Taller de Sistemas Operativos

Administración del tiempo

Agenda

- Conceptos generales
- Relojes de Hardware y Timers
- Timer Interrupt Handler
- El tiempo real
- Timers
- Ejecución demorada
- Busy looping
- Demoras pequeñas
- Schedule timeout

Conceptos generales

- El paso del tiempo es muy importante para el kernel
- Un gran número de funciones son “time driven”
  - Balanceo de las runqueues, refresco de pantalla
- El kernel puede agendar trabajo a futuro
- El kernel debe manejar el “system uptime” y la fecha y hora actual
- Debe diferenciarse entre tiempo absoluto y relativo
- Debe diferenciarse entre eventos periódicos y eventos que se agendan para un momento en el futuro
- También existen los dynamic timers, que se usan para agendar eventos una única vez después que un lapso haya transcurrido.

Noción de tiempo del kernel

- El kernel debe interactuar con el hardware para comprender y administrar el tiempo
- El system timer envía una señal a una frecuencia preprogramada (tick rate). En este momento genera una interrupción que es manejada por un manejador especial
- Como se conoce el tick rate, se conoce el tiempo entre dos interrupciones de tiempo consecutivas
- De esta manera el kernel maneja el *uptime* y el *wall time*

Noción de tiempo del kernel

- La interrupción de tiempo es muy importante para el manejo del SO. El trabajo períodico incluye
  - Actualizar el uptime del sistema
  - Actualizar la fecha y la hora
  - En sistemas SMP balancear las runqueues
  - Verificar si un proceso terminó su time slice
  - Ejecutar los dynamic timers que expiraron
  - Actualizar las estadísticas de uso de CPU y recursos
- Alguno de estos trabajos ocurren en cada interrupción
- Otras funciones ocurren cada n interrupciones

Tick Rate : HZ

- La frecuencia del system timer es configurada en el boot del sistema según el valor estático definido en `HZ`.
- Este valor es dependiente de la arquitectura `<asm/param.h>`
- El Tick Rate tiene una frecuencia de `HZ` hertz y un período de `1/HZ` segundos.
- Cuando desarrolle código del kernel nunca asuma que HZ tiene un valor dado

El valor ideal de HZ

- Durante el desarrollo del kernel 2.5 la frecuencia se aumentó a 1000 Hz
- Aumentando el tick rate significa que la interrupción del timer ejecuta más frecuentemente. Estos son los beneficios:
  - La interrupción del timer tiene una resolución más alta y por lo tanto todos los eventos de tiempo tienen una mayor resolución
  - System calls como `poll()` y `select()` que emplean un valor de timeout ejecutan con mayor precisión
  - Medidas de usos de recursos o system uptime son registrados con una resolución más fina
  - Preemption ocurre más correctamente

El valor ideal de HZ

- Las desventajas de un mayor HZ son:
  - Las interrupciones de timer son más frecuentes por lo que existe un overhead mayor
  - El procesador pasa más tiempo ejecutando la interrupción del timer
  - El thrashing del processor cache es más frecuente

Jiffies

- La variable global `jiffies` almacena el número de ticks que sucedieron desde que se inició el sistema
- En el proceso de boot el kernel inicializa `jiffies` a cero
- Es incrementado en 1 en cada interrupción del timer
- Hay `HZ` interrupciones de timer en un segundo, hay `HZ jiffies` en un segundo
- El *uptime* del sistema es `jiffies/HZ` segundos
- Se declara en `<linux/jiffies.h>` como

```
extern unsigned long volatile jiffies;
```

