Time, Delays, and Deferred Work

Reading

- Please read Chapter 7 of the LDD book

Topics

- Measuring time lapses and comparing times
- Knowing the current time
- Delaying operation for a specified amount of time
- Scheduling asynchronous functions to happen at a later time

Measuring Time Lapses

- Kernel keeps track of time via timer interrupts
  - Generated by the timing hardware
  - Programmed at boot time according to **`HZ`**
    - Architecture-dependent value defined in **`<linux/param.h>`**
    - Usually 100 to 1,000 interrupts per second
- Every time a timer interrupt occurs, a kernel counter called **`jiffies`** is incremented
  - Initialized to 0 at system boot

Using the `jiffies` Counter

- To access the 64-bit counter **`jiffie_64`** on 32-bit machines, call

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

Using the **`jiffies`** Counter

- Must treat **`jiffies`** as read-only
- Example

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

unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n*HZ/1000; /* n milliseconds */
```

Using the `jiffies` Counter

- Jiffies may wrap - use these macro functions

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

/* check if a is after b */
int time_after(unsigned long a, unsigned long b);

/* check if a is before b */
int time_before(unsigned long a, unsigned long b);

/* check if a is after or equal to b */
int time_after_eq(unsigned long a, unsigned long b);

/* check if a is before or equal to b */
int time_before_eq(unsigned long a, unsigned long b);
```

Using the `jiffies` Counter

- 32-bit counter wraps around every 50 days
- To exchange time representations, call

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

unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies,
                         struct timespec *value);
```

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
};
```

```
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies,
                        struct timeval *value
```

```
struct timeval {
  time_t tv_sec;
  susecond_t tv_usec;
};
```

Knowing the Current Time

- `jiffies` represents only the time since the last boot
- To obtain wall-clock time, use

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

/* near microsecond resolution */
void do_gettimeofday(struct timeval *tv);

/* based on xtime, near jiffy resolution */
struct timespec current_kernel_time(void);
```

Processor-Specific Registers

- To obtain high-resolution timing
  - Need to access the CPU cycle counter register
    - Incremented once per clock cycle
    - Platform-dependent
      - Register may not exist
      - May not be readable from user space
      - May not be writable
        - Resetting this counter discouraged
        - Other users/CPUs might rely on it for synchronizations
      - May be 64-bit or 32-bit wide
        - Need to worry about overflows for 32-bit counters

Processor-Specific Registers

- Timestamp counter (TSC)
  - Introduced with the Pentium
  - 64-bit register that counts CPU clock cycles
  - Readable from both kernel space and user space

Processor-Specific Registers

- To access the counter, include **`<asm/msr.h>`** and use the following marcos

```
/* read into two 32-bit variables */
rdtsc(low32,high32);

/* read low half into a 32-bit variable */
rdtscl(low32);

/* read into a 64-bit long long variable */
rdtscll(var64);
```

- 1-GHz CPU overflows the low half of the counter every 4.2 seconds

Processor-Specific Registers

- To measure the execution of the instruction itself

```
unsigned long ini, end;
rdtscll(ini); rdtscll(end);
printk("time lapse: %li\n", end - ini);
```

Processor-Specific Registers

- Linux offers an architecture-independent function to access the cycle counter

```
#include <linux/tsc.h>
cycles_t get_cycles(void);
```

  - Returns 0 on platforms that have no cycle-counter register

Processor-Specific Registers

- More about timestamp counters
  - Not necessary synchronized across multiprocessor machines
  - Need to disable preemption for code that queries the counter

Delaying Execution

- From silliest way to most useful…

- Long (multi-jiffy) delays
  - Busy waiting
  - Yielding the processor
  - Timeouts
- Short delays

Busy Waiting

- Easiest way to delay execution (not recommended)

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

  - `j1` is the `jiffies` value at the expiration of the delay
  - `cpu_relax()` is an architecture-specific way of saying that you're not doing much with the CPU

Busy Waiting

- Severely degrades system performance
  - If the kernel does not allow preemption
    - Loop locks the processor for the duration of the delay
    - Scheduler never preempts kernel processes
    - Computer looks dead until time **`j1`** is reached
  - If the interrupts are disabled when a process enters this loop
    - **`jiffies`** will not be updated
    - Even for a preemptive kernel

Busy Waiting

- Behavior of a simple busy-waiting program

