SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(8), 883–896 (10 JULY 1998)

Redesigning the BSD Timer Facilities*

adam m. costello[1] and george varghese[2]
[1]*University of California at Berkeley, Computer Science Division, 475 Soda Hall #1776, Berkeley, CA 94720-1776, USA.*
http://www.cs.berkeley.edu/~amc/
(email: amc@cs.berkeley.edu)
[2]*Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA.*
http://www.ccrc.WUSH.edu/~varghese/
(email: varghese@cs.wustl.edu)

SUMMARY

We describe a reimplementation of the BSD timer facilities. Older BSD kernels take time proportional to the number of outstanding timers to set or cancel timers. Our implementation (in NetBSD) takes constant time to start, stop, and maintain timers; this leads to a highly scalable design that can support thousands of outstanding timers without much overhead. Unlike the existing implementation, our routines are guaranteed to lock out interrupts only for a small, bounded amount of time. We also extend the `setitimer()` interface to allow a process to have multiple outstanding timers, thereby reducing the need for users to maintain their own timer packages. The changes to the kernel are small (548 lines of code added, 80 removed) and are available on the World Wide Web. © **1998 John Wiley & Sons, Ltd.**

key words: timer; timing wheel; BSD; NetBSD callout

INTRODUCTION

To satisfy the needs of real-time applications and communication protocols, operating systems provide timers. In BSD† kernels, each timer (called a *callout*) registers a function and argument to be called at a future time. A clock interrupt routine periodically checks the outstanding timers, and calls the functions for any due to expire.

There are four measures of performance of this timer facility:

1. Time to set a timer.
2. Time to cancel a timer.
3. Time to process a clock tick.
4. Time to process an expired timer.

Also of interest is the maximum amount of time that interrupts can be locked out

Contract/grant sponsor: NSF; contract/grant number: NCR 940997.

* This paper is a slight revision of Washington University Computer Science Technical Report 95-23 published on 2 November 1995.

Throughout this paper, "BSD" refers to 4.4BSD-Lite and its derivatives, such as NetBSD and FreeBSD.

CCC 0038–0644/98/080883–14$17.50

Received 17 July 1997
Revised and Accepted 27 February 1998

during these operations. If interrupts are locked out for too long, the system might miss time-critical I/O events.

The existing implementation takes constant time (usually) to process a clock tick or an expired timer, but takes linear time (in the number of outstanding timers) to set or cancel a timer. Interrupts are locked out for the duration of a set or cancel operation. Our new implementation achieves worst-case constant time for all operations except processing a clock tick, for which it achieves average-case constant time. It also never locks out interrupts for more than a constant amount of time.

Why redesign?

When we examined the BSD kernel, we found only five or six outstanding timers (used by RPC, NFS, and TCP). With such a small number of outstanding timers, taking linear time to start or stop a timer is not very onerous, and the BSD implementation seems adequate. So why bother with a new implementation anyway?

Clearly, our new implementation is motivated by future environments which have a large number of outstanding timers. Network protocol implementations provide the easiest example. Typically, for every packet sent by a reliable transport protocol such as TCP, a retransmission timer is started. If an acknowledgement arrives promptly, the timer is cancelled; when the timer expires, the packet is retransmitted. Several network implementations[1] have been tuned to send packets at a rate of 25,000–40,000 packets per second. The situation is exacerbated because most transport protocols use sliding window protocols (in which a large number of packets are allowed to be outstanding to allow pipelined throughput) and because servers often maintain several outstanding conversations with clients. Most other network protocols also use several timers for error recovery and rate control.

How then can the BSD TCP implementation get away with two timers? This is possible because the TCP implementation maintains its own timers for all outstanding packets, and uses two kernel timers as clocks to run its own timers. TCP maintains its packet timers in the simplest fashion: whenever its kernel timer expires, it ticks away at all its outstanding packet timers. This method works reasonably well if the granularity of timers is low.* However, it is desirable to improve the resolution of the retransmission timer to allow speedier recovery. With a large number of finer granularity timers, it is necessary to have more efficient timer algorithms. Rather than have every protocol duplicate these efficient algorithms,[1,2] it is better to have the kernel directly provide an efficient timer facility.

Besides networking applications, other real-time applications like process control and performance monitors will also benefit from large numbers of fine granularity timers. Also, the number of users on a system may grow large enough to lead to a large number of outstanding timers. This is the reason cited (for redesigning the timer facility) by the developers of the IBM VM/XA SP1 operating system.[3]

With a large number of outstanding timers, it becomes important to bound the amount of time that interrupts can be locked out. Finally, regardless of the number of timers, the current limit of one user-level timer per process seems too restrictive. This forces user applications that need multiple timers either to fork additional processes or to maintain their own timers.