- Para convertir de segundos a `jiffies`

```
(segundos * HZ)
```

Jiffies

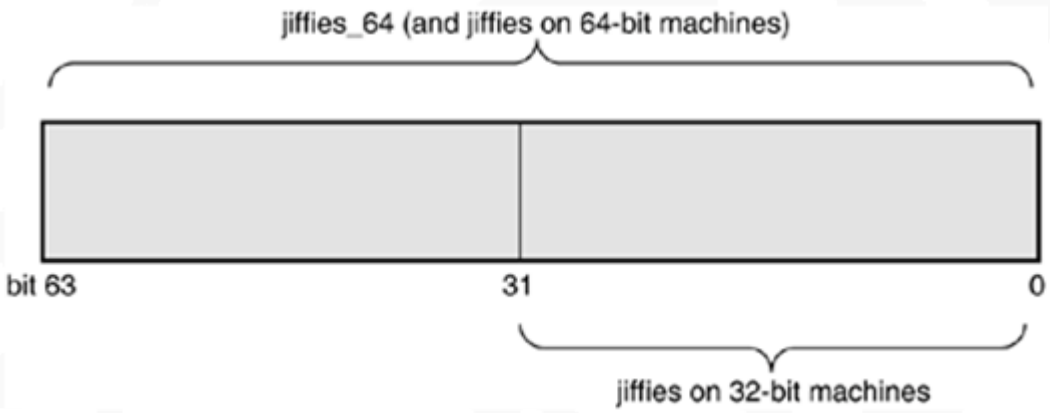
- Para convertir de `jiffies` a segundos `(jiffies/HZ)`
- `jiffies` es de 32 bits en arquitecturas de 32 bits y de 64 bits en arquitecturas de 64 bits
- Con `HZ=1000`, `jiffies` se desborda en 49,7 días en arquitecturas de 32 bits
- Si siempre fuera de 64 bits, no se desbordaría en la vida de nadie
- Se define una segunda variable en `<linux/jiffies.h>`

```
extern u64 jiffies_64
```

- El script que linkedita el kernel solapa la variable `jiffies` en el comienzo de `jiffies_64`

Jiffies

- Como se usa generalmente para medir lapsos, la mayoría del código toma en cuenta solamente los 32 bits más bajos
- Las rutinas de manejo de tiempo usan los 64 bits
- La función `get_jiffies_64 ()` se usa para acceder al valor de 64 bits
- En arquitecturas de 64 bits `jiffies` y `jiffies_64` hacen referencia a lo mismo



Jiffies

- Para manejar correctamente el desborde de `jiffies` existen las siguientes funciones
  - `#define time_after(unknown, known) ((long)(known) - (long)(unknown) < 0)`
  - `#define time_before(unknown, known) ((long)(unknown) - (long)(known) < 0)`
  - `#define time_after_eq(unknown, known) ((long)(unknown) - (long)(known) >= 0)`
  - `#define time_before_eq(unknown, known) ((long)(known) - (long)(unknown) >= 0)`
- Ejemplo

```
unsigned long timeout = jiffies + HZ/2;
/* timeout in 0.5s */
/* do some work ... */
if (time_before(jiffies, timeout))          /* timeout > jiffies */
{ /* we did not time out, good ... */ }
else
{ /* we timed out, error ... */ }
```

Relojes de HW y Timers

- Las arquitecturas proveen dos dispositivos para ayudar en el manejo del tiempo
  - Real Time Clock (RTC)
  - System Timer
- El RTC provee un dispositivo con memoria no volátil para almacenar la hora del sistema.
- Durante el boot el kernel lee del RTC y lo utiliza para inicializar la hora del sistema cargando la variable `xtime`
- El System Timer tiene un rol más importante en el manejo del tiempo. La idea es proveer un mecanismo para generar una interrupción de manera periódica.

El handler de la int. del Timer

- La interrupción del timer está dividida en dos partes, una dependiente de la arquitectura y una independiente
- La parte dependiente de la arquitectura se registra como handler de la interrupción del timer y hace lo siguiente:
  - Obtiene el `xtime_lock` que protege a `xtime` y `jiffies_64`
  - Maneja o resetea el system timer
  - Periódicamente guarda la hora del sistema en el RTC
  - Llama a la rutina independiente de la arquitectura `do_timer()`

