil des Initialisierungscodes, vollständige Initialisierung aller Subsysteme und anschließender Wechsel in den normalen Betrieb.

Wie dem Leser bekannt ist, wird für die erste Phase ein Bootloader verwendet, dessen Aufgaben sehr stark von den Aufgaben der jeweiligen Architektur abhängig sind. Da sehr viel detailliertes Wissen über Eigenheiten und vor allem Probleme der einzelnen Prozessoren notwendig ist, um die erste Phase in allen Einzelheiten zu verstehen, wollen wir nicht nicht näher darauf eingehen – die Architektur-Referenzhandbücher sind in diesem Fall eine wesentlich bessere Quelle für Informationen. Auch die zweite Phase ist weiterhin stark von der jeweiligen Hardware abhängig, weshalb wir sie nicht allzu genau betrachten wollen; wir werden aber anhand der IA-32-Architektur auf einige Eckpunkte eingehen, die dabei erledigt werden müssen.

In der dritten, systemunabhängigen Phase befindet sich der Kern bereits im Speicher; der Prozessor wurde (auf manchen Architekturen) vom Boot- in den Ausführungsmodus geschaltet, in dem der Kern später laufen wird. Auf IA-32-Maschinen muss der Prozessor bekanntlich von der 8086-„Emulation", die direkt nach dem Einschalten aktiv ist, in den *Protected Mode* umgeschaltet werden, um die Verwandlung in ein 32-Bit-System zu vollziehen. Aber auch auf anderen Architekturen fallen verschiedene Setuparbeiten an, beispielsweise muss häufig das Paging erst explizit aktiviert werden; außerdem müssen die zentralen Komponenten des Systems in einen definierten Ausgangszustand gebracht werden, der die weitere Arbeit damit ermöglicht. Alle diese Arbeiten müssen in Assembler durchgeführt werden und gehören damit nicht zu den angenehmsten Teilen des Kerns.

Die Konzentration auf die dritte Phase erspart uns die Behandlung vieler architekturspezifischer Kleinigkeiten und hat den zusätzlichen Vorteil, dass die restliche Vorgehensweise im Großen und Ganzen von der konkreten Plattform unabhängig ist, auf der der Kern ausgeführt wird.

D.1 Architekturspezifisches Setup auf IA-32-Rechnern

Nachdem der Kernel mit Hilfe des Bootloaders (LILO, Grub etc.) in den physikalischen Speicher geladen wurde, wird die Assembler-„Funktion" `setup` aus `arch/i386/boot/setup.S` ausgeführt, indem der Kontrollfluss durch eine Jump-Anweisung an die passende Stelle im Speicher

gelenkt wird. Dies ist möglich, da sich die `setup`-Funktion immer an der gleichen Stelle der Objektdatei befindet.

Der Code muss folgende Aufgaben ausführen, wofür sehr viel Assemblercode notwendig ist:

- Überprüfen, ob der Kern an die korrekte Stelle im Speicher geladen wurde, wozu eine Vier-Byte-Signatur verwendet wird, die in das Kernelimage integriert ist und sich unverändert und an der korrekten Position im RAM befinden muss.
- Feststellen, wie viel Speicher im System vorhanden ist.
- Initialisierung der Grafikkarte.
- Überprüfen, ob einige Systemkomponenten vorhanden sind oder nicht (MicroChannel-Bus, PS/2-Maus, APM-Bios).
- Verschieben des Kernelimages an eine Position im Speicher, wo er sich bei der später folgenden Dekompression nicht selbst im Wege steht.
- Umschalten der CPU in den Protected Mode.

Nach Abschluss dieser Arbeiten springt der Code zur Funktion `startup_32` aus `arch/i386/boot/compressed/head.S`, die folgende Aufgaben erledigt:

- Anlegen eines provisorischen Kernelstacks.
- Ausfüllen der uninitialisierten Kerneldaten mit Nullbytes. Dabei handelt es sich um den Bereich zwischen den Konstanten `_edata` und `_end`, die beim Linken des Kerns automatisch mit den korrekten Werten belegt werden, die sich für das Kernelbinary ergeben.
- Dekomprimieren des Kerns und Schreiben des dekomprimierten Maschinencodes an Position 0x100000, also unmittelbar hinter das erste MiB des Speichers. Das Entpacken ist der erste Schritt des Kerns, der durch eine Meldung auf den Bildschirm sichtbar gemacht wird, indem die Zeichenketten `Uncompressing Linux...` und `Ok, booting the kernel` ausgegeben werden.

Der letzte Teil der prozessorspezifischen Initialisierung wird eingeleitet, indem der Kontrollfluss zu einer anderen Stelle umgeleitet wird, die in `arch/i386/kernel/head.S` unter der Bezeichnung `startup_32` zu finden ist. Achtung: Es handelt sich dabei um eine *andere* Routine als die eben angesprochene `startup_32`, da sie in einer anderen Datei definiert wird. Der Kern muss sich nicht darum kümmern, dass beide „Funktionen" mit dem gleichen Label bezeichnet werden, da er direkt an die passende Adresse springt, die vom Assembler eingepatcht wird, und sich nicht auf die symbolischen Labels bezieht, die im Quellcode verwendet werden.

Dieser Bootabschnitt führt folgende Schritte aus:

- Der Paging-Modus wird aktiviert und ein endgültiger Kernelstack eingerichtet.
- Das `.bss`-Segment, das sich zwischen `__bss_start` und `__bss_stop` befindet, wird mit Nullbytes aufgefüllt.
- Die Interrupt-Deskriptor-Tabelle wird initialisiert. Als Handler wird für alle Interrupts aber die Dummy-Routine `ignore_int` eingetragen; die eigentlichen Handler werden später installiert.