* Currently TCP uses two virtual clocks: one with 200 ms ticks and one with 500 ms ticks.

Therefore, we believe we have a case for new kernel timer and user-level timer implementations that are scalable, robust, and flexible. Unlike many scalable designs, there is no performance penalty when the number of outstanding timers is small. Also, the increase in the code size is small (158 lines for the new kernel timer facility, and 310 lines for the extended user-level timer facility). Thus, considering the general environments in which UNIX kernels are and will be deployed, the new implementation appears to be a wise and cheap piece of insurance.

Previous work

Varghese and Lauck[4] described a number of new schemes to implement timers; our implementation is based on Scheme 6, which centers around a data structure called a *hashed timing wheel*. The description of the algorithm is mostly theoretical and does not consider some of the issues that occur in actual operating systems; for example, it assumes that all the per tick bookkeeping can be done at the interrupt level, which is clearly infeasible in a real system. A few fairly well-known networking implementations[1,2] have used timing wheels in specialized timer packages for their networking routines (as opposed to a general operating system facility). Brown[5] extends hashed timing wheels to *calendar queues*; the major difference is that calendar queue implementations also periodically resize the wheel in order to reduce the overhead* of stepping through empty buckets. For timer applications, the clock time must be incremented on every clock tick anyway, so adding a few instructions to step through empty buckets is not significant. Davison[3] describes a timer implementation for the IBM VM/XA SP1 operating system based on calendar queues. In our setting, the small improvement in per-tick bookkeeping (from resizing the wheel periodically) does not appear to warrant the extra complexity of resizing.

Subsequent work

Justin Gibbs (email: gibbs@freebsd.org) has ported our kernel timer implementation to FreeBSD, fixing a slight bug and changing the interface. The bug and its fix (but not the new interface) are described in footnotes in later sections.

Organization

The rest of the paper is organized as follows. The next section describes the existing implementation of the kernel timer facility in NetBSD 1.0. We review the timing wheel algorithm,[4] and illustrate how we implemented this algorithm in the NetBSD 1.0 kernel and the implementation issues we encountered. We also describe an extension to the user-level timer facility to allow multiple outstanding timers; this was essential for the performance tests of the new kernel timer facility, but we believe this feature is useful in its own right. Performance results are presented, and future work in the areas of dynamic storage allocation and cleaning up the user-level timer interface described. Finally, we state our conclusions.

* The improvement is not worst-case, but is demonstrated empirically for certain benchmarks.

OLD TIMER IMPLEMENTATION

In the existing BSD implementation, each timer is represented by a **callout** structure containing a pointer to the function to be called (**c_func**), a pointer to the function's argument (**c_arg**), and a time (**c_time**) expressed in units of clock ticks. Outstanding timers are kept in a linked list, sorted by their expiration times. The **c_time** member of each structure is differential, not absolute—the first timer in the list stores the number of ticks from now until expiration, and each subsequent timer in the list stores the number of ticks between its own expiration and the expiration of its predecessor.

This data structure (called **calltodo**, see Figure 1) dictates the performance. Because the times are differential, the clock interrupt routine **hardclock()** needs only to decrement the time of the first timer in the list and check whether it becomes zero, which takes constant time. If the first timer has expired, a software clock interrupt is generated. Its handler, **softclock()**, repeatedly checks the timer at the head of the list, and if it is expired (**c_time** member equal to zero), removes it and calls its function. Interrupts are locked out while the **calltodo** list is being manipulated, but not while the function is executing.

Timers are set and canceled using the following interface: **timeout(func, arg, time**) registers **func(arg)** to be called at the specified time, and **untimeout(func, arg**) cancels the timer with matching function and argument. Because the **calltodo** list must be searched linearly, both operations take time proportional to the number of outstanding callouts. Interrupts are locked out for the duration of the search.

There is one complication: because the processing of expired timers by **softclock()**, which can be interrupted by **hardclock()**, might take longer than one clock tick, timers might become overdue. This is represented in the **calltodo** list by timers at the start of the list with negative **c_time** members; a value of $-t$ indicates that the callout is overdue by t ticks. **hardclock()** must decrement all non-positive **c_time** members at the beginning of the list, as well as the first positive **c_time** member. Therefore, the processing of a clock tick does not necessarily take constant time.

NEW TIMER ALGORITHM

The algorithm used by the existing implementation is very similar to Scheme 2 of Varghese and Lauck.[4] The new implementation is based on Scheme 6.

