Flex

Un generador de Scanners libre

Jose Pablo Vargas Campos

2013116365

Instituto Tecnológico de Costa Rica

Compiladores e Intérpretes

Semestre 2017

Table of contents

Introducción

Introducción

Flex es una herramienta de análisis lexico desarrollada para la generación de Scanners de lenguajes. Su nombre significa "fast lexical analyzer generator". Es la alternativa gratis y open-source a la herramienta "lex".

Scanning

El proceso de Scanning es el proceso por el cual se identifican los diferentes lexemas de un lenguaje. El proceso es tan simple como la ejecución de un Automata Deterministico Finito. Para la generación del Scanner con Flex se utilizan las expresiones regulares, conocidas como 'RegEx', para indicarle a Flex que construya apartir de las expresiones regulares un DFA en C, el cual luego se usa para adquirir los diferentes lexemas del lenguaje que se planea 'Scannear'.
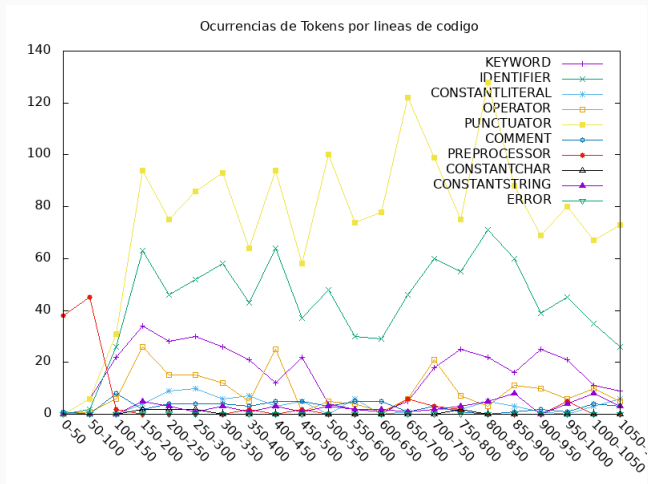
Analisis Léxico

Histograma

A continuación se presenta un histograma el cual indica cuantas veces cada token fue encontrado cada 50 lineas, en el *axis y* se puede ver la cantidad de ocurrencias mientras en el *axis x* se muestra en cual rango de lineas de codigo sucedieron.