- Der verwendete Prozessortyp wird erkannt, wozu bei halbwegs aktuellen Modellen die `cpuid`-Anweisung verwendet werden kann, die verschiedene Informationen über Typ und Fähigkeiten des Prozessors liefert (die Unterscheidung zwischen 80386 und 80486, die die Anweisung noch nicht unterstützen, erfolgt mit Hilfe verschiedener Assembler-Tricks, die aber weder besonders wichtig noch besonders interessant sind).

Damit ist das Ende der plattformspezifischen Initialisierung erreicht: Der Code springt nun zur Funktion `start_kernel`, die – im Gegensatz zum bisher beschriebenen Code – als ganz normale C-Funktion implementiert und daher wesentlich angenehmer zu handhaben ist.

D.2 High-Level Initialisierung

`start_kernel` dient als Dispatcher-Funktion, die sowohl plattformunabhängige wie auch plattformabhängige Initialisierungsaufgaben erledigt, die aber allesamt in C implementiert sind. Die Funktion ist dafür verantwortlich, die High-Level-Initialisierungsroutinen beinahe aller Subsysteme des Kerns aufzurufen. Da die Funktion als eine der ersten Aktionen den Linux-Banner auf den Bildschirm ausgibt, kann der Benutzer erkennen, wann der Kern in diese Phase der Initialisierung eingetreten ist. Auf einem der Systeme des Autors liefert dies beispielsweise folgende Meldung:[1]

```
Linux version 2.6.0-test4 (root@jupiter) (gcc version 3.2.1) #2 Thu Sep 25 23:27:54
```

Die Anzahl der Bildschirmausgaben nimmt in den folgenden Schritten drastisch zu, da die initialisierten Subsysteme verschiedene Statusinformationen auf die Konsole ausgeben, die vor allem beim Beheben von Problemen gute Dienste leisten.

Die folgenden Abschnitte werden sich detaillierter mit `start_kernel` auseinandersetzen und daher den Startprozess des Kerns unmittelbar nach dem Ende der vollständig architekturabhängigen Phase beleuchten.

D.2.1 Initialisierung der Subsysteme

Abbildung D.1 auf der nächsten Seite zeigt ein Codeflussdiagramm, das einen ersten Überblick zu den Aufgaben und Zielen der Funktion geben soll.

Als erster Schritt wird die bereits erwähnte Versionsmeldung ausgegeben, deren Text durch die globale Variable `linux_banner` festgelegt ist, die in `init/version.c` definiert wird. Anschließend folgt ein weiterer architekturespezifischer Initialisierungsschritt, der sich aber nicht mehr mit den tieferen Prozessordetails des Systems beschäftigt, sondern in C geschrieben ist und auf den meisten Systemen vor allem das Rahmenwerk für die Initialisierung der High-Level-Speicherverwaltung legt. Nachdem die Kommandozeilenargumente ausgewertet wurden, die dem Kern beim Starten übergeben werden können – beispielsweise, um die Root-Partition festzulegen –, folgt der Hauptteil der Initialisierungsarbeit in `start_kernel`: Das Aufsetzen zentraler Datenstrukturen der verschiedenen Teilsysteme des Kerns. Da praktisch jedes Subsystem davon betroffen ist, handelt es sich um eine sehr umfangreiche Aufgabe, die auf viele kleine weitere Prozeduren verteilt ist, auf die wir gleich eingehen werden. Als letzter Schritt wird der Idle-Prozess generiert, den der Kern aufruft, wenn absolut nichts besseres zu tun ist, und außerdem der `init`-Prozess mit PID 1 gestartet, der zunächst die Initialisierungsroutinen verschiedener Subsysteme

1 Achtung: Die Meldung wird zwar früh im Bootvorgang erzeugt, aber erst dann auf den Bildschirm ausgegeben, wenn das Konsolen-Subsystem initialisiert ist. Vorher wird sie in einem Puffer zwischengespeichert.

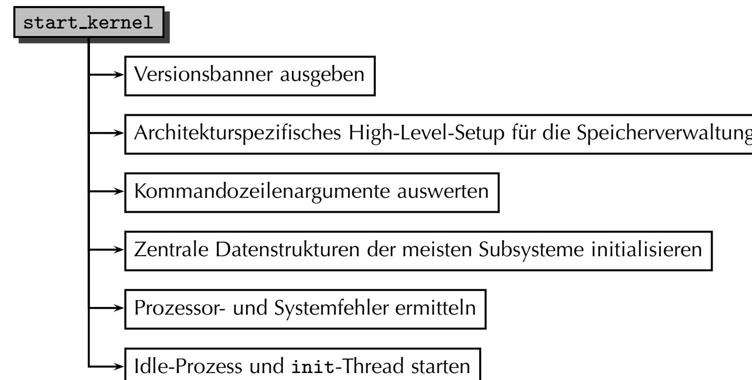

Abbildung D.1: Codeflussdiagramm zu `start_kernel`

ausführt und anschließend `/sbin/init` als ersten Userspace-Prozess des Systems startet, womit die kernelseitige Initialisierung abgeschlossen ist.

Architekturspezifisches Setup

`setup_arch` ist – wie ihr Name unmissverständlich klarmacht – eine architekturspezifische Funktion. Sie führt in C geschriebene Setuparbeiten aus, die sich größtenteils um die Initialisierung verschiedener Aspekte der Speicherverwaltung kümmern. Auf den meisten Systemen wird an dieser Stelle beispielsweise das Paging endgültig aktiviert und passende Datenstrukturen für den Kernmodus eingerichtet. Manche Architekturen wie IA-64 oder Alpha, die in unterschiedlichen Variationen existieren, führen an dieser Stelle ein variantenspezifisches Setup durch.

