The Linux Kernel

Luca Abeni, Claudio Scordino

Outline

The Kernel Source Tree

- About 16,000 source files
- Main directories in the kernel source:

`arch/`	Architecture-specific code
`Documentation/`	Kernel source documentation
`drivers/`	Device drivers
`fs/`	File systems
`include/`	Kernel headers
`kernel/`	**Core**
`net/`	Networking

Differences wrt normal user-space applications

- Not a single entry point: a different entry point for any type of interrupt recognized by the kernel
- No memory protection
  - No control over illegal memory access
- Synchronization and concurrency are major concerns
  - Susceptible to race conditions on shared resources!
  - Use spinlocks and semaphores.
- No libraries to link to
  - Never include the usual header files, like `<stdio.h>`
- A fault can crash the whole system
- No debuggers
- Small stack: 4 or 8 KB
  - Do not use large variables
  - Allocate large structures at runtime (`kmalloc`)
- No floating point arithmetic

Programming language

- Like all Unix-like OSs, Linux is coded mostly in C
- No access to the C library
  - No `printf`: use `printk`:
    ```
    printk(KERN_ERR "This is an error!");
    ```
- Not coded in ANSI C
  - Both ISO C99 and GNU C extensions used
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:
    ```
    static inline void foo (...);
    ```
  - Branch annotation:
    ```
    if (likely(pippo)) {
    /*...*/
    }
    ```

Programming language

- Like all Unix-like OSs, Linux is coded mostly in C
- No access to the C library
  - No `printf`: use `printk`:
    `printk(KERN_ERR "This is an error!");`
- Not coded in ANSI C
  - Both ISO C99 and GNU C extensions used
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:
    `static inline void foo (...);`
  - Branch annotation:

```
if (likely(pippo)) {
/*...*/
}
```

Programming language (2)

- Few small critical functions coded in Assembly (around 10% of the code)
  - Architecture-dependent code placed in `linux/arch`
  - The symbolic link `linux/include/asm` identifies all architecture-dependent header files
  - Inline assembly (`asm` primitive)

Loadable Kernel Modules

- Linux provides the ability of inserting (and removing) services provided by the kernel at runtime
- Every piece of code that can be dynamically loaded (and unloaded) is called **Kernel Module**

Loadable Kernel Modules (2)

- A kernel module provides a new service (or services) available to users
- Event-driven programming:
  - Once inserted, a module just registers itself in order to serve future requests
  - The initialization function terminates immediately
- Once a module is loaded and the new service registered
  - The service can be used by all the processes, as long as the module is in memory
  - The module can access all the kernel's public symbols
- After unloading a module, the service is no longer available
- In the 2.6 series, modules have extensions `.ko`

Loadable Kernel Modules (3)

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - `EXPORT_SYMBOL(name);`
  - `EXPORT_SYMBOL_GPL(name);`

    makes the symbol available only to GPL-licensed modules

Loadable Kernel Modules (3)

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - `EXPORT_SYMBOL(name);`
  - `EXPORT_SYMBOL_GPL(name);`
    makes the symbol available only to GPL-licensed modules

Why using kernel modules

- Not all kernel services of features are required every time into the kernel: a module can be loaded only when it is necessary, saving memory
- Easier development: kernel modules can be loaded and unloaded several times, allowing to test and debug the code without rebooting the machine.

How to write a kernel module

Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
   - Modify the `Kconfig` and the main `Makefile`
   - Create a patch for each new kernel version
2. Write the code in a separate directory, without modifying any file in the main source tree
   - More flexible
   - In the 2.6 series, the modules are linked against object files in the main source tree:
     - ⇒ The kernel must be already configured and compiled

How to write a kernel module

Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
   - Modify the `Kconfig` and the main `Makefile`
   - Create a patch for each new kernel version
2. Write the code in a separate directory, without modifying any file in the main source tree
   - More flexible
   - In the 2.6 series, the modules are linked against object files in the main source tree:
     - ⇒ The kernel must be already configured and compiled

Loading/unloading a module

- Only the superuser can load and unload modules
- insmod inserts a module and its data into the kernel.
- The kernel function sys_init_module:
  1. Allocates (through vmalloc) memory to hold the module
  2. Copies the module into that memory region
  3. Resolves kernel references in the module via the kernel symbol table (works like the linker ld)
  4. Calls the module's initialization function
- modprobe works as insmod, but it also checks module dependencies. It can only load a module contained in the /lib/modules/ directory
- rmmod removes a loaded module and all its services
- lsmod lists modules currently loaded in the kernel
  - Works through /proc/modules