Instead of a single sorted list of timers, we use a circular array of unsorted lists. The array, called **callwheel** (see Figure 2), contains **callwheelsize** entries. All timers scheduled to expire at time t appear in the list **callwheel** [t **% callwheelsize**], and their **c_time** members are set to t **/ callwheelsize. callwheelsize** should be comparable to the maximum possible number of outstanding callouts.

On each clock tick, the appropriate list must be traversed completely, the **c_time** member of each timer in the list decremented, and the expired timers handled and removed. In the worst case, all outstanding timers could be in the same list, but

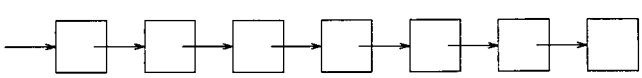

Figure 1. The **calltodo** data structure

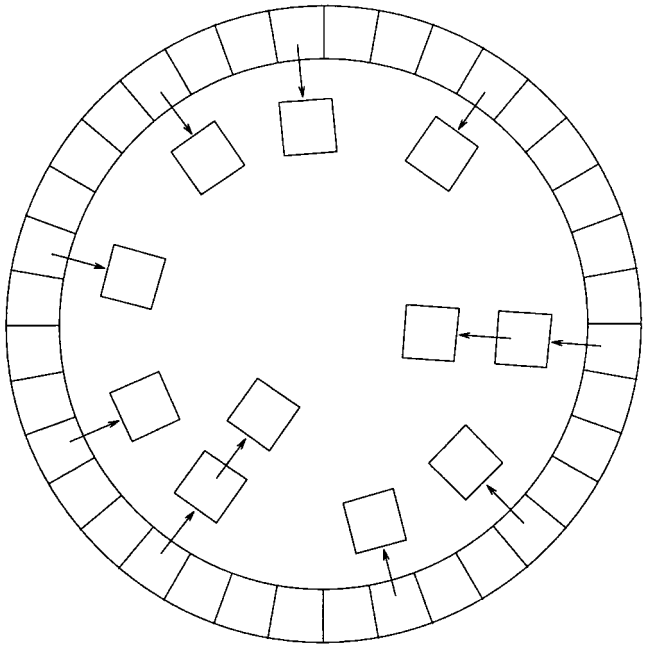

Figure 2. The **`callwheel`** *data structure*

since the number of lists is comparable to the maximum number of outstanding timers, and the lists are processed in a round-robin fashion, the average list length is a small constant. Thus, it takes average-case constant time to process a clock tick. To handle an expired timer still requires nothing more than removing the timer from a list, which take worst-case constant time, and then calling its function.

Setting a timer requires determining the appropriate list and inserting at the head,* which takes worst-case constant time. For canceling a timer, the original Scheme 6 algorithm assumes that it will be given a handle to the timer (i.e. an identifier which locates the timer immediately), in which case it needs only to delete the timer from the list, taking worst-case constant time. But the existing **`timeout()`** interface does not use handles, so this becomes a problem in the implementation. In the next section we describe how this problem has been overcome in two different ways. The original algorithm also assumes that each timer routine (especially the equivalent of **`hardclock()`**, runs to completion, and does not worry about the details of mutual exclusion and locking out interrupts. These problems are also addressed.

* This is a bug. When two timers have the same expiration time, you want them to expire in the order they were scheduled (that's what the existing implementation does), which means you want to insert at the tail. Justin Gibbs has fixed this bug in his port of this implementation to FreeBSD. Inserting at the tail requires a pointer to the tail, which is easy (use the 'previous' pointer of the first structure in the list), but then you need to be able to locate and repair that pointer when you delete the last item in the list. Justin's elegant solution was to store *t* instead of *t* / **`callwheelsize`** in **`c_time`**, so that **`callwheel[p->c_time % callwheelsize]`** locates the list containing timer **`p`**. Of course all code dealing with **`c_time`** needed to be updated use the new representation.

NEW TIMER IMPLEMENTATION

callwheelsize is constrained to be a power of 2, equal to 2**callwheelbits**. This simplifies the division and remainder operations to bit shifting and masking. The lists in **callwheel** are doubly-linked (requiring an additional member in the **callout** structures), allowing the removal of a timer given only a pointer to it.