Der Einfachheit halber wollen wir nur die Implementierung von `setup_arch` für IA-32-Rechner betrachten, der wir in Kapitel 3 („Speicherverwaltung") bereits kurz begegnet sind. Abbildung D.2 zeigt das zugehörige Codeflussdiagramm.

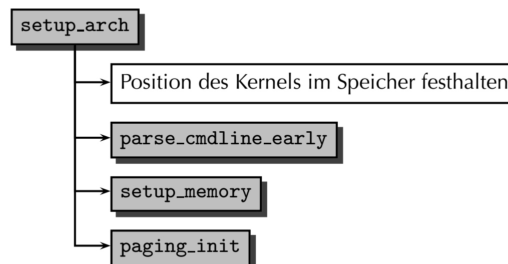

Abbildung D.2: Codeflussdiagramm für `setup_arch` *auf IA-32*

Zunächst wird die Position des Kerns im physikalischen und virtuellen Speicher festgehalten, wozu beim Übersetzen des Kerns durch den Linker eingefügte Konstanten verwendet werden, die Anfangs- und Endadresse der verschiedenen Segmente angeben (siehe dazu auch Kapitel E („Das ELF-Binärformat")):

arch/i386/kernel/setup.c

```
init_mm.start_code = (unsigned long) _text;
init_mm.end_code = (unsigned long) _etext;
```

```
init_mm.end_data = (unsigned long) _edata;
init_mm.brk = (unsigned long) _end;

code_resource.start = virt_to_phys(_text);
code_resource.end = virt_to_phys(_etext)-1;
data_resource.start = virt_to_phys(_etext);
data_resource.end = virt_to_phys(_edata)-1;
```

`parse_cmdline_early` wird verwendet, um eine teilweise Auswertung der Kommandozeilenparameter durchzuführen. Es werden nur Argumente berücksichtigt, die für das Setup der Speicherverwaltung relevant sind, beispielsweise die Gesamtgröße des vorhandenen physikalischen Speichers oder die Position spezieller ACPI- und Bios-Speicherbereiche. Dadurch kann der Benutzer Werte manuell überschreiben, die der Kern falsch erkennt. Mit diesem Wissen gerüstet macht sich `setup_memory` daran, die Anzahl physikalischer Speicherseiten im Low- und Highmem-Bereich zu erkennen; außerdem wird der Bootmem-Allokator initialisiert.

In `paging_init` wird anschließend zunächst die Referenz-Seitentabelle des Kerns aufgesetzt, mit deren Hilfe nicht nur der physikalische Speicher eingeblendet wird, sondern auch die `vmalloc`-Bereiche verwaltet werden, wie in Kapitel 3 besprochen wurde. Durch Einfügen der Adresse von `pg_swapper_dir` – der Variablen, in der die Seitentabellendatenstrukturen gespeichert werden – in das CR3-Register des Prozessors wird die neue Seitentabelle aktiviert.

Um die Initialisierung der Speicherverwaltung vorerst abzuschließen und den Bootmem-Allokator zurückzulassen, der für den weiteren Bootvorgang verwendet wird, wird von `start_kernel` aus noch die Funktion `build_all_zonelists` aufgerufen, die bereits aus Kapitel 3 („Speicherverwaltung“) bekannt ist und die Zonenlisten der Speicherverwaltung aufsetzt.

Auswerten der Kommandozeilenargumente

`parse_args`, das von `start_kernel` aus aufgerufen wird, übernimmt die Auswertung der Kommandozeilenparameter, die dem Kern beim Booten übergeben wurden. Das zugrunde liegende Problem ist zur Genüge aus dem Userspace bekannt: Eine Zeichenkette, die Schlüssel/Wert-Paare der Form `key1=val1 key2=val2` enthält, muss in ihre Bestandteile zerlegt werden. Die gesetzten Optionen müssen im Kern gespeichert werden bzw. spezifische Reaktionen hervorrufen.

Das Parameterproblem tritt im Kern nicht nur beim Booten, sondern auch beim Einfügen von Modulen auf, weshalb es naheliegt, in beiden Fällen die gleichen Mechanismen zu verwenden, um es zu lösen: Dies vermeidet unnötige Codeduplikationen.

Für jeden Kernelparameter – sowohl in dynamisch zuladbaren Modulen wie auch im statischen Kernelbinary – existiert eine Instanz der Datenstruktur `kernel_param` in der Binärdatei, die folgendermaßen aufgebaut ist:

<moduleparam.h>

```
/* Returns 0, or -errno. arg is in kp->arg. */
typedef int (*param_set_fn)(const char *val, struct kernel_param *kp);
/* Returns length written or -errno. Buffer is 4k (ie. be short!) */
typedef int (*param_get_fn)(char *buffer, struct kernel_param *kp);

struct kernel_param {
        const char *name;
        param_set_fn set;
        param_get_fn get;
        void *arg;
};
```

Während `name` die Bezeichnung des Parameters angibt, sind `set` und `get` Funktionen, die den Parameterwert setzen bzw. auslesen. `arg` ist ein (optionales) Zusatzargument, das den beiden

genannten Funktionen mit übergeben wird. Es dient wie üblich dazu, die gleiche Funktion für verschiedene Parameter verwenden zu können.

Um einen Parameter beim Kern zu registrieren, werden die Makros `module_param`, `module_param_named` etc. verwendet, die eine Instanz von `kernel_param` mit den passenden Werten ausfüllen und diese in die `__param`-Sektion der Binärdatei schreiben.