```
loop {
  /* print begin jiffie */
  /* busy wait for one second */
  /* print end jiffie */
}
```

  - Nonpreemptive kernel, no background load
    - Begin: 1686518, end: 1687518
    - Begin: 1687519, end: 1688519
    - Begin: 1688520, end: 1689520

Busy Waiting

- Nonpreemptive kernel, heavy background load
  - Begin: 1911226, end: 1912226
  - Begin: 1913323, end: 1914323
- Preemptive kernel, heavy background load
  - Begin: 14940680, end: 14942777
  - Begin: 14942778, end: 14945430
  - The process has been interrupted during its delay

Yielding the Processor

- Explicitly releases the CPU when not using it

```
while (time_before(jiffie, j1))
    { schedule();
}
```

  - Behavior similar to busy waiting under a preemptive kernel
    - Still consumes CPU cycles and battery power
    - No guarantee that the process will get the CPU back soon

Timeouts

- Ask the kernel to do it for you

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

long wait_event_timeout(wait_queue_head_t q, condition,
                        long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q,
                                      condition, long timeout);
```

  - Bounded sleep
  - **`timeout`**: in number of **`jiffies`** to wait, signed
  - If the timeout expires, return 0
  - If the call is interrupted, return the remaining jiffies

Timeouts

- Example:

```
DECLARE_WAIT_QUEUE_HEAD(name);    (Static Initialization)
        (or)
wait_queue_head_t wait;

init_waitqueue_head(&wait);       (Dynamic Initialization)
wait_event_interruptible_timeout(wait, 0, delay);
```

  - **`condition`** = 0 (no condition to wait for)
  - Execution resumes when
    - Someone calls **`wake_up()`**
    - Timeout expires

Timeouts

- Behavior is nearly optimal for both preemptive and nonpreemptive kernels
  - Begin: 2027024, end: 2028024
  - Begin: 2028025, end: 2029025
  - Begin: 2029026, end: 2930026

Timeouts

- Another way to schedule timeout waiting for an event

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

signed long schedule_timeout(signed long timeout);
```

  - **`timeout`**: the number of **`jiffies`** to delay
  - Require the caller set the current process state

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

  - A process may not resume immediately after the timer expires

Other Alternatives

- Non-busy-wait alternatives for millisecond or longer delays

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

void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

- **`msleep`** and **`ssleep`** are not interruptible
- **`msleeps_interruptible`** returns the remaining milliseconds

Short Delays

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

void ndelay(unsigned long nsecs); /* nanoseconds */
void udelay(unsigned long usecs); /* microseconds */
void mdelay(unsigned long msecs); /* milliseconds */
```

- Perform busy waiting

Kernel Timers

- A *kernel timer* schedules a function to run at a specified time, without blocking the current process
  - E.g., polling a device at regular intervals

Kernel Timers

- The scheduled function is run as a software interrupt
  - Needs to observe constraints imposed on this *interrupt/atomic context*
    - Not associated with any user-level process
      - No access to user space
      - The **`current`** pointer is not meaningful
    - No sleeping or scheduling may be performed
      - No calls to **`schedule()`**, **`wait_event()`**, **`kmalloc(…, GFP_KERNEL)`**, or semaphores

Kernel Timers

- To check if a piece of code is running in special contexts, call
  - `int in_interrupt();`
    - Returns nonzero if the CPU is running in either a hardware or software interrupt context
  - `int in_atomic();`
    - Returns nonzero if the CPU is running in an atomic context
      - Scheduling is not allowed
      - Access to user space is forbidden (can cause scheduling to happen)
  - Both defined in <asm/hardirq.h>

Kernel Timers

- More on kernel timers
  - A task can reregister itself (e.g., polling)
  - Reregistered timer tries to run on the same CPU
  - A potential source of race conditions, even on uniprocessor systems
    - Need to protect data structures accessed by the timer function (via atomic types or spinlocks)

The Timer API

- Basic building blocks

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

struct timer_list {
  /* ... */
  unsigned long expires;
  void (*function) (unsigned long);
  unsigned long data;
};

void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
void add_timer(struct timer_list *timer);
int del_timer(struct timer_list *timer);
```

`jiffies` value when the timer is expected to run

Called with data as argument; pointer cast to **`unsigned long`**

The Timer API

- Example