Histograma

Codigo fuente

A continuación se presenta el codigo fuente con colores demostrando la división de Tokens.

```c
/*
 *  linux/init/main.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  GK 2/5/95  -  Changed to support mounting root fs via NFS
 *  Added initrd & change_root: Werner Almesberger & Hans Lermen
 *  Moan early if gcc is old, avoiding bogus kernels - Paul Gort
 *  Simplified starting of init:  Michael A. Griffith <grif@acm.
 */
```

```c
#define DEBUG /* Enable initcall_debug */

#include <linux/types.h>
#include <linux/extable.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/binfmts.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/stackprotector.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/delay.h>
```

```
#include <linux/ioport.h>
#include <linux/init.h>
#include <linux/initrd.h>
#include <linux/bootmem.h>
#include <linux/acpi.h>
#include <linux/console.h>
#include <linux/nmi.h>
#include <linux/percpu.h>
#include <linux/kmod.h>
#include <linux/vmalloc.h>
#include <linux/kernel_stat.h>
#include <linux/start_kernel.h>
#include <linux/security.h>
#include <linux/smp.h>
#include <linux/profile.h>
```

```
#include <linux/rcupdate.h>
#include <linux/moduleparam.h>
#include <linux/kallsyms.h>
#include <linux/writeback.h>
#include <linux/cpu.h>
#include <linux/cpuset.h>
#include <linux/cgroup.h>
#include <linux/efi.h>
#include <linux/tick.h>
#include <linux/interrupt.h>
#include <linux/taskstats_kern.h>
#include <linux/delayacct.h>
#include <linux/unistd.h>
#include <linux/rmap.h>
#include <linux/mempolicy.h>
```

```c
#include <linux/key.h>
#include <linux/buffer_head.h>
#include <linux/page_ext.h>
#include <linux/debug_locks.h>
#include <linux/debugobjects.h>
#include <linux/lockdep.h>
#include <linux/kmemleak.h>
#include <linux/pid_namespace.h>
#include <linux/device.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/sched/init.h>
#include <linux/signal.h>
#include <linux/idr.h>
#include <linux/kgdb.h>
```

```c
#include <linux/ftrace.h>
#include <linux/async.h>
#include <linux/kmemcheck.h>
#include <linux/sfi.h>
#include <linux/shmem_fs.h>
#include <linux/slab.h>
#include <linux/perf_event.h>
#include <linux/ptrace.h>
#include <linux/blkdev.h>
#include <linux/elevator.h>
#include <linux/sched_clock.h>
#include <linux/sched/task.h>
#include <linux/sched/task_stack.h>
#include <linux/context_tracking.h>
#include <linux/random.h>
```

```c
#include <linux/list.h>
#include <linux/integrity.h>
#include <linux/proc_ns.h>
#include <linux/io.h>
#include <linux/cache.h>
#include <linux/rodata_test.h>

#include <asm/io.h>
#include <asm/bugs.h>
#include <asm/setup.h>
#include <asm/sections.h>
#include <asm/cacheflush.h>

static int kernel_init(void *);
```

```
extern void init_IRQ(void);
extern void fork_init(void);
extern void radix_tree_init(void);



/*
 * Debug helper: via this flag we know that we are in 'early boo
 * where only the boot processor is running with IRQ disabled.
 * two things - IRQ must not be enabled before the flag is clear
 * operations which are not allowed with IRQ disabled are allowe
 * flag is set.
 */
```

```
bool early_boot_irqs_disabled __read_mostly;

enum system_states system_state __read_mostly;
EXPORT_SYMBOL(system_state);



/*
 * Boot command-line arguments
 */
```

```
#define MAX_INIT_ARGS CONFIG_INIT_ENV_ARG_LIMIT
#define MAX_INIT_ENVS CONFIG_INIT_ENV_ARG_LIMIT

extern void time_init(void);


/* Default late time init is NULL. archs can override this later

void (*__initdata late_time_init)(void);



/* Untouched command line saved by arch-specific code. */
```

```c
char __initdata boot_command_line[COMMAND_LINE_SIZE];


/* Untouched saved command line (eg. for /proc) */


char *saved_command_line;


/* Command line for parameter parsing */


static char *static_command_line;
```

```c
/* Command line for per-initcall parameter parsing */


static char *initcall_command_line;

static char *execute_command;
static char *ramdisk_execute_command;



/*
 * Used to generate warnings if static_key manipulation function
 * before jump_label_init is called.
 */
```

```
bool static_key_initialized __read_mostly;
EXPORT_SYMBOL_GPL(static_key_initialized);



/*
 * If set, this is an indication to the drivers that reset the u
 * device before going ahead with the initialization otherwise d
 * rely on the BIOS and skip the reset operation.
 *
 * This is useful if kernel is booting in an unreliable environm
 * For ex. kdump situation where previous kernel has crashed, BI
 * skipped and devices will be in unknown state.
 */
```

```c
unsigned int reset_devices;
EXPORT_SYMBOL(reset_devices);

static int __init set_reset_devices(char *str)
{
    reset_devices = 1;
    return 1;
}

__setup("reset_devices", set_reset_devices);

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

```c
"TERM=linux", NULL, };
static const char *panic_later, *panic_param;

extern const struct obs_kernel_param __setup_start[],
__setup_end[];

static bool __init obsolete_checksetup(char *line)
{
        const struct obs_kernel_param *p;
        bool had_early_param = false;

        p = __setup_start;
        do {
                int n = strlen(p->str);
                if (parameqn(line, p->str, n)) {
```

```c
                    if (p->early) {


    /* Already done in parse_early_param?
     * (Needs exact match on param part).
     * Keep iterating, as we can have early
     * params and __setups of same names 8( */



                    if (line[n] == '\0' || line[n] == '=')
                            had_early_param = true;
            } else if (!p->setup_func) {
                    pr_warn("Parameter %s is obsolete, ignored"
                            p->str);
                    return true;
```