The Makefile

- The `Makefile` uses the extended GNU *make* syntax
- Structure of the `Makefile`:

```
## Name of the module:
obj-m = mymodule.o

## Source files:
example-objs = file1.o file2.o
```

- Command line:

```
make -C kernel_dir M=`pwd` modules
```

Example 1: the include part

- We now see how to write a simple module that writes "Hello World" at module insertion/removal
- For a simple module we need to include at least the following

  ```
  #include <linux/module.h>
  #include <linux/kernel.h>
  #include <linux/init.h>
  ```

  that define some essential macros and function prototypes.

Example 1: the init function

- Function called when the module is inserted:

```
static int __init hello_init(void)
{
   printk(KERN_ALERT "Hello world!\n");
   return 0;
}

module_init(hello_init);
```

- The function is defined `static` because it shouldn't be visible outside of the file
- The `__init` token tells the kernel that the function can be dropped after the module is loaded
  - Similar tag for data: `__initdata`
- The `module_init` macro specifies which function must be called when the module is inserted

Example 1: the cleanup function

- The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded

```
static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world!\n");
}

module_exit(hello_exit);
```

- The `__exit` token tells the compiler that the function will be called only during the unloading stage (the compiler puts this function in a special section of the ELF file)
- The `module_exit` macro specifies which function must be called when the module is removed
- It **must** release any resource and undo everything the *init* function built up
- If it is not defined, the kernel does not allow module unloading

Other information

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also `"GPL v2"`, `"GPL and additional rights"`, `"Dual BSD/GPL"`, `"Dual MPL/GPL"` and `"Proprietary"`
- Convention: put all information at the end of the file

Example 2: using the proc filesystem

- We now see how to write a module that creates a new entry in the proc filesystem
- The entry will be created during the initialization phase and removed by the cleanup function
- Since modifications to the proc filesystem cannot be done at user level, we have to work at kernel level. A kernel module is perfect for this job!
- Requires

```
#include <linux/proc_fs.h>
```

Example 2: creating a directory

- A new directory is created through

  `struct proc_dir_entry* proc_mkdir(const char *name, struct proc_dir_entry * parent);`

- It returns a pointer to

  `struct proc_dir_entry* lkh_pde;`

Example 2: the code

```
static struct proc_dir_entry *lkh_pde;

static int __init ex2_init(void)
{
    lkh_pde = proc_mkdir("lkh", NULL);
    if (!lkh_pde) {
        printk(KERN_ERR "%s: error creating proc_dir!\n", \
            MODULE_NAME);
    return -1;
    }
    printk("Proc dir created!\n");
    return 0;
}

static void __exit ex2_exit(void)
{
    remove_proc_entry("lkh", NULL);
    printk("Proc dir removed!\n");
}
```

Module parameters

- Both `insmod` and `modprobe` accept parameters given at loading time
- Require `#include <linux/moduleparam.h>`
- A module parameter is defined through a macro:

```
static int myvar = 13;
module_param(myvar, int, SIRUGO);
```

  - All parameters should be given a default value
  - The last argument is a permission bit-mask (see `linux/stat.h`)
  - The macro should be placed outside of any function

Module parameters (2)

- Supported types: `bool`, `charp`, `int`, `long`, `short`, `uint`, `ulong`, `ushort`
- The module ex3 can be loaded assigning a value to the parameter myvar:

  `insmod ex3 myvar=27`
- Another macro allows to accept array parameters:
  `module_param_array(name, type, num, permission);`
  - The module loader refuses to accept more values than will fit in the array

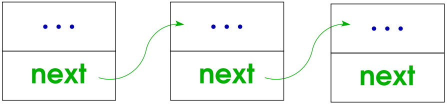
Example: Kernel Linked Lists

- Data structure that stores a certain amount of **nodes**
- The nodes can be dynamically created, added and removed at runtime
  - Number of nodes unknown at compile time
  - Different from array
- For this reason, the nodes are linked together
  - Each node contains at least one pointer to another element