Dies erleichtert die Parameterauswertung beim Booten sehr; es wird nur eine Schleife benötigt, die folgende Aktionen ausführt, bis alle Parameter abgearbeitet wurden:

- `next_arg` extrahiert das nächste Name/Wert-Paar aus der Kommandozeile, die vom Kern in Form eines Textstrings bereitgehalten wird.
- `parse_one` durchläuft die Liste aller registrierten Parameter, vergleicht den übergebenen Wert mit dem `name`-Element der `kernel_param`-Instanzen und ruft die `set`-Funktion auf, wenn eine Übereinstimmung gefunden wurde.

Initialisierung zentraler Datenstrukturen und Caches

Die umfangreichste Aufgabe von `setup_arch` ist es, zahlreiche Subroutinen zur Initialisierung beinahe aller wichtigen Teilsysteme des Kerns aufzurufen, wie ein Blick in die Kernelquellen zeigt:

init/main.c

```
        trap_init();
        init_IRQ();
        pidhash_init();
        sched_init();
        softirq_init();
        time_init();

        /*
         * HACK ALERT! This is early. We're enabling the console before
         * we've done PCI setups etc, and console_init() must be aware of
         * this. But we do want output early, in case something goes wrong.
         */
        console_init();
        profile_init();
        local_irq_enable();

        page_address_init();
        mem_init();
        kmem_cache_init();

        calibrate_delay();
        pidmap_init();
        pgtable_cache_init();
        pte_chain_init();
        fork_init(num_physpages);
        proc_caches_init();
        buffer_init();
        vfs_caches_init(num_physpages);
        radix_tree_init();
        signals_init();
        /* rootfs populating might need page-writeback */
        page_writeback_init();
#ifdef CONFIG_PROC_FS
        proc_root_init();
#endif
```

Die meisten Funktionen sind allerdings nicht besonders spannend, da sie nur mit Hilfe des Bootmem-Allokators Speicher reservieren, der zur Instantiierung von Datenstrukturen verwen-

det wird. Da die wichtigsten beteiligten Funktionen außerdem bereits in den Subsystem-spezifischen Kapiteln angesprochen wurden, wollen wir hier nur überblicksweise zusammenfassen, welche Bedeutung die einzelnen Aktionen haben:

- `trap_init` und `init_IRQ` setzen die Handler für Traps und IRQs, was eine architekturspezifische Aufgabe ist. Auf IA-32 wird beispielsweise folgender Code verwendet, um Trap-Handler für Fehlermeldungen zu registrieren, die vom Prozessor geliefert werden können:

  arch/i386/kernel/traps.c

```
void __init trap_init(void)
{
        set_trap_gate(0,&divide_error);
        set_intr_gate(1,&debug);
        set_intr_gate(2,&nmi);
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_intr_gate(14,&page_fault);
        set_trap_gate(15,&spurious_interrupt_bug);
        set_trap_gate(16,&coprocessor_error);
        set_trap_gate(17,&alignment_check);
        set_trap_gate(19,&simd_coprocessor_error);

        set_system_gate(SYSCALL_VECTOR,&system_call);

        /* default LDT is a single-entry callgate to lcall7 for iBCS
         * and a callgate to lcall27 for Solaris/x86 binaries
         */
       set_call_gate(&default_ldt[0],lcall7);
       set_call_gate(&default_ldt[4],lcall27);
}
```

  Wie der Code zeigt, wird an dieser Stelle auch der für Systemaufrufe verwendete Interrupt als System-Gate definiert (`SYSCALL_VECTOR` ist entsprechend auf 0x80 gesetzt) und die Call-Gates für die Binäremulation mit den verschiedenen BSD-Varianten aufgesetzt.

  Die Initialisierung der IRQ-Handler läuft nach einem ähnlichen Schema ab, weshalb wir hier nicht weiter darauf eingehen.

- `sched_init` initialisiert die Datenstrukturen des Schedulers (hier für den Hauptprozessor); insbesondere werden die Runqueues angelegt.

- `pidhash_init` alloziert die Hashtabellen, die vom PID-Allokator zur Verwaltung freier und belegter PIDs verwendet werden.

- `softirq_init` registriert die SoftIRQ-Queues für Tasklets normaler und hoher Priorität, `TASKLET_SOFTIRQ` und `HI_SOFTIRQ`

- `time_init` liest die Systemzeit aus, die in der Hardwareuhr gespeichert ist. Da unterschiedliche Architekturen hierfür unterschiedliche Mechanismen verwenden, ist dies eine prozessorabhängige Funktion.

- `init_console` initialisiert die Konsolen des Systems. Auf Systemen, die mit einem „early printk"-Mechanismus ausgestattet sind, die Meldungen auf die Konsole ausgeben können, bevor diese vollständig initialisiert wurde (auf anderen Systemen werden die ausgegebenen Meldungen in einem Puffer gespeichert, bis die Konsole aktiviert ist), wird dieser zusätzlich deaktiviert.
- `page_address_init` setzt die Hashtabelle auf, mit deren Hilfe der PKMAP-Mechanismus die physikalische Seitenadresse eines permanenten Kernelmappings anhand einer gegebenen virtuellen Adresse ermittelt.
- `mem_init` setzt den Bootmem-Allokator außer Kraft (und führt noch einige architekturspezifischen Kleinigkeiten durch, die uns aber nicht weiter interessieren), während `kmem_cache_init` den Slab-Allokator in einem mehrstufigen Prozess initialisiert, der in Kapitel 3 im Detail beschrieben ist.
- `calibrate_delay` berechnet den berühmten BogoMIPS-Wert, der angibt, wie viele leere Schleifen in einem Jiffies-Tick vom Prozessor durchlaufen werden können. Der Kern benötigt diesen Wert, um das Zeitmaß für manche Aufgaben abschätzen zu können, die im Polling- oder Busy-Waiting-Verfahren erledigt werden. Zur Berechnung des Wertes wird folgender Code verwendet, der eine gute Näherung für die exakte Anzahl von Schleifendurchläufen pro Jiffy berechnet und das Ergebnis in `loops_per_jiffy` speichert:

init/main.c

```
void __init calibrate_delay(void)
{
        unsigned long ticks, loopbit;
        int lps_precision = LPS_PREC;

        loops_per_jiffy = (1<<12);

        printk("Calibrating delay loop... ");
        while (loops_per_jiffy <<= 1) {
                /* wait for "start of" clock tick */
                ticks = jiffies;
                while (ticks == jiffies)
                        /* nothing */;
                /* Go .. */
                ticks = jiffies;
                __delay(loops_per_jiffy);
                ticks = jiffies - ticks;
                if (ticks)
                        break;
        }

/* Do a binary approximation to get loops_per_jiffy set to equal one clock
    &process`comment((up to lps`precision bits) )*/
        loops_per_jiffy >>= 1;
        loopbit = loops_per_jiffy;
        while ( lps_precision-- && (loopbit >>= 1) ) {
                loops_per_jiffy |= loopbit;
                ticks = jiffies;
                while (ticks == jiffies);
                ticks = jiffies;
                __delay(loops_per_jiffy);
                if (jiffies != ticks)   /* longer than 1 tick */
                        loops_per_jiffy &= ~loopbit;
        }

/* Round the value and print it */
        printk("%lu.%02lu BogoMIPS\n",
```

```
                loops_per_jiffy/(500000/HZ),
                (loops_per_jiffy/(5000/HZ)) % 100);
}
```

Besonders interessant ist dabei folgende Konstruktion, die in C normalerweise keinerlei Sinn ergeben bzw. in eine Endlosschleife münden würde:

init/main.c

```
ticks = jiffies;
while (ticks == jiffies)
       /* nothing */;
```

Die Schleife bricht aber irgendwann ab, da der Wert von `jiffies` bei jedem Tick der Systemuhr (die bekanntlich mit Frequenz `HZ` schläft) in der Interrupt-Handlerroutine um 1 inkrementiert wird, weshalb die Bedingung in der `while`-Schleife nach einiger Zeit einen falschen Wert ergibt und damit zum Schleifenabbruch führt.

- `pidmap_init` alloziert das Array, in dem die freien Positionen des PID-Allokators gespeichert werden, und reserviert außerdem die (nicht verwendete) PID 0 für alle PID-Typen.
- `pgtable_init` setzt die Slab-Caches für mittlere Seitenverzeichnisse und die Seitentabellen selbst auf.
- `pte_chain_init` erzeugt den Slab-Cache für `pte_chain`-Instanzen, die vom Reverse Mapping-Subsystem benötigt werden.
- `fork_init` alloziert den `task_struct`-Slab-Cache (sofern kein architekturspezifischer Mechanismus zum Erzeugen und Cachen von `task_struct`-Instanzen zur Verfügung steht) und berechnet außerdem die maximale Threadanzahl, die erzeugt werden kann.
- `proc_caches_init` erstellt Slab-Caches für die restlichen Datenstrukturen, die die Beschreibung eines Prozesses ausmachen. Folgende Strukturen werden dabei berücksichtigt: `sighand`, `signal`, `files`, `fs`, `fs_struct` und `mm_struct`.
- `buffer_init` erzeugt einen Cache für `buffer_heads` und berechnet den Wert der Variablen `max_buffer_heads`, die die maximal mögliche Anzahl von Pufferköpfen im System vorgibt, so, dass diese nie mehr als 10 Prozent des in `ZONE_NORMAL` enthaltenen Speichers beanspruchen.
- `vfs_caches_init` legt Caches für verschiedene Datenstrukturen an, die von der VFS-Schicht benötigt werden.
- `radix_tree_init` legt einen Slab-Cache für `radix_tree_node`-Instanzen an, die in der Speicherverwaltung benötigt werden.
- `page_writeback_init` initialisiert den Flushing-Mechanismus und legt insbesondere den Grenzwert dreckiger Seiten fest, ab dem er in Kraft tritt.
- `proc_root_init` initialisiert den Inodencache des `proc`-Dateisystems, registriert procfs im Kern und erzeugt die zentralen Dateisystemeinträge, beispielsweise `/proc/meminfo`, `/proc/uptime`, `/proc/version` etc.

Suche nach bekannten Systemfehlern

Nicht nur Software hat Fehler – auch bei der Implementierung von Prozessoren passieren gelegentlich diverse Missgeschicke, die dazu führen, dass die Chips nicht so arbeiten, wie sie eigentlich sollten. Glücklicherweise ist es in den meisten Fällen möglich, diese Fehler durch einen Workaround zu umgehen, der das Fehlverhalten kompensiert. Damit dieser aktiviert werden kann, muss der Kern aber erst einmal wissen, ob ein Prozessor mit Fehlern ausgestattet ist oder nicht. Zu diesem Zweck existiert die architekturabhängige Funktion `check_bugs`, die nach entsprechenden Problemen sucht.

Für IA-32 findet sich beispielsweise folgender Code:

include/asm-i386/bugs.h

```
static void __init check_bugs(void)
{
        identify_cpu(&boot_cpu_data);

        check_config();
        check_fpu();
        check_hlt();
        check_popad();
        system_utsname.machine[1] = '' + (boot_cpu_data.x86 > 6 ?
                                          6 : boot_cpu_data.x86);
        alternative_instructions();
}
```

Die letzte Anweisung (`alternative_instructions`) ruft zusätzlich eine Funktion auf, die bestimmte Assembler-Anweisungen je nach Prozessortyp durch modernere, schnellere Varianten austauscht: Dies ermöglicht Distributoren, Kernelimages zu erstellen, die auf einer Vielzahl von Maschinen funktionsfähig sind, ohne auf neuere Features verzichten zu müssen.