```
struct my_data {
  wait_queue_head_t wait;
  struct timer_list timer;
  unsigned long prevjiffies;
  unsigned char *buf;
  int loops;
};

int tdelay = 10; /* jiffies */
struct my_data *data;
unsigned long j = jiffies;

data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data) { return -ENOMEM; }
```

The Timer API

```
/* fill the data for our timer function */
data->prevjiffies = j;
data->buf = /* first line in the buffer */
data->loops = NUM_ASYNC_LOOPS; /* 5 */

init_timer(&data->timer);
data->timer.data = (unsigned long) data;
data->timer.function = my_timer_fn;
data->timer.expires = j + tdelay; /* parameter */
add_timer(&data->timer); /* register the timer */

/* wait for the buffer to fill */
init_waitqueue_head(&data->wait);
wait_event_interruptible(data->wait, !data->loops);
/* print buf */
```

The Timer API

```
void my_timer_fn(unsigned long arg) {
  struct my_data *data = (struct my_data *) arg;
  unsigned long j = jiffies;

  data->buf
    += sprintf(data->buf, "%9li %3li %i %6i %i %s\n", j,
                j - data->prevjiffies, in_interrupt() ? 1 : 0,
                current->pid, smp_processor_id(), current->comm);
  if (--data->loops) {
    data->timer.expires += tdelay;
    data->prevjiffies = j;
    add_timer(&data->timer);
  } else {
    wake_up_interruptible(&data->wait);
  }
}
```

The Timer API

- The output lines represent the environment where the kernel func is running.

Time	delta	inirq	pid	cpu	command
33565847	10	1	1271	0	sh
33565857	10	1	1273	0	cpp0
33565867	10	1	1273	0	cpp0
33565877	10	1	1274	0	cc1
33565887	10	1	1274	0	cc1

The Timer API

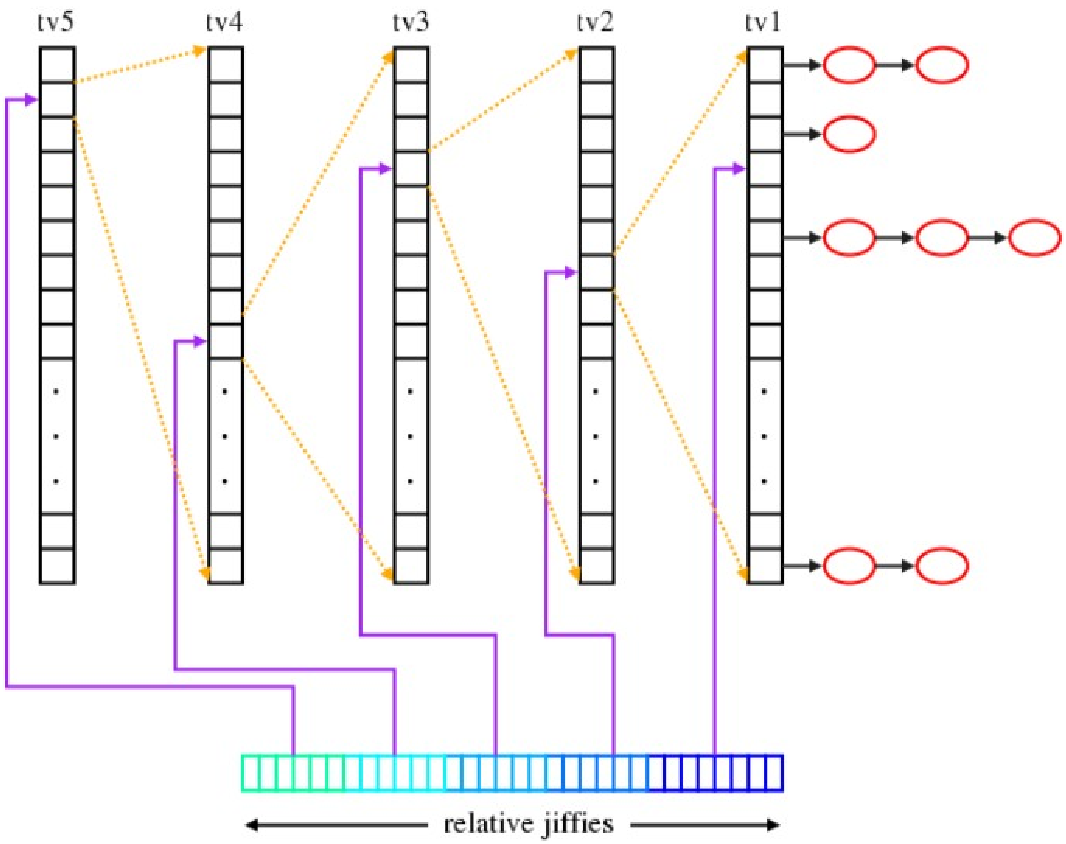
- Other functions

