Bottom Halves

Linux Code Patterns(2)

Bottom Halves

Bottom halves are Linux's way of delaying processing. The most advantageous place for this is in an interrupt handler. In an interrupt handler, it will process no other interrupts or events of any kind, and you would like to do what you absolutely have to and delay the rest of the manipulation until later but before returning to user space. This can be done easily in Linux by creating a bottom half. The bottom halves do a great deal of work in Linux, including much of the networking receive code, all the timer handling, and much of the terminal driver.

Also Known As

- Software Interrupt
- softirq

Context

Split the amount of work to be performed inside an interrupt handler into two parts: immediate work and the work that can be postponed until later.

Problem

When the CPU receives an interrupt, it stops whatever it is doing, saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state.

One of the main problems with interrupt handling is how to perform longish tasks within an interrupt handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other.

Solution

The solution to this problem is for the interrupt handler to do what needs to be done immediately, usually read something from the hardware or send something to the hardware, and then schedule the handling of the new information at a later time (this is called the ‘bottom half’) and return. The kernel is then guaranteed to call the bottom half as soon as possible -- and when it does, everything allowed in kernel modules will be allowed.

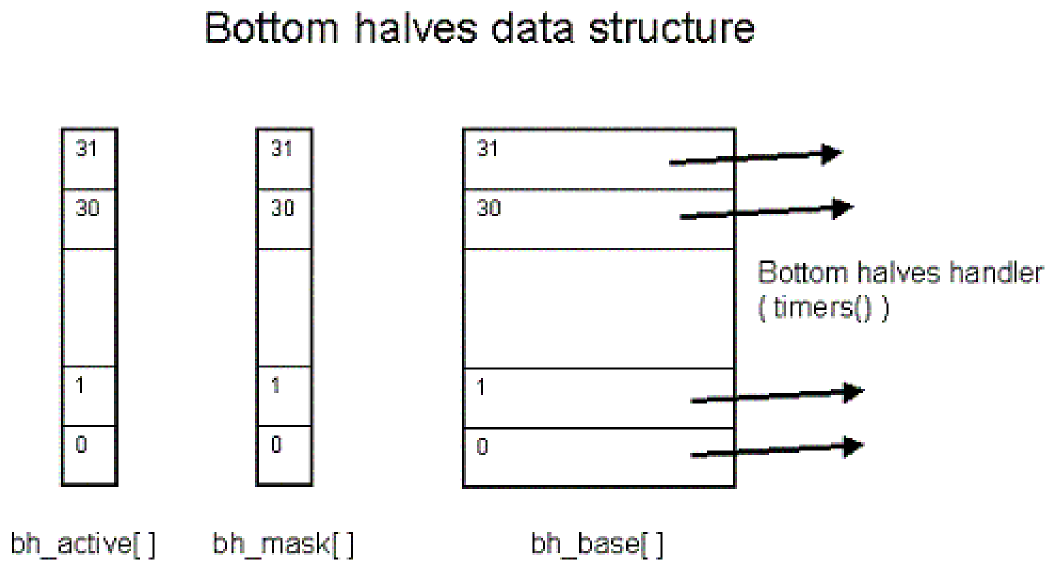
Thus interrupt handlers are divided into two parts: a top and a bottom half. The top half is the real interrupt handler: often it just tells the kernel to run the bottom half, and exits. The kernel guarantees that the top half is never re-entered: if another interrupt arrives, it is queued until the top half is finished. Because the top half disables interrupts, it has to be fast.

The bottom half is run after the interrupts are processed, or off the timer. The interrupts are not off while the bottom half is run, so it can do slower actions.