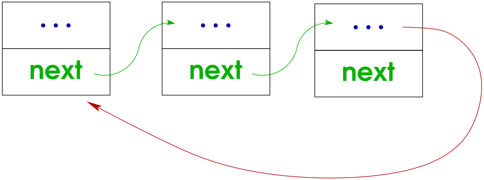
Singly linked lists

```
struct list_element {
    int data;
    struct list_element *next;
};
```

- Singly linked list:



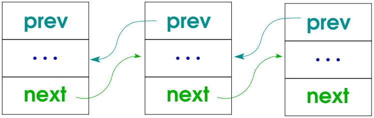
- Circular singly linked list:



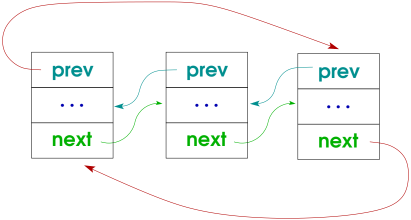
Doubly linked lists

```
struct list_element {
    int data;
    struct list_element *next;
    struct list_element *prev;
};
```

- Doubly linked list:



- Circular doubly linked list:



Kernel's linked list implementation

- Circular doubly linked list
- No head pointer: does not matter where you start...
  - All individual nodes are called *list heads*
- Declared in `linux/list.h`
- Data structure:

```
struct list_head {
    struct list_head* next;
    struct list_head* prev;
};
```

- No locking: your responsibility to implement a locking scheme

Defining linked lists

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists: Example

How to remove from the linked list the node having value 7:

```
struct my_node {
    struct list_head klist;
    int value;
};

struct list_head my_list;

struct list_head *h;
list_for_each_safe(h, &my_list)
    if ((list_entry(h, struct my_node, klist))->value == 7)
        list_del(h);
```

Using linked lists (3)

- Add a new node after the given list head: `list_add_tail();`
- Delete a node and reinitialize it: `list_del_init();`
- Move one node from one list to another: `list_move();`, `list_move_tail();`
- Check if a list is empty: `list_empty();`
- Join two lists: `list_splice();`
- Iterate without prefetching: `__list_for_each();`
- Iterate backward: `list_for_each_prev();`
- If your loop may delete nodes in the list: `list_for_each_safe();`

Synchronization

- Sources of concurrency:
  1. Processes using the same driver at the same time
  2. Interrupt handlers invoked at the same time that the driver is doing something else
  3. Kernel timers run asynchronously as well
  4. Kernel running on a symmetric multiprocessor (SMP)
  5. Preemptible kernel: uniprocessors behave like multiprocessors

- Kernel and drivers code must allow multiple instances to run at the same time in different contexts

Synchronization (2)

- When programming the kernel it is crucial to forbid execution flows (asynchronous functions, exception and system call handlers) to badly interfere with each other (**race conditions**).
- **Keep concurrency in mind!**
- The Linux kernel offers a large number of synchronization primitives

Synchronization (3)

- A large number of synchronization primitives are used for efficiency reasons: the kernel must reduce to a minimum the time spent waiting for a resource
- In particular, most of the mutual exclusion mechanisms have been introduced to allow some kernel core components to scale well in large Enterprise systems
- Simplifying a little bit, mutual exclusion can be enforced by using
  1. **mutexes** (used to be semaphores in old kernels)
  2. **spinlocks** (optionally coupled with interrupt disabling)

Mutexes

- Mutexes (mutex exclusion semaphores) can be used to protect shared data structures that are only accessed in process context
- Like user-space (pthread) mutexes, kernel mutexes are synchronization objects aimed at controlling the access to the resources shared among the processes in the system
- While a process is waiting on a busy mutex, it is blocked (put in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and replaced by another runnable process
- **Remember: mutexes cannot be used in interrupt context!**

Basically, a semaphore cannot be used in interrupt context!

Mutexes

- Mutexes (mutex exclusion semaphores) can be used to protect shared data structures that are only accessed in process context
- Like user-space (pthread) mutexes, kernel mutexes are synchronization objects aimed at controlling the access to the resources shared among the processes in the system
- While a process is waiting on a busy mutex, it is blocked (put in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and replaced by another runnable process
- **Remember: mutexes cannot be used in interrupt context!**

Basically, a semaphore cannot be used in interrupt context!