Zum Vergleich der CPU-Qualität geben wir noch die `check_bugs`-Routine von Sparc, Sparc64, S390, Alpha, PA-Risc und PPC64 wieder: `:-)`

```
static void check_bugs(void) { }
```

Idle- und `init`-Thread

Die letzten beiden Schritte von `start_kernel` führen folgende Aktionen durch:

- Mit Hilfe von `rest_init` wird ein neuer Thread gestartet, der nach einigen weiteren Initialisierungsaktionen, die wir in diesem Abschnitt besprechen werden, schließlich das Userspace-Initialisierungsprogramm `/sbin/init` aufruft.
- Der bestehende, bisher einzige Kernelthread wird zum Idle-Thread, der immer dann aufgerufen wird, wenn das System nichts anderes zu tun hat.

Die Implementierung von `rest_init` ist in wenigen Zeilen Code erledigt:

init/main.c

```
static void rest_init(void)
{
        kernel_thread(init, NULL, CLONE_KERNEL);
        unlock_kernel();
        cpu_idle();
}
```

Nachdem ein neuer Kernelthread mit der Bezeichnung `init` gestartet worden ist, entsperrt der Kern das bisher aktive Big Kernel Lock mit `unlock_kernel` und verwandelt den bestehenden Thread durch Aufruf von `cpu_idle` in den Idle-Thread, der zum einen möglichst wenig

Systemleistung verbrauchen soll (was besonders für Embedded-Systeme wichtig ist); zum anderen soll er die CPU so schnell wie möglich für ablauffähige Prozesse freimachen, wenn solche zur Verfügung stehen. Auf IA-32 wird dazu folgende Endlosschleife verwendet (die Implementierung auf anderen Architekturen unterscheidet sich nicht wesentlich vom hier gezeigten Code):

arch/i386/kernel/process.c

```
void cpu_idle (void)
{
        /* endless idle loop with no priority at all */
        while (1) {
                while (!need_resched()) {
                        void (*idle)(void) = pm_idle;

                        if (!idle)
                                idle = default_idle;

                        irq_stat[smp_processor_id()].idle_timestamp = jiffies;
                        idle();
                }
                schedule();
        }
}
```

`pm_idle` ist ein Zeiger auf eine Funktion, der je nach Konfiguration des Kernels mit einem anderen Wert belegt ist. Wenn ACPI-Unterstützung mit Support für Power-Management einkompiliert ist, wird beispielsweise `acpi_processor_idle` verwendet, die das System in längerdauernden Inaktivitätszeiten in verschiedene Stromsparmodi versetzt, was nicht nur den Prozessor, sondern praktisch alle Systemkomponenten betrifft und daher eine mit vielen praktischen Problemen beladene Aufgabe ist, auf die wir nicht näher eingehen wollen. `default_idle` ist die Standardfunktion des Kerns, die aufgerufen wird, wenn keine andere Form des Power-Managements einkompiliert wurde; sie macht nichts anderes, als den Prozessor in den Haltezustand zu versetzen, in dem er weniger Strom verbraucht.

Parallel zum Idle-Thread läuft der `init`-Thread, dessen Codeflussdiagramm in Abbildung D.3 zu sehen ist.

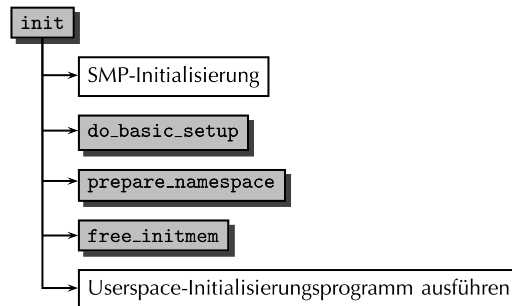

Abbildung D.3: Codeflussdiagramm für `init`

Nachdem der Kern bisher nur eine einzige CPU von Multiprozessorsystemen verwendet hat, ist es an der Zeit, auch die anderen Prozessoren des Systems zu aktivieren. Dies geschieht in drei Schritten:

- `smp_prepare_cpus` sorgt dafür, dass die architekturspezifischen Bootsequenzen für die restlichen CPUs des Systems ausgeführt werden, um die Prozessoren zu aktivieren. Allerdings

werden sie dabei noch nicht in den Scheduling-Mechanismus des Kern eingebunden und können deshalb noch nicht benutzt werden.

- `do_pre_smp_initcalls` ist – entgegen seinem Namen – eine Mischung aus SMP- und Uniprozessor-Initialisierungsroutinen: Aus SMP-Sicht wird vor allem die aus Kapitel 2 („Prozessverwaltung") bekannte Migrationsqueue initialisiert, die zum Verschieben von Prozessen zwischen CPUs verwendet wird. Zusätzlich werden die SoftIRQ-Daemonen gestartet.[2]
- `smp_init` aktiviert die restlichen CPUs im Kernel, weshalb sie ab diesem Zeitpunkt als produktive Mitglieder des Systems verwendet werden können.

Treibersetup Der nächste Schritt von `init` besteht darin, die allgemeine Initialisierung von Treibern und Subsystemen zu starten. Dazu wird die Funktion `do_basic_setup` verwendet, deren Codeflussdiagramm in Abbildung D.4 zu finden ist.