The old **hardclock()** had to manipulate the **calltodo** structure, but the new one has nothing to do but increment the global **ticks** variable* (which the old **hardclock()** did anyway), which represents the current time. **softclock()** repeatedly checks whether its own static variable **softticks** equals **ticks**, and if not, increments **softticks** and checks all the timers in **callwheel[softticks & callwheelmask]**, expiring the ones with **c_time** equal to zero and decrementing **c_time** for the others. As an optimization, after **hardclock()** increments **ticks**, it checks to see if **softticks** equals **ticks** – **1** and **callwheel[ticks & callwheelmask]** is empty. If not, a software interrupt is generated to cause **softclock()** to run. Otherwise, **hardclock()** increments **softticks** itself rather than incur the overhead of a software interrupt, since that is all **softclock()** would have done.

Figures 3, 4, and 5 show the new code for **hardclock()** and **softclock()**.

If it were not for the lack of handles in the existing **timeout()**/**untimeout()** interface, the implementation of those functions would be straightforward, and there would be nothing to say about them here. There are two possible solutions to the problem. We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching timer in constant time. A hash table is the obvious mechanism. The disadvantage is that a hash table cannot guarantee constant time in the worst case; it can only provide constant time in the expected case. The other solution would be to provide new interface functions called, say, **setcallout()** and **unsetcallout()**, which work very much like **timeout()** and **untimeout()**,

```
struct callout {
    struct  callout *c_next;          /* next callout in queue */
    struct  callout **c_back;         /* pointer back to the ptr */
                                      /* pointing at this struct */
    struct  callout *hash_next;       /* next and back pointers */
    struct  callout **hash_back;      /* for the callhash array */
    /* hash_back is NULL iff this callout is not in the hash table. */
    struct callout_handle {
        unsigned long lo, hi;
    } handle;                         /* handle for this callout */
    /* The lowest calloutbits of handle.hi are the   */
    /* index into the callout array of this callout. */
    void    *c_arg;                   /* function argument */
    void    (*c_func) __P((void *));  /* function to call */
    int     c_time;                   /* ticks to the event >> callwheelbits */
};

int ticks;                             /* Current time, in ticks. */
static int softticks;                  /* Like ticks, but for softclock(). */
static struct callout *nextsoftcheck;  /* Next callout to be checked. */
```

Figure 3. Declarations used by **hardclock()** *and* **softclock()**

* Actually, **hardclock()** also does other things entirely unrelated to the timer facility.

```
void hardclock(frame)
    register struct clockframe *frame;
{
    /* The real hardclock() also does other, non-callout-related things. */

    ticks++;

    if (callwheel[ticks & callwheelmask]) {  /* This condition changed. */
        if (CLKF_BASEPRI(frame)) {
            /* Save the overhead of a software interrupt;         */
            /* it will happen as soon as we return, so do it now. */
            (void)splsoftclock();
            softclock();
        } else
            setsoftclock();
    }
    else if (softticks + 1 == ticks) ++softticks;  /* This line is new. */
}
```

Figure 4. The new **hardclock()** *function. From the old function, several lines dealing with* **calltodo** *have been removed, one line has been changed, and one line has been added*

except that **setcallout()** returns a handle, and **unsetcallout()** takes a handle as its only argument. The disadvantage here is that existing code elsewhere in the kernel already uses the existing interface. In the new implementation, both solutions are used. Both interfaces are available for adding and removing timers to and from a single **callwheel**. A hash table is used only by the old interface.

At first, we used a non-chaining hash table. If **untimeout(func, arg)** were called, and no outstanding timer matched the specified function and argument, the entire hash table would have to be searched. We did not expect **untimeout()** ever to be called this way, but it turns out that such calls are quite common. In the old implementation, the cost of canceling a non-existent timer was about the same as canceling an existent timer (both require a linear search of the **calltodo** structure). We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to a chaining hash table, in which only one bucket needs to be searched in any case. This required the addition of two more members to the **callout** structure, so that each timer, which might already belong to a doubly-linked list in **callwheel**, might also belong to a doubly-linked list in **callhash** (the hash table). **timeout()** always inserts the new timer into the hash table, and **untimeout()** always removes it, but **softclock()** must check to see whether an expiring timer is in the hash table before removing it. Fortunately, it need not compute the hash function to do this; it can merely check the links in the **callout** structure.