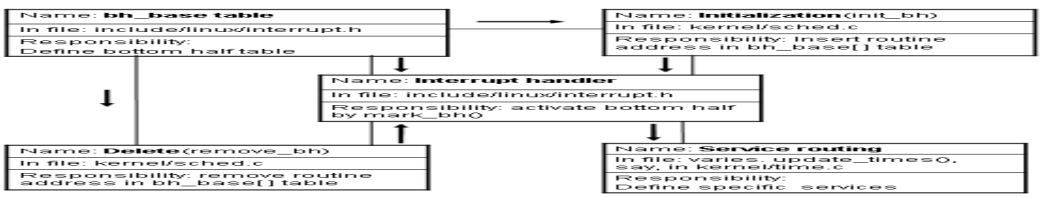
Structure

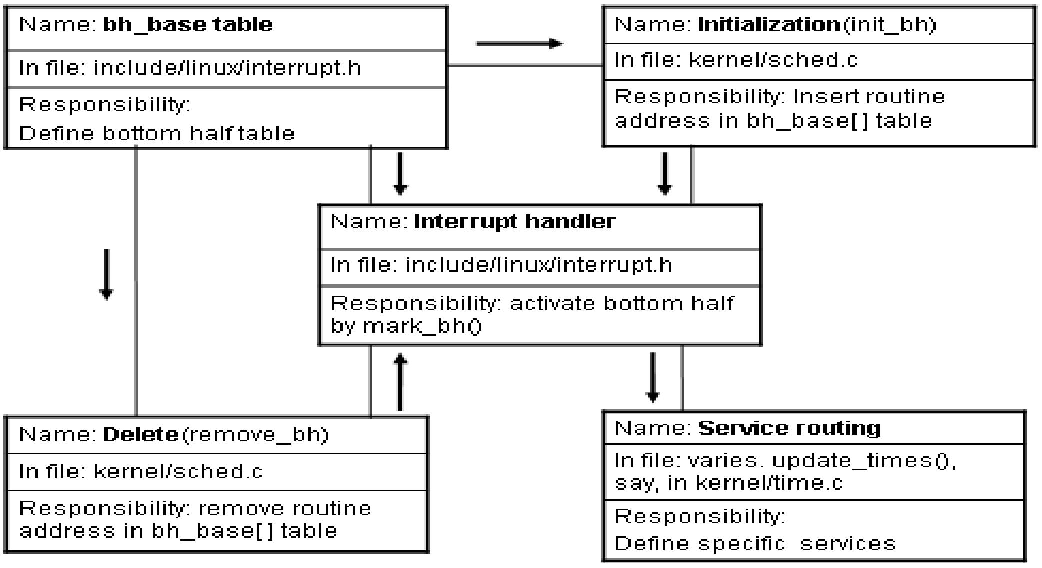
Bottom halves mainly relate to three kernel components(files):

include/linux/interrupt.h
kernel/sched.c
kernel/softirq.c



Linux makes use of an array called the bh_base table to group all bottom halves together. It is an array of pointers to bottom halves and can include up to 32 entries, one for each type of bottom half.





Define the slots for bottom halves

include/linux/interrupt.h

```
/* Who gets which entry in bh_base. Things which will occur most often
should come first */

enum {
TIMER_BH = 0,
TQUEUE_BH,
DIGI_BH,
SERIAL_BH,
RISCOM8_BH,
SPECIALIX_BH,
AURORA_BH,
ESP_BH,
SCSI_BH,
IMMEDIATE_BH,
CYCLADES_BH,
CM206_BH,
JS_BH,
MACSERIAL_BH,
ISICOM_BH
};
```

Init the bottom half functions

kernel/sched.c

```
void __init sched_init(void)
{
/*
* We have to do a little magic to get the first
* process right in SMP mode.
*/
int cpu = smp_processor_id();
```

```
int nr;

init_task.processor = cpu;

for(nr = 0; nr < PIDHASH_SZ; nr++)
pidhash[nr] = NULL;

init_timervecs();

init_bh(TIMER_BH, timer_bh);
init_bh(TQUEUE_BH, tqueue_bh);
init_bh(IMMEDIATE_BH, immediate_bh);
... }
```

Init_bh() installs a bottom half handler pointed to by routine argument into slot nr.

kernel/softirq.c

```
void init_bh(int nr, void (*routine)(void))
{
bh_base[nr] = routine;
mb();
}
```

remove_bh() deletes a bottom half handler pointed to by routine argument from slot nr.

kernel/softirq.c

```
void remove_bh(int nr)
{
tasklet_kill(bh_task_vec+nr);
bh_base[nr] = NULL;
}
```

Mark bottom half in slot nr for execution, so that this bottom half is performed at the next available opportunity.

include/linux/interrupt.h

```
static inline void mark_bh(int nr)
{
tasklet_hi_schedule(bh_task_vec+nr);
}

```

Implementation

There can be up to 32 different bottom half handlers; bh_base is a vector of pointers to each of the kernel's bottom half handling routines. bh_active and bh_mask have their bits set according to what handlers have been installed and are active. If bit N of bh_mask is set then the Nth element of bh_base contains the address of a bottom half routine. If bit N of bh_active is set then the N'th bottom half handler routine should be called as soon as the scheduler deems reasonable. These indices are statically defined; the timer bottom half handler is the highest priority (index 0), the console bottom half handler is next in priority (index 1) and so on. Typically the bottom half handling routines have lists of tasks associated with them. For example, the immediate bottom half handler works its way through the immediate tasks queue (tq_immediate) which contains tasks that need to be performed immediately.