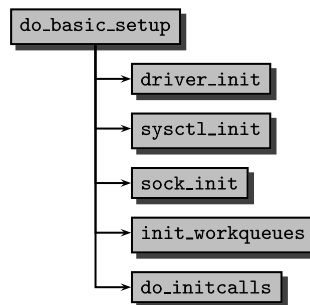

Abbildung D.4: Codeflussdiagramm für `do_basic_setup`

Die ersten vier Funktionen sind teilweise recht umfangreich, aber dennoch nicht besonders interessant, da sie nur weitere Datenstrukturen des Kerns initialisieren, die in den Kapiteln über die jeweiligen Subsysteme behandelt wurden: `driver_init` setzt die Datenstrukturen des allgemeinen Driver Models auf, `sysctl_init` initialisiert die Datenstrukturen des Sysctl-Mechanismus und registriert die entsprechenden Einträge im `proc`-Dateisystem, `sock_init` setzt einige Datenstrukturen und Caches des Netzwerklayers auf, und `init_workqueues` erzeugt schließlich die `keventd`-Workqueue.

Wesentlich interessanter ist `do_initcalls`, die sich darum kümmert, die Treiber-spezifischen Initialisierungsfunktionen aufzurufen. Da der Kern individuell konfigurierbar ist, muss ein Mechanismus bereitgestellt werden, mit dessen Hilfe die aufzurufenden Funktionen erkannt werden können; außerdem muss auch die Reihenfolge festlegt werden, in der sie abgearbeitet werden. Dies bezeichnet man als *Initcall-Mechanismus*, auf den wir nun genauer eingehen werden.

Um die Initialisierungsroutinen kenntlich zu machen und gleichzeitig eine Rangfolge zwischen ihnen zu erstellen, definiert der Kern folgende Makros:

<init.h>
```
#define __define_initcall(level,fn) \
        static initcall_t __initcall_##fn __attribute__
                    ((unused,__section__ (".initcall" level ".init"))) = fn
```

2 Genauer gesagt aktiviert der Kern eine Callback-Funktion, die den Daemonen startet, wenn eine CPU vom Kern aktiviert wird. Da wir auf diesen Mechanismus allerdings nicht genauer eingehen wollen, begnügen wir uns mit der Feststellung, dass im Endeffekt eine Instanz des Daemons für jede CPU gestartet wird.

```
#define core_initcall(fn)               __define_initcall("1",fn)
#define postcore_initcall(fn)           __define_initcall("2",fn)
#define arch_initcall(fn)               __define_initcall("3",fn)
#define subsys_initcall(fn)             __define_initcall("4",fn)
#define fs_initcall(fn)                 __define_initcall("5",fn)
#define device_initcall(fn)             __define_initcall("6",fn)
#define late_initcall(fn)               __define_initcall("7",fn)
```

Die Makros werden verwendet, indem ihnen der Name einer Funktion als Parameter übergeben wird, wie man anhand der Beispiele `device_initcall(time_init_device)` oder `subsys_initcall(pcibios_init)` sehen kann. Dies erzeugt einen Eintrag in der Sektion `.initcall`*`level`*`.init`. Als Eintragstyp wird `initcall_t` verwendet, der wie folgt definiert ist:

```
typedef int (*initcall_t)(void);
```

<init.h>

Es handelt sich um Zeiger auf Funktionen, die kein Argument erwarten und ein Integer-Resultat als Status zurückgeben.

Mit Hilfe des Linkers werden alle Initcall-Sektionen hintereinander in der Binärdatei plaziert, wobei die Rangreihenfolge beachtet wird (wir verwenden hier das Linkerskript für IA-32-Prozessoren; auf anderen Systemen ist die Vorgehensweise aber praktisch identisch):

```
 __initcall_start = .;
 .initcall.init : {
       *(.initcall1.init)
       *(.initcall2.init)
       *(.initcall3.init)
       *(.initcall4.init)
       *(.initcall5.init)
       *(.initcall6.init)
       *(.initcall7.init)
 }
 __initcall_end = .;
```

arch/i386/kernel/vmlinux.lds.S

Anfang und Ende des Initcall-Bereichs werden vom Linker in den Variablen `__initcall_start` und `__initcall_end` festgehalten, die im Kern sichtbar sind und deren Nutzen sich gleich zeigen wird.

Achtung: Durch den gezeigten Mechanismus wird nur die Aufrufreihenfolge zwischen den unterschiedlichen Initcall-*Kategorien* festgelegt. Die Aufrufreihenfolge für die in den einzelnen Kategorien enthaltenen Funktionen ergibt sich implizit durch die Position der definierenden Binärdatei im Linkprozess und kann vom C-Code aus nicht manuell beeinflusst werden.

Durch die Vorarbeiten von Compiler und Linker ist die Aufgabe von `do_initcalls` nicht mehr allzu kompliziert, wie ein Blick in die Kernelquellen zeigt:

```
static void __init do_initcalls(void)
{
        initcall_t *call;
        int count = preempt_count();

        for (call = &__initcall_start; call < &__initcall_end; call++) {
                char *msg;

                if (initcall_debug)
                        printk("calling initcall 0x%p\n", *call);

                (*call)();
```

init/main.c

```
                msg = NULL;
                if (preempt_count() != count) {
                        msg = "preemption imbalance";
                        preempt_count() = count;
                }
                if (irqs_disabled()) {
                        msg = "disabled interrupts";
                        local_irq_enable();
                }
                if (msg) {
                        printk("error in initcall at 0x%p: "
                                "returned with %s\n", *call, msg);
                }
        }

        /* Make sure there is no pending stuff from the initcall sequence */
        flush_scheduled_work();
}
```

Im Wesentlichen iteriert der Code über alle Einträge der `.initcall`-Sektion, deren Grenzen durch die vom Linker automatisch definierten Variablen ersichtlich sind: Die Adressen der Funktionen werden extrahiert und anschließend aufgerufen. Wenn alle Initcalls ausgeführt wurden, flusht der Kern mittels `flush_scheduled_work` alle eventuell verbliebenen Einträge der `keventd`-Workqueue, die von den Routinen angelegt worden sein könnten.