The hash function was chosen to be fast to compute, but still provide a fairly even distribution among the buckets. **callhashsize** is constrained to be a power of 2, equal to 2**callhashbits**. Both the **func** and **arg** pointers are likely to be longword-aligned, so the lowest two bits of each are not used. The lower bits of function pointers are likely to be more random than the higher bits, so the next higher **callhashbits** bits of **func** are used. For data pointers, the low bits might be constant (in the case of pointers to large aligned structures, such as mbufs) or they might be the only bits that are significant (in the case of pointers into an array, for example). Therefore, we use not only the next higher **callhashbits** bits from **arg**, but also the next higher **callhashbits** bits above those. The three bit strings,

```
void softclock()
{
    register struct callout *c;
    register int steps;          /* Number of steps taken since */
                                 /* we last allowed interrupts. */
    register int s;

    s = splhigh();
    steps = 0;
    while (softticks != ticks) {
        if (++steps >= MAX_SOFTCLOCK_STEPS) {
            nextsoftcheck = NULL;
            splx(s);  /* Give hardclock() a chance. */
            (void) splhigh();
            steps = 0;
        }
        c = callwheel[++softticks & callwheelmask];
        while (c) {
            if (c->c_time > 0) {
                --c->c_time;
                c = c->c_next;
                if (++steps >= MAX_SOFTCLOCK_STEPS) {
                    nextsoftcheck = c;
                    splx(s);  /* Give hardclock() a chance. */
                    (void) splhigh();
                    c = nextsoftcheck;
                    steps = 0;
                }
            }
            else {
                if (nextsoftcheck = *c->c_back = c->c_next)
                    nextsoftcheck->c_back = c->c_back;
                if (c->hash_back) callhash_remove(c);
                else if (++c->handle.lo == 0) c->handle.hi += calloutsize;
                splx(s);
                c->c_func(c->c_arg);
                (void) splhigh();
                steps = 0;
                c->c_next = callfree;
                callfree = c;
                c = nextsoftcheck;
            }
        }
    }
    nextsoftcheck = NULL;
    splx(s);
}
```

Figure 5. The new **`softclock()`** *function. It is entirely different from the old one*

each of length **`callhashbits`**, are aligned and XORed to produce the index into the hash table. The computation requires three shifts and three bitwise boolean operations.

For the new **`setcallout()`** interface, the handle could not simply be a pointer to the timer, because the **`callout`** structures get recycled. Between the time **`setcallout()`** returns a pointer and the time **`unsetcallout()`** is passed that pointer, the timer could have expired and the structure reused, in which case the call to **`unsetcallout()`** will cancel someone else's timer. The solution was to give each timer an ID that gets incremented whenever the timer expires or is canceled.

The ID is included in the handle, so that **unsetcallout()** can compare the IDs before canceling the timer. The size of the ID must be large enough that it will never wrap around. 32 bits was deemed insufficient. But the handle also needs to include a 'pointer' to the **callout** structure, and a real pointer is overkill. Since all of the **callout** structures are allocated as a single array at boot time, we can use an index into this array, which takes **calloutbits** bits (which is much less than 32). So a **callout_handle** structure contains two unsigned longs, **lo** and **hi**. The lowest **calloutbits** bits of **hi** are the index into the **callout** array, and all remaining bits are used for the ID. Incrementing the ID amounts to incrementing **lo**, and if the result is zero, adding **calloutsize** = 2**calloutbits** to **hi**. Each **callout** structure contains a full **callout_handle** structure, with the lowest **calloutbits** bits of **hi** initialized at boot time, so that **unsetcallout()** may compare the entire handle, rather than just the ID portion.

The only remaining wrinkle in the implementation is motivated by the desire to limit the time that interrupts may be disabled. **timeout()** and **untimeout()**, which use the hash table, cannot guarantee a limit, but **setcallout()** and **unsetcallout()** can (and do). **hardclock()** always takes constant time.* The remaining concern is **softclock()**, which must lock out interrupts while it is manipulating the **callwheel** structure. While it is traversing a list (which could be long in the worst case), it cannot allow itself to be interrupted by **untimeout()** or **unsetcallout()**, because they might remove a timer right out from under **softclock()**. This problem was eliminated by a bit of unholy data-sharing between the three functions. As **softclock()** traverses a list, it uses a local pointer into the list. Whenever it wishes to enable interrupts briefly, it first copies its local pointer into the global variable **nextsoftcheck**, then enables interrupts, then disables interrupts, then copies **nextsoftcheck** back into its local pointer. Whenever **untimeout()** or **unsetcallout()** cancels a timer, it checks to see whether **nextsoftcheck** points to the structure just removed, and if so, fixes it to point to the next structure in the list.† **softclock()** keeps track of the number of steps it has taken since it last enabled interrupts, and whenever the count reaches **MAX_SOFTCLOCK_STEPS**, it briefly enables them. Therefore, **softclock()** never disables interrupts for more than a constant amount of time.

As in the old implementation, it is still possible for timers to expire late. If processing the expiring timers takes too long, **softticks** will lag behind **ticks**.