Using Mutexes

- First of all, declare the mutex as a shared variable seen by all the processes that need to use it:
  `struct mutex foo_mutex;`
- To acquire the shared resource protected by the mutex:
  `mutex_lock(&foo_mutex);` or
  `mutex_lock_interruptible(&foo_mutex);`
- To release the resource:
  `mutex_unlock(&foo_mutex);`

Spinlocks

- **Spinlocks** are used to protect data structures that can be possibly accessed in interrupt context
- A spinlock is a mutex implemented by an atomic variable that can have only two possible values: *locked* and *unlocked*
- When the CPU must acquire a spinlock, it reads the value of the atomic variable and sets it to *locked*. If the variable was already locked before the read-and-set operation, the whole step is repeated ("spinning").
- Therefore, a process waiting for a spinlock is never blocked!

Spinlocks

- **Spinlocks** are used to protect data structures that can be possibly accessed in interrupt context
- A spinlock is a mutex implemented by an atomic variable that can have only two possible values: *locked* and *unlocked*
- When the CPU must acquire a spinlock, it reads the value of the atomic variable and sets it to *locked*. If the variable was already locked before the read-and-set operation, the whole step is repeated ("spinning").
- Therefore, a process waiting for a spinlock is never blocked!

Spinlocks

- When using spinlocks it's easy to cause **deadlocks**.
- Some important issues to remember:
  - If the data structure protected by the spinlock is accessed also in interrupt context, we must disable the interrupts before acquiring the spinlock
  - The kernel automatically disables kernel preemption once a spinlock has been acquired

Using spinlocks

- To allocate and initialize a spinlock:
  `spinlock_t foo_lock;`
  `spin_lock_init(&foo_lock);` [unlocked]
- To disable interrupts and acquire the spinlock:
  `spin_lock_irqsave(&foo_lock, flags);` [locked]
- To release the spinlock and restore the previous interrupt status:
  `spin_lock_irqrestore(&foo_lock, flags);` [unlocked]

Time management

- Several kernel functions are time-driven
- Periodic functions:
  - Time of day and system uptime updating
  - Runqueue balancing on SMP
  - Timeslice checking

System timer

- Hardware timer issuing an interrupt at a programmable frequency called **tick rate**
- The interrupt handler is called timer interrupt
- The tick rate is defined by the static preprocessor define `HZ` (see `linux/param.h`)
- The value of HZ is architecture-dependent
- Some internal calculations assume $12 \leq$ `HZ` ≤ 1535 (see `linux/timex.h`)
- On x86 architectures the primary system timer is the *Programmable Interrupt Timer* (PIT)

Value of HZ

Larger HZ values: pros and cons

- Timer interrupt runs more frequently
- Benefits
  - Higher resolution of timed events
  - Improved accuracy of timed events
    - Average error = 5msec with HZ=100
    - Average error = 0.5msec with HZ=1000
  - Improved precision of syscalls employing a timeout
    - Examples: `poll()` and `select()`.
  - Measurements (e.g. resource usage) have finer resolution
  - Process preemption occurs more accurately
- Drawbacks
  - The processor spends more time executing the timer interrupt handler
  - Higher overhead
  - More frequent cache trashing

Measure of time

1. Relative times
   - Most important to kernel functions and device drivers
   - Example: 5 seconds from now
   - Kernel facilities: `jiffies`, clock cycles and `get_cycles()`

2. Absolute times
   - Current time of day
   - Called **"wall time"**
   - Most important to user-space applications
   - Kernel facilities: `xtime`, `mktime()` and `do_gettimeofday()`
   - Usually best left to user-space, where the C library offers better support
   - Dealing with absolute times in kernel space is often sign of bad implementation

Measure of time

1. Relative times
   - Most important to kernel functions and device drivers
   - Example: 5 seconds from now
   - Kernel facilities: `jiffies`, clock cycles and `get_cycles()`

2. Absolute times
   - Current time of day
   - Called **"wall time"**
   - Most important to user-space applications
   - Kernel facilities: `xtime`, `mktime()` and `do_gettimeofday()`
   - Usually best left to user-space, where the C library offers better support
   - Dealing with absolute times in kernel space is often sign of bad implementation