```
                } else if (p->setup_func(line + n))
                        return true;
            }
            p++;
    } while (p < __setup_end);

    return had_early_param;
}



/*
 * This should be approx 2 Bo*oMips to start (note initial shift
 * still work even if initially too large, it will just take sli
 */
```

```c
unsigned long loops_per_jiffy = (1<<12);
EXPORT_SYMBOL(loops_per_jiffy);

static int __init debug_kernel(char *str)
{
        console_loglevel = CONSOLE_LOGLEVEL_DEBUG;
        return 0;
}

static int __init quiet_kernel(char *str)
{
        console_loglevel = CONSOLE_LOGLEVEL_QUIET;
        return 0;
}
```

```c
early_param("debug", debug_kernel);
early_param("quiet", quiet_kernel);

static int __init loglevel(char *str)
{
        int newlevel;



/*
 * Only update loglevel value when a correct setting was passed,
 * to prevent blind crashes (when loglevel being set to 0) that
 * are quite hard to debug
 */
```

```c
        if (get_option(&str, &newlevel)) {
                console_loglevel = newlevel;
                return 0;
        }

        return -EINVAL;
}

early_param("loglevel", loglevel);



/* Change NUL term back to "=", to make "param" the whole string
```

```
static int __init repair_env_string(char *param, char *val,
                        const char *unused, void *arg)
{
      if (val) {


/* param=val or param="val"? */

              if (val == param+strlen(param)+1)
                    val[-1] = '=';
              else if (val == param+strlen(param)+2) {
                    val[-2] = '=';
```

```
                memmove(val-1, val, strlen(val)+1);
                val--;
        } else
                BUG();
    }
    return 0;
}



/* Anything after -- gets handed straight to init. */
```

```
static int __init set_init_arg(char *param, char *val,
                  const char *unused, void *arg)
{
      unsigned int i;

      if (panic_later)
            return 0;

      repair_env_string(param, val, unused, NULL);

      for (i = 0; argv_init[i]; i++) {
            if (i == MAX_INIT_ARGS) {
                  panic_later = "init";
                  panic_param = param;
```

```
                return 0;
            }
        }
        argv_init[i] = param;
        return 0;
    }



/*
 * Unknown boot options get handed to init, unless they look lik
 * unused parameters (modprobe will find them in /proc/cmdline).
 */
```

```c
static int __init unknown_bootoption(char *param, char *val,
                        const char *unused, void *arg)
{
    repair_env_string(param, val, unused, NULL);



    /* Handle obsolete-style parameters */


    if (obsolete_checksetup(param))
            return 0;
```

```
/* Unused module parameter. */


      if (strchr(param, '.') && (!val || strchr(param, '.') <
val))
           return 0;

      if (panic_later)
           return 0;

      if (val) {


/* Environment option */
```

```
unsigned int i;
for (i = 0; envp_init[i]; i++) {
    if (i == MAX_INIT_ENVS) {
        panic_later = "env";
        panic_param = param;
    }
    if (!strncmp(param, envp_init[i], val -
param))

        break;
}
envp_init[i] = param;
} else {
```

```c
/* Command line option */


            unsigned int i;
            for (i = 0; argv_init[i]; i++) {
                    if (i == MAX_INIT_ARGS) {
                            panic_later = "init";
                            panic_param = param;
                    }
            }
            argv_init[i] = param;
        }
        return 0;
}
```

```c
static int __init init_setup(char *str)
{
        unsigned int i;

        execute_command = str;


/*
 * In case LILO is going to boot us with default command line,
 * it prepends "auto" before the whole cmdline which makes
 * the shell think it should execute a script with such name.
 * So we ignore all arguments entered _before_ init=... [MJ]
 */
```

```
        for (i = 1; i < MAX_INIT_ARGS; i++)
                argv_init[i] = NULL;
        return 1;
}
__setup("init=", init_setup);

static int __init rdinit_setup(char *str)
{
        unsigned int i;

        ramdisk_execute_command = str;


/* See "auto" comment in init_setup */
```

```c
        for (i = 1; i < MAX_INIT_ARGS; i++)
                argv_init[i] = NULL;
        return 1;
}
__setup("rdinit=", rdinit_setup);