MULTIPLE USER-LEVEL TIMERS

The original motivation for extending the user-level timer facility to provide multiple timers per process was to allow the performance of the new kernel timer implementation to be tested by a user-level program. A process could cause timers to be set, canceled, and expired through the **getitimer()** and **setitimer()** system calls, but it was allowed only one outstanding real-time timer.

Although our need was very specialized, we believe that users will benefit from the ability to allow each process to have multiple outstanding real-time timers. For example, a protocol implemented in user space, or a daemon that polls for various

* For its timer-related duties, that is.

† If **timeout()** and **setcallout()** are fixed to insert at the tail rather than the head, then they too must participate in this **nextsoftcheck** business, because they might extend a list that **softclock()** thinks it has just finished traversing.

conditions at varying intervals, would be freed from having to build its own timer facility on top of the system facility. Our extensions involved adding a new timer type tag, **ITIMER_MULTIREAL**, and overloading the semantics of the other arguments passed to **getitimer()** and **setitimer()**.

The details of the interface extensions are described in Figures 6 and 7. The extended interface allows the user process to create many real-time timers, which have associated with them 64-bit kernel-chosen handles and 64-bit user-chosen labels. The user process can change the values and labels of existing timers, and destroy existing timers. Whenever a timer expires, the timer is put in a queue, and a **SIGALRM** is sent to the process. The process can pop an expired timer off of this queue and obtain both its handle and its label.

Implementing this extension required new code in the **getitimer()** and **setitimer()** system calls, of course. Also, two new pointer members were added to the **proc** structure to keep track of running and queued timers belonging to the processes.* Since each user-level timer is represented by an **mrtimer** structure allocated from a shared pool, code needed to be added to **exit()** to free any timers still belonging to the dying process. Each structure contains a pointer to the process that owns it, so one process cannot forge handles and trick the kernel into manipulat-

The existing interface, `setitimer(ITIMER_REAL, value, ovalue)`, sets the process's real-time interval timer to `value->it_value`, putting the former contents into `ovalue` if `ovalue` is not `NULL`. The timer decrements in real time, causing a `SIGALRM` when it reaches zero. If `value->it_interval` was not zero, the timer reloads with that value, otherwise it stops.

The new interface, `setitimer(ITIMER_MULTIREAL, value, ovalue)`, behaves similarly, but with the following differences:

- ovalue *must not* be `NULL`; it must point to a `struct itimerval`.
- If `value->it_value` is not zero, and `ovalue->it_value` is `{-1, -1}`, this is a create-new-timer operation. `ovalue->it_interval` must already be set to an arbitrary user-chosen 'label' for the new timer. `ovalue->it_value` will be overwritten with a kernel-chosen 'handle' for the timer. The timer will be initialized with `value`.
- If `value->it_value` is not zero, and `ovalue->it_value` is not `{-1, -1}`, this is a reset-existing-timer operation. `ovalue->it_value` is interpreted as a timer handle, and `ovalue->it_interval` must already be set to an arbitrary user-chosen label, which may differ from the existing timer's current label. The existing timer is removed from the expired timer queue (see `getitimer()`) if necessary, and its value and label are overwritten with `value` and `ovalue->it_interval`. As with `ITIMER_REAL`, `ovalue` is overwritten with the former value of the timer. If the handle in `ovalue->it_value` did not refer to an existing timer, -1 is returned and `errno` is set to `EINVAL`.
- If `value->it_value` is zero, this is a cancel-timer operation. `value->it_interval` and `ovalue->it_interval` are ignored. `ovalue->it_value` is interpreted as a timer handle. If the handle is valid and the timer exists, it is destroyed, and its final value is stored in `ovalue`. Otherwise, -1 is returned and `errno` is set to `EINVAL`. Note that timers in the expired queue (see `getitimer()`) are still considered to exist.

Figure 6. Extensions to **setitimer()**

* It would have been easier to use three pointers, but there were already eight bytes of padding in the structure. By using only two pointers, it was possible to add the members without changing the size of the structure.

The existing interface, `getitimer(ITIMER_REAL, value)`, writes the current contents of the process's real-time interval timer into `value`.

The new interface, `getitimer(ITIMER_MULTIREAL, value)`, behaves similarly, but with the following differences:

- `value->it_value` must be set by the user before the call. `value->it_interval` is ignored.
- If `value->it_value` is {`-1, -1`}, this is a reap-expired-timer operation. As timers expire, they are enqueued. This call removes the timer at the head of the queue, writes its handle into `value->it_value`, and writes its label into `value->it_interval`. If the timer is not set up to reload, it is destroyed, otherwise it starts running again (its next expire time is based on it last expire time, not the time at which it was reaped). If the queue is empty, -1 is returned and `errno` is set to `EAGAIN`. When a process exits, all of its running and queued timers are destroyed. It is impolite for a long-lived process to let timers expire and neither reap them nor cancel them, because that wastes system resources.
- In all other cases, `value->it_value` is interpreted as a timer handle. If the handle is valid and the timer exists, `value` is overwritten with the current timer value. Otherwise, it is left untouched, -1 is returned, and `errno` is set to `EINVAL`.