Some of the kernel's bottom half handlers are device specific, but others are more generic: For example, TIMER handler is marked as active each time the system's periodic timer interrupts and is used to drive the kernel's timer queue mechanisms,

Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in bh_active. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The bh_active bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31.

The bit in bh_active is cleared as each bottom half handling routine is called. Thus any activation causes exactly one execution. bh_active is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

Bottom halves that are waiting will be executed only when one of the following events occurs:

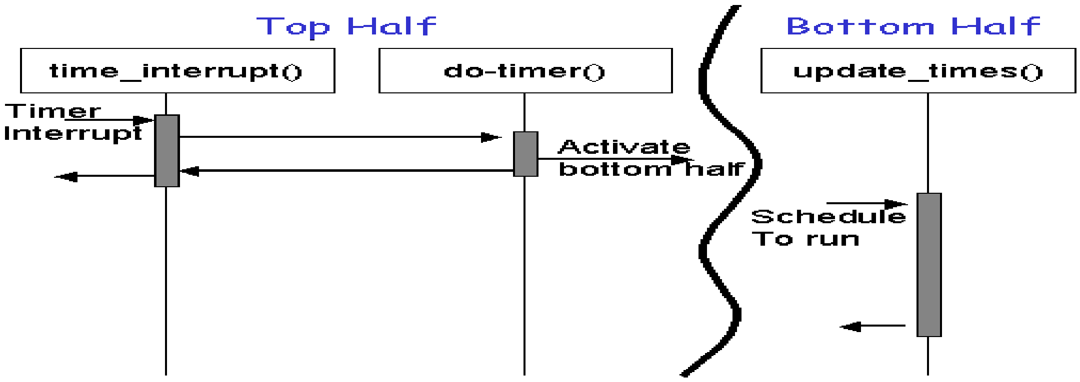
- The kernel finishes handling an exception.
- The kernel finishes handling a system call.
- The kernel finishes handling an interrupt.
- The kernel executes the schedule() function to select a new process.

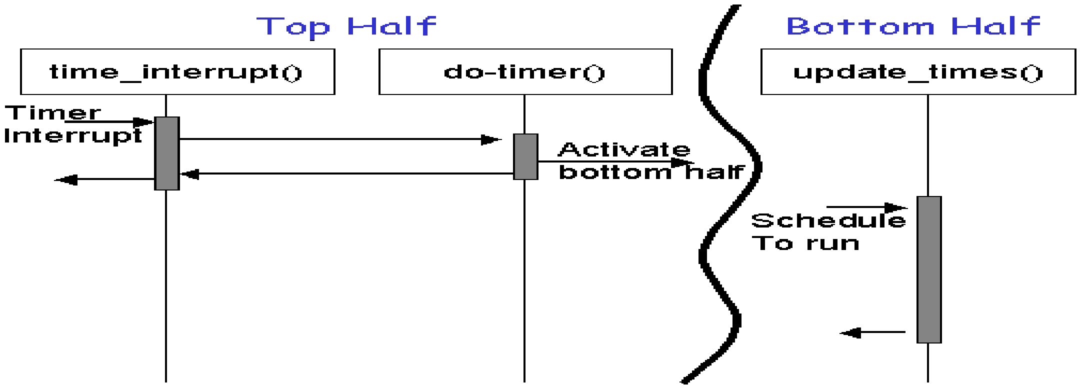
Dynamics and Example

Scenario I shows how the kernel updates the Time using bottom halves.

Kernel needs timing measurements to: keep track of current time and date for use by e.g. gettimeofday(), and maintain timers that notify the kernel or a user program that an interval of time has elapsed.