Jiffies

- Global variable `jiffies`
- Number of ticks occurred since the system booted
- Read-only
- Incremented at any timer interrupt
- Not updated when interrupts are disabled
- The system uptime is therefore `jiffies/HZ` seconds
- Declared in `linux/jiffies.h` as

  ```
  extern unsigned long volatile jiffies;
  ```
- Declared as `volatile` to tell the compiler not to optimize memory reads
- `unsigned long` (32 bits) for backward compliance

Jiffies (2)

- For high values of `HZ`, `jiffies` wraps around very quickly
- Four macros to handle wraparounds:
  - `time_after(unknown, known)`
  - `time_before(unknown, known)`
  - `time_after_eq(unknown, known)`
  - `time_before_eq(unknown, known)`
- The macros convert the values to `signed long` and perform a subtraction
- The `unknown` parameter is typically `jiffies`
- See `linux/jiffies.h`

Jiffies_64

- Extended variable `jiffies_64`
- Read-only
- Declared in `linux/jiffies.h` as
  `extern u64 jiffies_64;`
- `jiffies` is the lower 32 bits of the full 64-bit `jiffies_64` variable
- The access is not atomic on 32-bit architectures
- Can be read through the function `get_jiffies_64()`

High-resolution processor-specific timing

- Many architectures provide high-resolution counter registers
- Incremented once **at each clock cycle**
- Architecture-dependent: readable from user space, writable, 32 or 64 bits, etc.
- x86 processors (from Pentium) have TimeStamp Counter (TSC)
  - 64-bit register
  - Readable from both kernel and user spaces
  - See `asm/msr.h` (*"machine-specific registers"*)
  - Three macros:

```
rdtsc(low32, high32);
rdtscl(low32);
rdtscll(var64);
```

High-resolution architecture-independent timing

- The kernel offers an architecture-independent function
- `cycles_t get_cycles(void);`
- Defined in `asm/timex.h`
- Defined for every platform
  - Returns `0` on platforms without cycle-counter register

Absolute times: the `xtime` variable

- The `xtime` variable
- Defined in `kernel/timer.c` as `struct timespec xtime;`
- Timespec data structure:

```
struct timespec
{
    time_t  tv_sec;          /* seconds */
    long    tv_nsec;         /* nanoseconds */
};
```

- Time elapsed since January 1st 1970 (*"epoch"*)
- Jiffies granularity
- Not atomic access
- Read through

  `struct timespec current_kernel_time(void);`

Absolute times: do_gettimeofday()

- Function `do_gettimeofday()`
- Exported by `linux/time.h`
- Prototype:

  `void do_gettimeofday(struct timeval *tv);`
- Timeval data structure:

```
struct timeval {
    time_t         tv_sec;        /* seconds */
    suseconds_t    tv_usec;  /* microseconds */
};
```

- Can have resolution near to microseconds
  - Interpolation: see what fraction of the current jiffy has already elapsed
  - m68k and Sun3 systems cannot offer more than jiffy resolution

Absolute times: mktime()

- Function `mktime()`
- Turns a wall-clock time into a jiffies value
- Prototype:

```
unsigned long mktime (unsigned int year, \
                      unsigned int mon,  \
                      unsigned int day,  \
                      unsigned int hour, \
                      unsigned int min,  \
                      unsigned int sec); \
```

- See `linux/time.h`

Delaying Execution

- The wrong way: busy waiting

```
while (time_before(jiffies, j1))
cpu_relax();
```

- Works because `jiffies` is declared as `volatile`
- Crash if interrupts are disabled

Delaying Execution (2)

- Release the CPU

```
while (time_before(jiffies, j1))
schedule();
```

- Still not optimal
- There is always at least one runnable process
- The idle task never runs
- Waste of energy

Delaying Execution (3)

- The best way to implement a delay is to ask the kernel to do it!
- Facilities:
  1. `ndelay()`, `udelay()`, `mdelay()`
  2. `schedule_timeout()`
  3. Kernel timers

Small delays

- Sometimes the kernel code requires very short and rather precise delays
- Example: synchronization with hardware devices
- The kernel provides the following functions:
  - `void ndelay (unsigned long nsecs);`
  - `void udelay (unsigned long usecs);`
  - `void mdelay (unsigned long msecs);`
- Busy looping for a certain number of cycles
- Trivial usage:
  `udelay(150);` for 150 $\mu secs$.
- The delay is **at least** the requested value
- See `linux/delay.h`