Figure 7. Extensions to **`getitimer()`**

ing timers belonging to another process. The user-level timers are built on top of kernel timers, so they have the option of using either the **`timeout()`** interface or the **`setcallout()`** interface. Code exists for both, and the choice is made statically when the kernel is compiled.

PERFORMANCE

Three kernels were tested on a Sun 4/360. All included the user-level timer extension. In one kernel, the user-level timers used the **`timeout()`** interface to the old kernel timer facility. In another kernel, the user-level timers used the **`timeout()`** interface to the new kernel timer facility. In the last kernel, the user-level timers used the **`setcallout()`** interface to the new kernel timer facility.

In each test, one process created a number of outstanding timers set for random times far in the future, causing a number of outstanding callouts. It then created one more timer, and repeatedly set it for a random time farther in the future than the others, causing repeated calls to **`untimeout()`** and **`timeout()`** (or **`unsetcallout()`** and **`setcallout()`**, depending on which kernel was being used). The results show that the time for the original kernel timer implementation increases linearly with the number of outstanding timers, whereas the time for the new implementation is constant with respect to the number of outstanding timers, for both the old interface (using hashing) and the new interface (using handles) (see Figure 8 and Table I). The new interface is very slightly faster, and provides guaranteed constant time operations, but the old interface is needed for compatibility with the rest of the kernel.

The exact number of outstanding timers cannot be known exactly, because other parts of the kernel also use timers. However, a search through the kernel code revealed only about five or six of them, used by RPC, NFS, and TCP.

Table II shows the number of instructions required for the execution of each

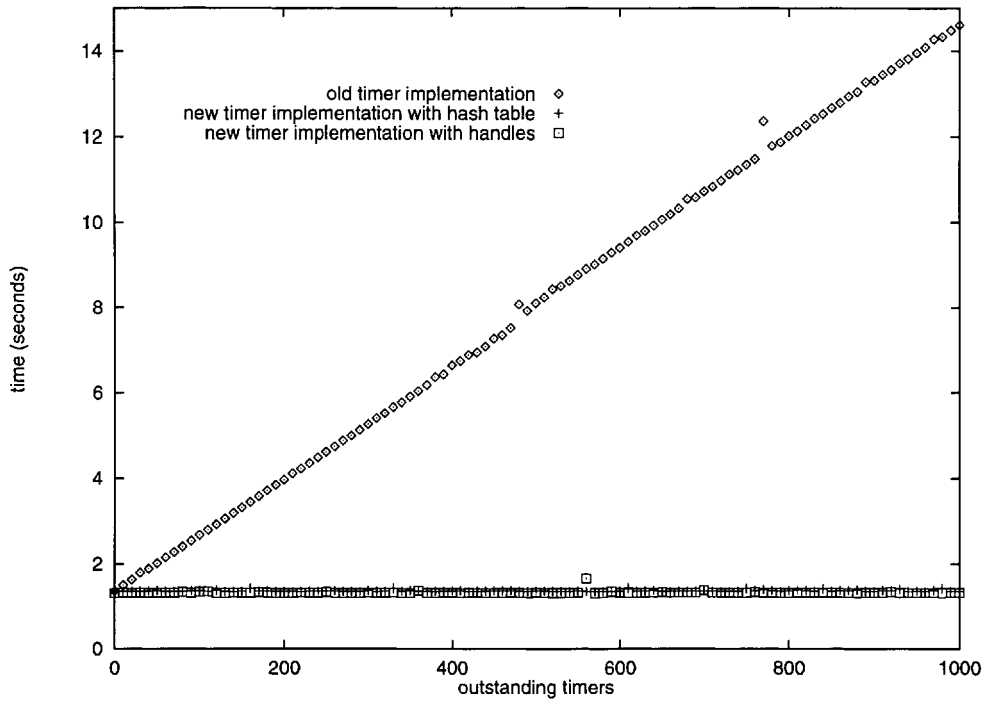

Figure 8. Real-time performance comparison of kernel timer implementations, showing the time to perform 10,000 set/cancel iterations on one timer while other timers are outstanding

Table I. Selected data points from the graph of Figure 8

Kernel	Number of outstanding timers					
	0	200	400	600	800	1000
old callout	1.367	3.969	6.638	9.404	12.017	14.608
new callout with hash table	1.366	1.360	1.377	1.363	1.379	1.364
new callout with handles	1.319	1.325	1.329	1.327	1.325	1.329

function. The longest paths were counted (except a few paths used in only one out of every 2^{32} cases). The compiler used was gcc 2.4.5 for SPARC (which came with NetBSD) with the **–O2** option.