Every timer interrupt: Update jiffies; Update time and date; Determine how long a process has been executing and preempt it, if it finishes its allocated timeslice; Update resource usage statistics; Invoke functions for elapsed interval timers.





when signal on IRQ 0 is generated, timer_interrupt() is invoked w/ interrupts disabled (SA_INTERRUPT flag is set to denote this). do_timer() is ultimately executed. Simply increments jiffies & allocates other tasks to "bottom half handlers". Bottom half (bh) handlers update time and date, statistics, execute fns after specific elapsed intervals and invoke schedule() if necessary, for rescheduling processes.

```
kernel/timer.c
674 void do_timer(struct pt_regs *regs)
675 {
676 (*(unsigned long *)&jiffies)++;
677 #ifndef CONFIG_SMP
678 /* SMP process accounting uses the local APIC timer */
679
680 update_process_times(user_mode(regs));
681 #endif
682 mark_bh(TIMER_BH);
683 if (TQ_ACTIVE(tq_timer))
684 mark_bh(TQUEUE_BH);
685 }
```

do_timer() runs with the interrupts disabled, and must be executed as quickly as possible. It simply updates one fundamental value while delegating all remaining activities to two bottom halves.

The timer_bh() associated with the TIMER_BH bottom half invokes the update_times(), run_timer_list().

```
kernel/timer.c
668 void timer_bh(void)
669 {
670 update_times();
671 run_timer_list();
672 }
```

The update_times() function invoked by the TIMER_BH bottom half updates xtime by disabling interrupts.

```
kernel/timer.c
640 /* jiffies at the most recent update of wall time */
641 unsigned long wall_jiffies;
```

```

/*
* This spinlock protect us from races in SMP while playing with xtime.
*/
rwlock_t xtime_lock = RW_LOCK_UNLOCKED;

static inline void update_times(void)
{
unsigned long ticks;

/*
* update_times() is run from the raw timer_bh handler so we
* just know that the irqs are locally enabled and so we don't
* need to save/restore the flags of the local CPU here. -arca
*/
write_lock_irq(&xtime_lock);

ticks = jiffies - wall_jiffies;
if (ticks) {
wall_jiffies += ticks;
update_wall_time(ticks);
}
write_unlock_irq(&xtime_lock);
calc_load(ticks);
}

```

Whenever some code wants to schedule a bottom half for running, it calls mark_bh. In the older BH implemention, mark_bh would set a bit in a bit mask, allowing the corresponding bottom-half handler to be found quickly at runtime. In modern kernels, it just calls tasklet_scheduleto schedule the bottom-half routine for execution.

Variants

Software interrupt

Similar to linux 2.2 bottom halves, it lets you to defer the execution of a kernel function. The difference between the software interrupt and the bottom halves: bottom halves are strictly serialized (because no 2 bottom halves can be executed in the same time), the software interrupt is not serialized. So it will give the possibility that to run 2 the same interrupts on 2 CPUs, in this case software interrupt must be reentrant. Networking greatly benefits from it, because it's much more efficient for the multiprocessor system when it uses 2 software interrupts rather than the old NET_BH bottom half

Tasklet

mechanism similar to bottom half, built on top of software interrupts but serialized. 2 CPU can execute 2 tasklets at the same time, but these tasklets must be different.

Linux 2.4 Bottom half

Bottom half continues to exist in Linux 2.4, but now built on top of tasklets. They still can not run at the same time even in the different CPU. Bottom halves degrade significantly the performance of multiprocessor systems.

Task queues

Task queues can be though of as a dynamic extension to old bottom halves.

See Also

Known Uses

The bottom halves do a great deal of work in Linux, including much of the networking receive code, all the timer handling, and much of the terminal driver. Example of network interface: Top Half gets packet data; Bottom Half processes the data.

Consequences

The application of Bottom Halves has several benefits:

- Bottom halves are the oldest mechanism for deferred execution of kernel tasks and have been available since Linux 1.x.
- The existence of bottom halves is very important to fulfill the kernel's responsibility to service interrupts from multiple devices quickly.
- Device drivers and other parts of the Linux kernel could queue work to.

Bottom Halves have several liabilities:

- One important drawback: due to their structure, their execution was serialized strictly among CPUs. That is, the same BH could not be executed by more than one CPU at the same time. This obviously prevented any kind of kernel parallelism on SMP machines and seriously affected performance. softirqs represent the 2.4-age evolution of BHs and, together with tasklets, belong to the family of kernel software interrupts, pieces of code that can be executed by the kernel when requested, without strict response-time guarantees.
- Bottom halves are consumed with a spinlock held so they cannot block.

Previous Home Next

Find...

Edit Rename Changes History Upload Download Back to Top