Small delays (2)

- To avoid overflows, there is a check for constant parameters
  - `Unresolved symbol __bad_udelay`
- **Do not use for big amounts of time!**
- Architecture-dependent (see `asm/delay.h`)
- BogoMIPS:
  - How many loops the processor can complete in a second
  - Stored in the `loops_per_jiffy` variable
  - See `proc/cpuinfo`

Small delays without busy waiting

- Another way of achieving msec delays
- The kernel provides the following functions:
  1. `void msleep (unsigned int msecs);`
     - Uninterruptible
  2. `unsigned long msleep_interruptible (unsigned int msecs);`
     - Interruptible
     - Normally returns `0`
     - Returns the number of milliseconds remaining if the process is awakened earlier
  3. `void ssleep (unsigned int seconds);`
     - Uninterruptible
- See `linux/delay.h`

Small delays without busy waiting

- Another way of achieving msec delays
- The kernel provides the following functions:
  1. `void msleep (unsigned int msecs);`
     - Uninterruptible
  2. `unsigned long msleep_interruptible (unsigned int msecs);`
     - Interruptible
     - Normally returns `0`
     - Returns the number of milliseconds remaining if the process is awakened earlier
  3. `void ssleep (unsigned int seconds);`
     - Uninterruptible
- See `linux/delay.h`

Small delays without busy waiting

- Another way of achieving msec delays
- The kernel provides the following functions:
  1. `void msleep (unsigned int msecs);`
     - Uninterruptible
  2. `unsigned long msleep_interruptible (unsigned int msecs);`
     - Interruptible
     - Normally returns `0`
     - Returns the number of milliseconds remaining if the process is awakened earlier
  3. `void ssleep (unsigned int seconds);`
     - Uninterruptible
- See `linux/delay.h`

schedule_timeout()

- Prototype:

  ```
  signed long schedule_timeout(signed long delay);
  ```

- See `linux/sched.h`
- Returns `0` unless the function returns before the given delay has elapsed (e.g. signal)
- Usage:

  ```
  set_current_state(TASK_INTERRUPTIBLE);
  schedule_timeout (delay);
  ```

- Use `TASK_UNINTERRUPTIBLE` for uninterruptible delays

Kernel timers

- Allow to schedule an action to happen later without blocking the current process until that time arrives
- Have `HZ` resolution
- Example: shut down the floppy drive motor
- Also called **"dynamic timers"** or just *"timers"*
- Asynchronous execution: run in **interrupt context**
- Potential source of race conditions ⇒ protect data from concurrent access
- On SMPs the timer function is executed by the same CPU that registered it to achieve better cache locality

Kernel timers (2)

- Can be dynamically created and destroyed
- Not cyclic
- No limit on the number of timers
- See `linux/timer.h` and `kernel/timer.c`
- Represented by the `struct timer_list` structure