#ifndef CONFIG_SMP
static const unsigned int setup_max_cpus = NR_CPUS;
static inline void setup_nr_cpu_ids(void) { }
static inline void smp_prepare_cpus(unsigned int maxcpus) { }
#endif
```

```
/*
 * We need to store the untouched command line for future refere
 * We also need to store the touched command line since the para
 * parsing is performed in place, and we should allow a componen
 * store reference of name/value for future reference.
 */


static void __init setup_command_line(char *command_line)
{
        saved_command_line =
                memblock_virt_alloc(strlen(boot_command_line) +
1, 0);
        initcall_command_line =
                memblock_virt_alloc(strlen(boot_command_line) +
```

```
1, 0);
      static_command_line =
memblock_virt_alloc(strlen(command_line) + 1, 0);
      strcpy(saved_command_line, boot_command_line);
      strcpy(static_command_line, command_line);
}




/*
 * We need to finalize in a non-__init function or else race con
 * between the root thread and the init thread may cause start_k
 * be reaped by free_initmem before the root thread has proceede
 * cpu_idle.
 *
```

```
 * gcc-3.4 accidentally inlines this function, so use noinline.
 */




static __initdata DECLARE_COMPLETION(kthreadd_done);

static noinline void __ref rest_init(void)
{
    struct task_struct *tsk;
    int pid;

    rcu_scheduler_starting();
```

```
/*
 * We need to spawn init first so that it obtains pid 1, however
 * the init task will end up wanting to create kthreads, which,
 * we schedule it before we create kthreadd, will OOPS.
 */


    pid = kernel_thread(kernel_init, NULL, CLONE_FS);


/*
 * Pin init on the boot CPU. Task migration is not properly work
 * until sched_init_smp() has been run. It will set the allowed
 * CPUs for init to the non isolated CPUs.
 */
```

```
    rcu_read_lock();
    tsk = find_task_by_pid_ns(pid, &init_pid_ns);
    set_cpus_allowed_ptr(tsk,
cpumask_of(smp_processor_id()));
    rcu_read_unlock();

    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS |
CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid,
&init_pid_ns);
    rcu_read_unlock();
```

```
/*
 * Enable might_sleep() and smp_processor_id() checks.
 * They cannot be enabled earlier because with CONFIG_PRREMPT=y
 * kernel_thread() would trigger might_sleep() splats. With
 * CONFIG_PREEMPT_VOLUNTARY=y the init task might have scheduled
 * already, but it's stuck on the kthreadd_done completion.
 */
```

```c
        system_state = SYSTEM_SCHEDULING;

        complete(&kthreadd_done);



    /*
     * The boot idle thread must execute schedule()
     * at least once to get things moving:
     */


        schedule_preempt_disabled();
```

```
/* Call into cpu_idle with preempt disabled */



        cpu_startup_entry(CPUHP_ONLINE);
}




/* Check for early params. */



static int __init do_early_param(char *param, char *val,
                        const char *unused, void *arg)
{
        const struct obs_kernel_param *p;
```

```
for (p = __setup_start; p < __setup_end; p++) {
        if ((p->early && parameq(param, p->str)) ||
        (strcmp(param, "console") == 0 &&
        strcmp(p->str, "earlycon") == 0)
        ) {
                if (p->setup_func(val) != 0)
                        pr_warn("Malformed early option '%s'\n",
param);
        }
}


/* We accept everything at this stage. */
```

```
        return 0;
}

void __init parse_early_options(char *cmdline)
{
        parse_args("early options", cmdline, NULL, 0, 0, 0,
NULL,
                do_early_param);
}


/* Arch code calls this early on, or if not, just before other p
```

```c
void __init parse_early_param(void)
{
        static int done __initdata;
        static char tmp_cmdline[COMMAND_LINE_SIZE] __initdata;

        if (done)
                return;