The speed of the new **hardclock()** and **softclock()** was not measured, but the instruction counts suggest that **hardclock()** is about as fast as before, while **softclock()** is about half as fast as before (given that usually $i = 1$, $j = 0$, and $k + m + n \leq 1$).

The memory usage of the new implementation is within a constant factor of the original. Originally, n **callout** structures required $4n$ longwords. The new implementation needs $9n$ longwords for them, plus n to $2n$ for the **callwheel** array, plus $2n$ to $4n$ for the **callhash** array (unless the vast majority of the **callout** structures

Table II. Instruction counts for kernel timer functions

Function	Instruction count	
	New version	Original
hardclock	37	$34 + 11v$
softclock	$27 + 20i + 15j + 12k + 54m + 37n$	$27 + 28e$
timeout	81	$46 + 11t$
untimeout	$80 + 11h$	$45 + 11t$
setcallout	60	
unsetcallout	55	

e timers expire.
h timers in a **callhash** bucket do not match.
i: **softclock** is incremented *i* times.
j: **MAX_SOFTCLOCK_STEPS** is reached *j* times.
k unexpired timers are traversed.
m hashed timers expire.
n non-hashed timers expire.
t timers are skipped in a search of **calltodo**.
v timers are overdue at start of **calltodo**.

are intended for use with the **setcallout()** interface, in which case **callhash** is very small).

The user-level timer extension requires ten longwords per timer, plus two per process.

FUTURE WORK

Currently, the kernel reboots if it runs out of **callout** structures. There was a comment in the old code saying that new structures should be allocated dynamically. Should that wish be fulfilled, the **callout_handle** structure will have to be enlarged, because it will no longer be able to use an index into a single **callout** array—it will have to use an honest-to-goodness pointer. The ID will then have two unsigned longs all to itself.

There is a caveat: increasing the maximum possible number of outstanding timers without increasing the size of **callwheel** or **callhash** could degrade performance, because the average list lengths could increase. Allowing **callwheel** and **callhash** to grow dynamically is conceivable, but probably not worth the effort.

The appearance of the extended user-level timer facility ought to be improved. Although its design is functionally clean, the overloading of the **value** and **ovalue** parameters is distasteful. This overloading should be hidden from the user, or there should at least be alternate names for the structure members, corresponding to their new uses.

After using the two longwords of padding in the **proc** structure for the user-level timer extension, we learned that the NetBSD authors claimed one of those long words for another purpose. This conflict needs to be resolved, probably by enlarging the structure.

CONCLUSION

We have described a new implementation of the NetBSD timer facilities that appears to be more scalable, robust, and flexible than the existing implementation. It is scalable (see Figure 8) in that the overhead to start, stop, or maintain timers does not depend on the number of outstanding timers. It is robust in that we can precisely bound the amount of time interrupts are locked out in terms of a parameter, `MAX_SOFTCLOCK_STEPS`. It is flexible in that user processes are allowed to have multiple outstanding timers. The new implementation is fully compatible with existing software because existing interfaces are supported. However, applications that require slightly better performance (handles for deleting callouts) or flexibility (more than one outstanding timer) must use the new interfaces. The implementation does not incur any extra cost for these new features, and the code expansion is small (468 lines total). The software is available at: http://www.cs.berkeley.edu/ amc/research/timer/

acknowledgements

We wish to thank Ron Minnich and Chuck Cranor for their helpful comments on this paper.

REFERENCES

1. D. D. Clark, V. Jacobson, J. Romkey and H. Salwen, ‘An analysis of TCP processing overhead’, *IEEE Communications* **27**(6), 23–29 (1989).
2. C. Thekkath, T. Nguyen, E. Moy and E. Lazowska, ‘Implementing network protocols at user level’, *IEEE Transactions on Networking*, **1**(5), 554–564 (1993).
3. G. Davison, ‘Calendar p’s and q’s’, *Communications of the ACM*, **32**(10), 1241–1242 (1989).
4. G. Varghese and A. Lauck, ‘Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility’, *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987, pp. 171–180.
5. R. Brown, ‘Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem’, *Communications of the ACM*, **31**(10), 1220–1227 (1988).