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    void (*function)(unsigned long);
    unsigned long data;
    struct tvec_t_base_s * base;
};
```

Kernel timers (2)

- The `expires` field represents when the timer will fire (expressed in jiffies)
- When the timer fires, it runs the function `function` with `data` as argument.

Using kernel timers

1. Define a timer:
   `struct timer_list my_timer;`
2. Define a function:
   `void my_timer_function(unsigned long data);`
3. Initialize the timer:
   `init_timer(&my_timer);`
4. Set an expiration time:
   `my_timer.expires = jiffies + `*`delay`*`;`
5. Set the argument of the function:
   `my_timer.data = 0;` or
   `my_timer.data = (unsigned long) &`*`param`*`;`
6. Set the handler function:
   `my_timer.function = my_function;`
7. Activate the timer:
   `add_timer(&my_timer);`

Using kernel timers

1. Define a timer:
   `struct timer_list my_timer;`
2. Define a function:
   `void my_timer_function(unsigned long data);`
3. Initialize the timer:
   `init_timer(&my_timer);`
4. Set an expiration time:
   `my_timer.expires = jiffies + *delay*;`
5. Set the argument of the function:
   `my_timer.data = 0;` or
   `my_timer.data = (unsigned long) *&param*;`
6. Set the handler function:
   `my_timer.function = my_function;`
7. Activate the timer:
   `add_timer(&my_timer);`

Using kernel timers

1. Define a timer:
   `struct timer_list my_timer;`
2. Define a function:
   `void my_timer_function(unsigned long data);`
3. Initialize the timer:
   `init_timer(&my_timer);`
4. Set an expiration time:
   `my_timer.expires = jiffies + delay;`
5. Set the argument of the function:
   `my_timer.data = 0;` or
   `my_timer.data = (unsigned long) &param;`
6. Set the handler function:
   `my_timer.function = my_function;`
7. Activate the timer:
   `add_timer(&my_timer);`

Using kernel timers

1. Define a timer:

   ```
   struct timer_list my_timer;
   ```

2. Define a function:

   ```
   void my_timer_function(unsigned long data);
   ```

3. Initialize the timer:

   ```
   init_timer(&my_timer);
   ```

4. Set an expiration time:

   ```
   my_timer.expires = jiffies + delay;
   ```

5. Set the argument of the function:

   `my_timer.data = 0;` or

   `my_timer.data = (unsigned long) &param;`

6. Set the handler function:

   ```
   my_timer.function = my_function;
   ```

7. Activate the timer:

   ```
   add_timer(&my_timer);
   ```

Using kernel timers

1. Define a timer:
   `struct timer_list my_timer;`
2. Define a function:
   `void my_timer_function(unsigned long data);`
3. Initialize the timer:
   `init_timer(&my_timer);`
4. Set an expiration time:
   `my_timer.expires = jiffies + delay;`
5. Set the argument of the function:
   `my_timer.data = 0;` or
   `my_timer.data = (unsigned long) &param;`
6. Set the handler function:
   `my_timer.function = my_function;`
7. Activate the timer:
   `add_timer(&my_timer);`

Using kernel timers

1. Define a timer:
   `struct timer_list my_timer;`
2. Define a function:
   `void my_timer_function(unsigned long data);`
3. Initialize the timer:
   `init_timer(&my_timer);`
4. Set an expiration time:
   `my_timer.expires = jiffies + *delay*;`
5. Set the argument of the function:
   `my_timer.data = 0;` or
   `my_timer.data = (unsigned long) &*param*;`
6. Set the handler function:
   `my_timer.function = my_function;`
7. Activate the timer:
   `add_timer(&my_timer);`

Using kernel timers

1. Define a timer:

   `struct timer_list my_timer;`

2. Define a function:

   `void my_timer_function(unsigned long data);`

3. Initialize the timer:

   `init_timer(&my_timer);`

4. Set an expiration time:

   `my_timer.expires = jiffies + `*`delay`*`;`

5. Set the argument of the function:

   `my_timer.data = 0;` or

   `my_timer.data = (unsigned long) &`*`param`*`;`

6. Set the handler function:

   `my_timer.function = my_function;`

7. Activate the timer:

   `add_timer(&my_timer);`

Using kernel timers (2)

8. Modify the timer:
   `mod_timer (&my_timer, jiffies + new_delay);`
9. Deactivate the timer:
   `del_timer (&my_timer);`
10. Deactivate the timer avoiding race conditions on SMPs :
    `del_timer_sync (&my_timer);`
11. Knowing timer's state:
    `timer_pending (&my_timer);`

Using kernel timers (2)

8. Modify the timer:
   `mod_timer (&my_timer, jiffies + new_delay);`
9. Deactivate the timer:
   `del_timer (&my_timer);`
10. Deactivate the timer avoiding race conditions on SMPs :
    `del_timer_sync (&my_timer);`
11. Knowing timer's state:
    `timer_pending (&my_timer);`

Using kernel timers (2)

Development

Kernel modules

Kernel Lists

Synchronization

Timing

8. Modify the timer:
   `mod_timer (&my_timer, jiffies + new_delay);`
9. Deactivate the timer:
   `del_timer (&my_timer);`
10. Deactivate the timer avoiding race conditions on SMPs :
    `del_timer_sync (&my_timer);`
11. Knowing timer's state:
    `timer_pending (&my_timer);`

Using kernel timers (2)

8. Modify the timer:
   `mod_timer (&my_timer, jiffies + new_delay);`
9. Deactivate the timer:
   `del_timer (&my_timer);`
10. Deactivate the timer avoiding race conditions on SMPs :
    `del_timer_sync (&my_timer);`
11. Knowing timer's state:
    `timer_pending (&my_timer);`

Implementation of kernel timers