/* All fall through to do_early_param. */
```

```
        strlcpy(tmp_cmdline, boot_command_line,
COMMAND_LINE_SIZE);
        parse_early_options(tmp_cmdline);
        done = 1;
}

void __init __weak arch_post_acpi_subsys_init(void) { }

void __init __weak smp_setup_processor_id(void)
{
}

# if THREAD_SIZE >= PAGE_SIZE
void __init __weak thread_stack_cache_init(void)
```

```
{
}
#endif

void __init __weak mem_encrypt_init(void) { }



/*
 * Set up kernel memory allocators
 */
```

```
static void __init mm_init(void)
{


/*
 * page_ext requires contiguous pages,
 * bigger than MAX_ORDER unless SPARSEMEM.
 */
```

```c
        page_ext_init_flatmem();
        mem_init();
        kmem_cache_init();
        pgtable_init();
        vmalloc_init();
        ioremap_huge_init();
}

asmlinkage __visible void __init start_kernel(void)
{
        char *command_line;
        char *after_dashes;
```

```
        set_task_stack_end_magic(&init_task);
        smp_setup_processor_id();
        debug_objects_early_init();

        cgroup_init_early();

        local_irq_disable();
        early_boot_irqs_disabled = true;



    /*
     * Interrupts are still disabled. Do necessary setups, then
     * enable them.
     */
```

```c
        boot_cpu_init();
        page_address_init();
        pr_notice("%s", linux_banner);
        setup_arch(&command_line);


/*
 * Set up the the initial canary and entropy after arch
 * and after adding latent and command line entropy.
 */
```

```
add_latent_entropy();
add_device_randomness(command_line,
strlen(command_line));
boot_init_stack_canary();
mm_init_cpumask(&init_mm);
setup_command_line(command_line);
setup_nr_cpu_ids();
setup_per_cpu_areas();
boot_cpu_state_init();
smp_prepare_boot_cpu();

/* arch-specific boot-cpu hooks */
```

```c
        build_all_zonelists(NULL);
        page_alloc_init();

        pr_notice("Kernel command line: %s\n",
boot_command_line);
        parse_early_param();
        after_dashes = parse_args("Booting kernel",
                           static_command_line, __start___param,
                           __stop___param - __start___param,
                           -1, -1, NULL, &unknown_bootoption);
        if (!IS_ERR_OR_NULL(after_dashes))
                parse_args("Setting init args", after_dashes,
NULL, 0, -1, -1,
```

```
                    NULL, set_init_arg);

        jump_label_init();



    /*
     * These use large bootmem allocations and must precede
     * kmem_cache_init()
     */
```

```c
setup_log_buf(0);
pidhash_init();
vfs_caches_init_early();
sort_main_extable();
trap_init();
mm_init();

ftrace_init();




/* trace_printk can be enabled here */
```

```
        early_trace_init();



    /*
     * Set up the scheduler prior starting any interrupts (such as t
     * timer interrupt). Full topology setup happens at smp_init()
     * time - but meanwhile we still have a functioning scheduler.
     */


        sched_init();
```

```
/*
 * Disable preemption - early bootup scheduling is extremely
 * fragile until we cpu_idle() for the first time.
 */


        preempt_disable();
        if (WARN(!irqs_disabled(),
                "Interrupts were enabled *very* early, fixing it\n"))
                local_irq_disable();
        radix_tree_init();
```

```
/*
 * Allow workqueue creation and work item queueing/cancelling
 * early.  Work item execution depends on kthreads and starts af
 * workqueue_init().
 */


        workqueue_init_early();

        rcu_init();



/* Trace events are available after this */
```

```
        trace_init();

        context_tracking_init();


   /* init some links before init_ISA_irqs() */


        early_irq_init();
        init_IRQ();
        tick_init();
        rcu_init_nohz();
        init_timers();
        hrtimers_init();
```