```
/* update the expiration time of a timer */
int mod_timer(struct timer_list *timer, unsigned long expires);

/* like del_timer, but SMP safe */
int del_timer_sync(struct timer_list *timer);

/* returns true if the timer is scheduled to run */
int timer_pending(const struct timer_list * timer);
```

The Implementation of Kernel Timers

- Requirements
  - Lightweight
  - Scale as the number of timers increases
  - Most timers expire within a few seconds
  - Run on the same registered CPU
- Solution
  - Per-CPU data structure

Timer Implementation



(from lwn.net)

Group	Ticks	Time
`tv1`	< 2^8	< .256 secs
`tv2`	< 2^14	< 16.4 secs
`tv3`	< 2^20	< 17.5 mins
`tv4`	< 2^26	< 18 hrs
`tv5`	< Inf	< Inf

Tasklets

- Resemble kernel timers
  - Always run at interrupt time
  - On the same CPU that schedules them
  - Receive an **`unsigned long`** argument
  - Can reregister itself
- Unlike kernel timers
  - Only can ask a tasklet to be run later (not at a specific time)

Tasklets

- Useful with hardware interrupt handling
  - Must be handled as quickly as possible
  - A tasklet is handled later in a soft interrupt
- Can be enabled/disabled (nested semantics)
- Can run at normal or high priority
- May run immediately, but no later than the next timer tick
- Cannot be run concurrently with itself

Tasklets

- Basic building blocks

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

struct tasklet_struct {
  /* ... */
  void (*func)(unsigned long);
  unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
                  void (*func) (unsigned long),
                               unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Tasklets

- Example

```
struct my_data {
  wait_queue_head_t wait;
  struct tasklet_struct tlet;
  int hi; /* 0 for normal priority */
  unsigned long prevjiffies;
  unsigned char *buf;
  int loops;
};

int tdelay = 10; /* jiffies */
struct my_data *data;
unsigned long j = jiffies;

data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data) { return -ENOMEM; }
```

Tasklets

```
/* fill the data for our timer function */
data->prevjiffies = j;
data->buf = /* first line in the buffer */
data->loops = NUM_ASYNC_LOOPS; /* 5 */

tasklet_init(&data->tlet, my_tasklet_fn, (unsigned long) data);
data->hi = /* arg from proc_read() */
if (data->hi)
  tasklet_hi_schedule(&data->tlet);
else
  tasklet_schedule(&data->tlet);

/* wait for the buffer to fill */
init_waitqueue_head(&wait);
wait_event_interruptible(data->wait, !data->loops);
/* print buf */
```

Tasklets

```
void my_tasklet_fn(unsigned long arg) {
  struct my_data *data = (struct my_data *) arg;
  unsigned long j = jiffies;
  data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n", j,
                 j - data->prevjiffies, in_interrupt() ? 1 : 0,
                 current->pid, smp_processor_id(), current->comm);
  if (--data->loops) {
    data->prevjiffies = j;
    if (data->hi)
      tasklet_hi_schedule(&data->tlet);
    else
      tasklet_schedule(&data->tlet);
  } else {
    wake_up_interruptible(&data->wait);
  }
}
```

Tasklets

- The kernel provides a set of ksoftirqd kernel threads, one per CPU, just to run "soft interrupt" handlers, such as the tasklet_action

Time	delta	inirq	pid	cpu	command
6076140	1	1	4368	0	cc1
6076141	1	1	4368	0	cc1
6076141	0	1	2	0	ksoftirqd/0
6076141	0	1	2	0	ksoftirqd/0
6076141	0	1	2	0	ksoftirqd/0

Tasklet Interface