El handler de la int. del Timer

- La rutina independiente de la arquitectura hace lo siguiente:
  - Incrementa `jiffies_64` en 1 (ya obtuvimos el `xtime_lock`)
  - Actualiza el uso de recursos para el proceso ejecutando
  - Ejecuta los dynamic timers expirados
  - Ejecuta `scheduler_tick()`
  - Actualiza la hora del sistema almacenada en `xtime`
  - Calcula el load average

El handler de la int. del Timer

```
void do_timer(struct pt_regs *regs)
{       jiffies_64++;
        update_process_times(user_mode(regs));
        update_times(); }
```

```
void update_process_times(int user_tick)
{       struct task_struct *p = current;
        int cpu = smp_processor_id();
        int system = user_tick ^ 1;
        update_one_process(p, user_tick, system, cpu);
        run_local_timers();
        scheduler_tick(user_tick, system); }
```

```
/*
 * update by one jiffy the appropriate time counter
 */
p->utime += user;
p->stime += system;
```

`run_local_timers()` marca una soft_irq para manejar la ejecución de los timers expirados

`scheduler_tick ()` decrementa el timeslice del proceso ejecutando y setea `need_resched` si es necesario

`do_timer ()` retorna al handler dep de la arquit que libera el `xtime_lock` y finalmente termina

Esto ocurre 1000 veces por segundo en un PC

```
void update_times(void)
{ unsigned long ticks;

  ticks = jiffies - wall_jiffies;
  if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
  }
  last_time_offset = 0;
  calc_load(ticks); }
```

Hora y fecha del sistema

- La estructura timespec se define en `<linux/time.h>`

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

- `xtime.tv_sec` segundos desde 01/01/1970
- `xtime.tv_nsec` nanosegundos desde el ultimo segundo
- Para actualizar `xtime` se requiere un `xtime_lock` (`write_seqlock`)
- Para leer `xtime` se requiere utilizar las rutinas `read_seqbegin()` y `read_seqretry()`
- Para recuperar la hora del sistema existe la función `gettimeofday()` implementada como `sys_gettimeofday()`

Timers

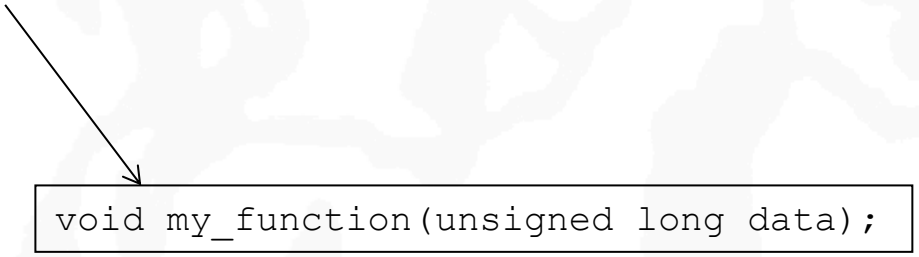
- Son esenciales para administrar el flujo de tiempo en el código del kernel. Se utilizan para demorar la ejecución de código para “después”
- Son fáciles de usar
  - Se realiza un setup inicial, especificando su expiración
  - Se especifica una función a ejecutar en expiración
  - Se activan
- No son cíclicos. Una vez expirados se destruyen
- Son representados por una `struct timer_list` definida en `<linux/timer.h>`

```
struct timer_list {
        struct list_head entry;           /* entry in linked list of timers */
        unsigned long expires;            /* expiration value, in jiffies */
        spinlock_t lock;                  /* lock protecting this timer */
        void (*function)(unsigned long);  /* the timer handler function */
        unsigned long data;               /* lone argument to the handler */
        struct tvec_t_base_s *base;       /* internal timer field, do not touch */
};
```

- El kernel provee una familia de funciones para manejar los timers
- Se encuentran las declaraciones en `<linux/timer.h>`
- La implementación se encuentra en `kernel/timer.c`

Timers

- Definir un timer
  - `struct timer_list my_timer`
- Inicializar valores internos
  - `init_timer (&my_timer)`
- Rellenar los restantes campos según requerimientos
  - `my_timer.expires = jiffies + delay; /*timer expires in delay ticks*/`
  - `my_timer.data = 0; /*zero is passed to the timer handler*/`
  - `my_timer.function = my_function; /*function to run when timer expires*/`
- Activar el timer
  - `add_timer (&my_timer)`