```
        softirq_init();
        timekeeping_init();
        time_init();
        sched_clock_postinit();
        printk_safe_init();
        perf_event_init();
        profile_init();
        call_function_init();
        WARN(!irqs_disabled(),
    "Interrupts were enabled early\n");
        early_boot_irqs_disabled = false;
        local_irq_enable();

        kmem_cache_init_late();
```

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


        console_init();
        if (panic_later)
                panic("Too many boot %s vars at '%s'",
panic_later,
                panic_param);
```

```
        lockdep_info();



/*
 * Need to run this when irqs are enabled, because it wants
 * to self-test [hard/soft]-irqs on/off lock inversion bugs
 * too:
 */


        locking_selftest();
```

```
/*
 * This needs to be called before any devices perform DMA
 * operations that might use the SWIOTLB bounce buffers. It will
 * mark the bounce buffers as decrypted so that their usage will
 * not cause "plain-text" data to be decrypted when accessed.
 */


    mem_encrypt_init();

#ifdef CONFIG_BLK_DEV_INITRD
    if (initrd_start && !initrd_below_start_ok &&
    page_to_pfn(virt_to_page((void *)initrd_start)) <
min_low_pfn) {
        pr_crit("initrd overwritten (0x%08lx < 0x%08lx) - dis
```

```
            page_to_pfn(virt_to_page((void *)initrd_start)),
            min_low_pfn);
            initrd_start = 0;
        }
#endif
        page_ext_init();
        kmemleak_init();
        debug_objects_mem_init();
        setup_per_cpu_pageset();
        numa_policy_init();
        if (late_time_init)
                late_time_init();
        calibrate_delay();
        pidmap_init();
        anon_vma_init();
```

```
        acpi_early_init();
#ifdef CONFIG_X86
        if (efi_enabled(EFI_RUNTIME_SERVICES))
                efi_enter_virtual_mode();
#endif
#ifdef CONFIG_X86_ESPFIX64


        /* Should be run before the first non-init thread is created */


        init_espfix_bsp();
#endif
        thread_stack_cache_init();
        cred_init();
```

```
fork_init();
proc_caches_init();
buffer_init();
key_init();
security_init();
dbg_late_init();
vfs_caches_init();
pagecache_init();
signals_init();
proc_root_init();
nsfs_init();
cpuset_init();
cgroup_init();
taskstats_init_early();
delayacct_init();
```

```
check_bugs();

acpi_subsystem_init();
arch_post_acpi_subsys_init();
sfi_init_late();

if (efi_enabled(EFI_RUNTIME_SERVICES)) {
        efi_free_boot_services();
}
```

```
/* Do the rest non-__init'ed, we're now alive */
```

```c
        rest_init();
}



/* Call all constructor functions linked into the kernel. */


static void __init do_ctors(void)
{
#ifdef CONFIG_CONSTRUCTORS
        ctor_fn_t *fn = (ctor_fn_t *) __ctors_start;

        for (; fn < (ctor_fn_t *) __ctors_end; fn++)
```

```c
                (*fn)();
#endif
}

bool initcall_debug;
core_param(initcall_debug, initcall_debug, bool, 0644);

#ifdef CONFIG_KALLSYMS
struct blacklist_entry {
     struct list_head next;
     char *buf;
};

static __initdata_or_module
LIST_HEAD(blacklisted_initcalls);
```

```c
static int __init initcall_blacklist(char *str)
{
      char *str_entry;
      struct blacklist_entry *entry;



      /* str argument is a comma-separated list of functions */


      do {
            str_entry = strsep(&str, ",");
            if (str_entry) {
                  pr_debug("blacklisting initcall %s\n",
```

```
str_entry);
                  entry = alloc_bootmem(sizeof(*entry));
                  entry->buf =
alloc_bootmem(strlen(str_entry) + 1);
                  strcpy(entry->buf, str_entry);
                  list_add(&entry->next,
&blacklisted_initcalls);
            }
      } while (str_entry);
```

```
        return 0;
}

static bool __init_or_module
initcall_blacklisted(initcall_t fn)
{
        struct blacklist_entry *entry;
        char fn_name[KSYM_SYMBOL_LEN];
        unsigned long addr;

        if (list_empty(&blacklisted_initcalls))
                return false;
```

```
      addr = (unsigned long)
dereference_function_descriptor(fn);
      sprint_symbol_no_offset(fn_name, addr);



/*
 * fn will be "function_name [module_name]" where [module_name]
 * displayed for built-in init functions.  Strip off the [module
 */
```

```c
        strreplace(fn_name, ' ', '\0');