Entfernen der Initialisierungsdaten Funktionen zur Initialisierung von Datenstrukturen oder Geräten werden üblicherweise nur beim Booten des Kerns benötigt; später werden sie nicht mehr aufgerufen. Um dies explizit kenntlich zu machen, definiert der Kern das Attribut `__init`, das der Funktionsdeklaration vorangestellt wird, wie in den vorhergehenden Ausschnitten aus den Kernelquellen oft zu sehen war. Das Attribut ist wie folgt definiert:

<init.h>

```
#define __init          __attribute__ ((__section__ (".init.text")))
#define __initdata      __attribute__ ((__section__ (".init.data")))
```

Mit `__initdata` stellt der Kern zusätzlich eine Möglichkeit bereit, um Daten als Initialisierungsdaten zu deklarieren.

Mit `__init` oder `__initdata` markierte Funktionen werden durch den Linker in einen speziellen Abschnitt der Binärdatei geschrieben (die Linkerskripte auf anderen Architekturen sind praktisch identisch zur gezeigten IA-32-Variante):

arch/i386/kernel/vmlinux.lds.S

```
/* will be freed after init */
. = ALIGN(PAGE_SIZE_asm);               /* Init code and data */
__init_begin = .;
.init.text : {
      _sinittext = .;
      *(.init.text)
      _einittext = .;
}
.init.data : { *(.init.data) }
. = ALIGN(16);
__setup_start = .;
.init.setup : { *(.init.setup) }
__setup_end = .;
__start___param = .;
__param : { *(__param) }
__stop___param = .;
__initcall_start = .;
.initcall.init : {
```

```
        *(.initcall1.init)
        *(.initcall2.init)
        *(.initcall3.init)
        *(.initcall4.init)
        *(.initcall5.init)
        *(.initcall6.init)
        *(.initcall7.init)
  }
  __initcall_end = .;
  __con_initcall_start = .;
  .con_initcall.init : { *(.con_initcall.init) }
  __con_initcall_end = .;
  SECURITY_INIT
  . = ALIGN(4);
  __alt_instructions = .;
  .altinstructions : { *(.altinstructions) }
  __alt_instructions_end = .;
 .altinstr_replacement : { *(.altinstr_replacement) }
  /* .exit.text is discard at runtime, not link time, to deal with references
     &process˙comment(from .altinstructions and .eh˙frame )*/
  .exit.text : { *(.exit.text) }
  .exit.data : { *(.exit.data) }
  . = ALIGN(PAGE_SIZE_asm);
  __initramfs_start = .;
  .init.ramfs : { *(.init.ramfs) }
  __initramfs_end = .;
  . = ALIGN(32);
  __per_cpu_start = .;
  .data.percpu  : { *(.data.percpu) }
  __per_cpu_end = .;
  . = ALIGN(PAGE_SIZE_asm);
  __init_end = .;
  /* freed after init ends here */
```

Es werden auch noch einige andere Sektionen in den Initialisierungsabschnitt aufgenommen, beispielsweise finden sich darin die weiter oben angesprochenen Initcall-Aufrufe. Der Übersichtlichkeit halber wollen wir aber nicht auf alle Arten von Daten und Funktionen eingehen, die der Kern nach Abschluss des Bootvorgangs aus dem Speicher entfernt.

`free_initmem` wird als eine der letzten Aktionen von `init` aufgerufen, um den Speicher zwischen `__init_begin` und `__init_end` freizugeben (der Wert der Variablen wird wie üblich automatisch vom Linker gesetzt):

arch/i386/mm/init.c

```
void free_initmem(void)
{
        unsigned long addr;

        addr = (unsigned long)(&__init_begin);
        for (; addr < (unsigned long)(&__init_end); addr += PAGE_SIZE) {
                ClearPageReserved(virt_to_page(addr));
                set_page_count(virt_to_page(addr), 1);
                free_page(addr);
                totalram_pages++;
        }
        printk (KERN_INFO "Freeing unused kernel memory: %dk freed\n",
                (__init_end - __init_begin) >> 10);
}
```

Obwohl es sich um eine architekturspezifische Funktion handelt, ist sie auf allen unterstützten Architekturen praktisch identisch definiert, weshalb wir uns wie üblich auf die Beschreibung der IA-32-Variante beschränken: Der Code iteriert über die einzelnen Seiten, die von den Initialisierungsdaten belegt sind, und gibt sie mit `free_page` an das Buddy-System zurück. Abschlie-

ßend wird eine Meldung ausgegeben, wie viel Speicher befreit werden konnte; die Menge liegt üblicherweise bei rund 150KiB.

Starten der Userspace-Initialisierung Als letzte Aktion von `init` muss ein Programm gestartet werden, das die Initialisierung im Userspace fortsetzt und das System letztendlich in einen Zustand bringt, der einem Benutzer ermöglicht, damit zu arbeiten. Klassischerweise fällt diese Aufgabe unter Unix und Linux an `/sbin/init`; sollte dies nicht vorhanden sein, probiert der Kern aber verschiedene Alternativen. Dem Kernel kann in der Kommandozeile mit `init=program` der Name eines anderen Programms übergeben werden, dessen Start vor den Defaultmöglichkeiten versucht wird (der Name wird beim Parsen der Kommandozeile in `execute_command` festgehalten). Funktioniert keine der Möglichkeiten, wird eine Kernelpanik ausgelöst, da das System in diesem Fall nicht betriebsfähig ist:

init/main.c

```
static int init(void * unused)
{
...
        if (execute_command)
                execve(execute_command,argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_sh,envp_init);
        panic("No init found.  Try passing init= option to kernel.");
}
```

`argv_init` und `envp_init` legen die Programmargumente und eine minimale Ausführungsumgebung fest, die dem Userspace-Prozess spendiert werden:

init/main.c

```
static char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, };
```