Timers

- Modificar un timer
  - `mod_timer(&my_timer, jiffies + new_delay); /*new expiration*/`
- Desactivar un timer antes que expire
  - del_time (&my_timer)
  - Puede haber una potencial “race condition”
  - La función nos asegura que cuando retorna el timer no está más activo
  - En equipos SMP el handler puede estar ejecutando en otro CPU
  - Para desactivar y esperar que termine un potencial handler en ejecución hay que utilizar `del_timer_sync ()`

Timers

- El kernel ejecuta los timers en un contexto de bottom half como softirqs. El handler de la interrupción de tiempo ejecuta `update_process_times()` quien a su vez ejecuta `run_local_timers()`.

```
void run_local_timers(void)
{  raise_softirq(TIMER_SOFTIRQ); }
```

- La softirq `TIMER_SOFTIRQ` es manejada por `run_timer_softirq()`. Esta función ejecuta todos los timers expirados en el procesador actual.

Ejecución demorada

- A veces el kernel debe demorar la ejecución por un tiempo sin utilizar timers ni mecanismos bottom-half
- Esto es usado tipicamente para permitir al hw completar ciertas tareas (en el orden de los microsegundos)
- Algunas soluciones saturan el procesador (BW). Otras no saturan el procesador pero no ofrecen garantías que continuará la ejecución en exactamente el tiempo requerido.

Busy Looping

- La idea es simple, permanecer en un loop hasta que pase el número deseado de ticks

```
unsigned long delay = jiffies + 10; /* ten ticks */
while (time_before(jiffies, delay))
;
```

- Una solución un mejor es utilizar el scheduler para permitir que otro proceso realice algún trabajo mientras esperamos

```
unsigned long delay = jiffies + 5*HZ;

while (time_before(jiffies, delay)) cond_resched();
```

- La llamada a `cond_resched()` agenda un nuevo proceso solamente si `need_resched` se encuentra activa

Demoras pequeñas

- A veces se necesitan demoras más chicas que un tick y a su vez precisas.
- Es imposible usar `jiffies` en estas demoras
- El kernel provee funciones para demoras de microsegundos y milisegundos que no usan `jiffies`
- Se definen en `<linux/delay.h>`

```
void udelay(unsigned long usecs)
void mdelay(unsigned long msecs)
```

- La función `udelay` se implementa en base a saber cuantas iteraciones pueden ser ejecutadas en un período de tiempo
- La función `mdelay` se implementa a partir de `udelay`

Schedule timeout

- La llamada a esta función pone un task a dormir por lo menos hasta que el tiempo especificado haya pasado
- Cuando el tiempo especificado transcurrió, el kernel despierta la tarea y la vuelve a poner en la runqueue

```
/* set task's state to interruptible sleep */
set_current_state(TASK_INTERRUPTIBLE);
/* take a nap and wake up in "s" seconds */
schedule_timeout(s * HZ);
```

- El código que la llama debe estar en process context y no debe tener nigún lock
- Es una aplicación de timers

Schedule timeout

```
signed long schedule_timeout(signed long timeout)
{
   timer_t timer;
  unsigned long expire;
  expire = timeout + jiffies;
  init_timer(&timer);
  timer.expires = expire;
  timer.data = (unsigned long) current;
  timer.function = process_timeout;
  add_timer(&timer);
  schedule();
  del_timer_sync(&timer);
  timeout = expire - jiffies;

out:
  return timeout < 0 ? 0 : timeout; }
```

Schedule timeout

- Como la tarea está marcada como `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE` el scheduler no ejecuta la tarea sino que escoge una nueva
- Cuando el timer expira ejecuta `process_timeout()`

```
void process_timeout(unsigned long data) {
    wake_up_process((task_t *) data);
}
```

- En el caso que la tarea se despierte prematuramente por una señal, se destruye el timer y la función retorna el tiempo dormido