        list_for_each_entry(entry, &blacklisted_initcalls,
next) {
                if (!strcmp(fn_name, entry->buf)) {
                        pr_debug("initcall %s blacklisted\n",
fn_name);

                        return true;
                }
        }

        return false;
}
#else
```

```c
static int __init initcall_blacklist(char *str)
{
        pr_warn("initcall_blacklist requires CONFIG_KALLSYMS\n");
        return 0;
}

static bool __init_or_module
initcall_blacklisted(initcall_t fn)
{
        return false;
}
#endif
__setup("initcall_blacklist=", initcall_blacklist);

static int __init_or_module
```

```
do_one_initcall_debug(initcall_t fn)
{
      ktime_t calltime, delta, rettime;
      unsigned long long duration;
      int ret;

      printk(KERN_DEBUG "calling  %pF @ %i\n", fn,
task_pid_nr(current));
      calltime = ktime_get();
      ret = fn();
      rettime = ktime_get();
      delta = ktime_sub(rettime, calltime);
      duration = (unsigned long long) ktime_to_ns(delta) >>
10;
      printk(KERN_DEBUG
```

```
    "initcall %pF returned %d after %lld usecs\n",
            fn, ret, duration);


    return ret;
}


int __init_or_module do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    int ret;
    char msgbuf[64];

    if (initcall_blacklisted(fn))
            return -EPERM;
```

```
if (initcall_debug)
        ret = do_one_initcall_debug(fn);
else
        ret = fn();

msgbuf[0] = 0;

if (preempt_count() != count) {
        sprintf(msgbuf, "preemption imbalance ");
        preempt_count_set(count);
}
if (irqs_disabled()) {
        strlcat(msgbuf, "disabled interrupts ",
sizeof(msgbuf));
        local_irq_enable();
```

```
        }
        WARN(msgbuf[0], "initcall %pF returned with %s\n",
fn, msgbuf);

        add_latent_entropy();
        return ret;
}


extern initcall_t __initcall_start[];
extern initcall_t __initcall0_start[];
extern initcall_t __initcall1_start[];
extern initcall_t __initcall2_start[];
extern initcall_t __initcall3_start[];
extern initcall_t __initcall4_start[];
```

```
extern initcall_t __initcall5_start[];
extern initcall_t __initcall6_start[];
extern initcall_t __initcall7_start[];
extern initcall_t __initcall_end[];

static initcall_t *initcall_levels[] __initdata = {
        __initcall0_start,
        __initcall1_start,
        __initcall2_start,
        __initcall3_start,
        __initcall4_start,
        __initcall5_start,
        __initcall6_start,
        __initcall7_start,
        __initcall_end,
```

```
};



/* Keep these in sync with initcalls in include/linux/init.h */


static char *initcall_level_names[] __initdata = {
    "early",
    "core",
    "postcore",
    "arch",
    "subsys",
    "fs",
    "device",
```

```
        "late",
};

static void __init do_initcall_level(int level)
{
        initcall_t *fn;

        strcpy(initcall_command_line, saved_command_line);
        parse_args(initcall_level_names[level],
                initcall_command_line, __start___param,
                __stop___param - __start___param,
                level, level,
                NULL, &repair_env_string);

        for (fn = initcall_levels[level]; fn <
```

```
initcall_levels[level+1]; fn++)
            do_one_initcall(*fn);
}

static void __init do_initcalls(void)
{
      int level;

      for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1;
level++)
            do_initcall_level(level);
}
```

```c
/*
 * Ok, the machine is now initialized. None of the devices
 * have been touched yet, but the CPU subsystem is up and
 * running, and memory and process management works.
 *
 * Now we can finally start doing some real work..
 */


static void __init do_basic_setup(void)
{
        cpuset_init_smp();
        shmem_init();
        driver_init();
        init_irq_proc();
```

```
        do_ctors();
        usermodehelper_enable();
        do_initcalls();
}

static void __init do_pre_smp_initcalls(void)
{
        initcall_t *fn;

        for (fn = __initcall_start; fn < __initcall0_start;
fn++)
                do_one_initcall(*fn);
}
```

```
/*
 * This function requests modules which should be loaded by defa
 * called twice right after initrd is mounted and right before i
 * exec'd.  If such modules are on either initrd or rootfs, they
 * loaded before control is passed to userland.
 */