```
/* make a tasklet stop running immediately; will not execute
   until it is enabled again */
void tasklet_disable(struct tasklet_struct *t);

/* disable the tasklet when it returns */
void tasklet_disable_nosync(struct tasklet_struct *t);

/* need the same number of enable calls as disable calls */
void tasklet_enable(struct tasklet_struct *t);

/* Ignore if the tasklet is already scheduled */
/* If a tasklet is already running, run the tasklet again after
   it completes */
void tasklet_schedule(struct tasklet_struct *t);
```

Tasklet Interface

```
/* schedule the tasklet with higher priority */
void tasklet_hi_schedule(struct tasklet_struct *t);

/* ensures that the tasklet is not scheduled to run again */
/* will finish scheduled tasklet */
void tasklet_kill(struct tasklet_struct *t);
```

Workqueues (may replace tasklets)

- Similar to tasklets
  - Kernel can request a function to be called later
  - Cannot access the user space
- Unlike tasklets
  - Queued task may run on a different CPU
  - Workqueue functions are associated with a special kernel processes
    - Can sleep
  - Can be delayed for an explicit interval

Workqueues

- Requires **`struct workqueue_struct`**
  - Defined in **`<linux/workqueue.h>`**
- To create a workqueue, call

```
/* create one workqueue thread per processor */
struct workqueue_struct *create_workqueue(const char *name);

/* create a single workqueue thread */
struct workqueue_struct *
  create_singlethread_workqueue(const char *name);
```

Workqueues

- To submit a task to a workqueue, you need to fill in a **`work_struct`** structure
  - At compile time, call

    ```
    DECLARE_WORK(name, void (*function)(void *), void *data);
    ```
  - At runtime, call one of the following

    ```
    /* does a more thorough job of initializing the structure */
    INIT_WORK(struct work_struct *work,
              void (*function) (void *), void *data);

    /* does not link the work_struct into the workqueue */
    PREPARE_WORK(struct work_struct *work,
                 void (*function) (void *), void *data);
    ```

Workqueues

- To submit work to a workqueue, call either

```
int queue_work(struct workqueue_struct *queue,
               struct work_struct *work);

/* may specify the delay in jiffies */
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work,
                       unsigned long delay);
```

- To cancel a pending workqueue entry, call

```
/* returns nonzero if the entry is still pending */
int cancel_delayed_work(struct work_struct *work);
```

Workqueues

- To flush a workqueue, call

```
void flush_workqueue(struct workqueue_struct *queue);
```

- To destroy a workqueue, call

```
void destroy_workqueue(struct workqueue_struct *queue);
```

The Shared Queue

- Example
  - To initialize the workqueue

```
static struct work_struct my_work;
/* in the module init function */
INIT_WORK(&my_work, my_print_wq, &my_data);
```

  - Submit Work

```
schedule_work(&my_work);
```

The Shared Queue

- The work function resubmits itself

```
static void my_print_wq(void *ptr) {
  struct clientdata *data = (struct clientdata *) ptr;

  if (!my_print(ptr)) /* print if space permits */
    return;

  if (data->delay) {
    /* instead of queue_delayed_work() */
    schedule_delayed_work(&my_work, data->delay);
  } else {
    /* instead of queue_work() */
    schedule_work(&my_work);
  }
}
```

The Shared Queue

- To cancel a work entry submitted to a shared queue, call

```
/* returns nonzero if the entry is still pending */
int cancel_delayed_work(struct work_struct *work);
```

- To flush a shared workqueue, call

```
/* instead of flush_workqueue */
void flush_scheduled_work(void);
```

Various Delayed Execution Methods

	Interruptible during the wait	No busy waiting	Good precision for Fine-grained delay	Scheduled task can access user space	Can sleep inside the scheduled task
Busy waiting	Maybe	No	No	Yes	Yes
Yielding the processor	Yes	Maybe	No	Yes	Yes
Timeouts	Maybe	Yes	Yes	Yes	Yes
msleep, ssleep	No	Yes	No	Yes	Yes
msleep_interruptible	Yes	Yes	No	Yes	Yes
ndelay, udelay, mdelay	No	No	Maybe	Yes	Yes
Kernel timers	Yes	Yes	Yes	No	No
Tasklets	Yes	Yes	No	No	No
Workqueues	Yes	Yes	Yes	No	Yes