void __init load_default_modules(void)
{
        load_default_elevator_module();
}

static int run_init_process(const char *init_filename)
{
```

```
        argv_init[0] = init_filename;
        return do_execve(getname_kernel(init_filename),
                (const char __user *const __user *)argv_init,
                (const char __user *const __user *)envp_init);
}

static int try_to_run_init_process(const char
*init_filename)
{
        int ret;

        ret = run_init_process(init_filename);

        if (ret && ret != -ENOENT) {
                pr_err("Starting init: %s exists but couldn't execute
```

```c
                init_filename, ret);
        }


        return ret;
}


static noinline void __init kernel_init_freeable(void);

#if defined(CONFIG_STRICT_KERNEL_RWX) || defined(CONFIG_STRICT_M
bool rodata_enabled __ro_after_init = true;
static int __init set_debug_rodata(char *str)
{
        return strtobool(str, &rodata_enabled);
}
__setup("rodata=", set_debug_rodata);
```

```
#endif

#ifdef CONFIG_STRICT_KERNEL_RWX
static void mark_readonly(void)
{
    if (rodata_enabled) {
        mark_rodata_ro();
        rodata_test();
    } else
        pr_info("Kernel memory protection disabled.\n");
}
#else
static inline void mark_readonly(void)
{
    pr_warn("This architecture does not have kernel memory prot
```

```
        }
        #endif

        static int __ref kernel_init(void *unused)
        {
                int ret;

                kernel_init_freeable();


                /* need to finish all async __init code before freeing the memor
```

```
async_synchronize_full();
ftrace_free_init_mem();
free_initmem();
mark_readonly();
system_state = SYSTEM_RUNNING;
numa_default_policy();
```

```
rcu_end_inkernel_boot();

if (ramdisk_execute_command) {
      ret = run_init_process(ramdisk_execute_command);
      if (!ret)
            return 0;
      pr_err("Failed to execute %s (error %d)\n",
      ramdisk_execute_command, ret);
}
```

```
/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
 */


        if (execute_command) {
                ret = run_init_process(execute_command);
                if (!ret)
                        return 0;
                panic("Requested init %s failed (error %d).",
                execute_command, ret);
        }
```

```c
        if (!try_to_run_init_process("/sbin/init") ||
        !try_to_run_init_process("/etc/init") ||
        !try_to_run_init_process("/bin/init") ||
        !try_to_run_init_process("/bin/sh"))
                return 0;

        panic("No working init found.  Try passing init= option to
        "See Linux Documentation/admin-guide/init.rst for guidance.
}

static noinline void __init kernel_init_freeable(void)
{
```

```
/*
 * Wait until kthreadd is all set-up.
 */


        wait_for_completion(&kthreadd_done);



/* Now the scheduler is fully set up and can do blocking allocat


        gfp_allowed_mask = __GFP_BITS_MASK;
```

```
/*
 * init can allocate pages on any node
 */


        set_mems_allowed(node_states[N_MEMORY]);

        cad_pid = task_pid(current);

        smp_prepare_cpus(setup_max_cpus);

        workqueue_init();

        init_mm_internals();
```

```
        do_pre_smp_initcalls();
        lockup_detector_init();

        smp_init();
        sched_init_smp();

        page_alloc_init_late();

        do_basic_setup();



    /* Open the /dev/console on the rootfs, this should never fail *
```

```c
        if
(sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
            pr_err("Warning: unable to open an initial console.\n

        (void) sys_dup(0);
        (void) sys_dup(0);


/*
 * check if there is an early userspace init.  If yes, let it do
 * the work
 */
```

```
        if (!ramdisk_execute_command)
                ramdisk_execute_command = "/init";

        if (sys_access((const char __user *)
ramdisk_execute_command, 0) != 0) {
                ramdisk_execute_command = NULL;
                prepare_namespace();
        }
```

```
/*
 * Ok, we have completed the initial bootup, and
 * we're essentially up and running. Get rid of the
 * initmem segments and start the user-mode stuff..
 *
 * rootfs is available now, try loading the public keys
 * and default modules
 */



    integrity_load_keys();
    load_default_modules();
}
```

¿Preguntas?