```go
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// Time-related runtime and pieces of package time.

package runtime

import (
    "internal/abi"
    "runtime/internal/atomic"
    "runtime/internal/sys"
    "unsafe"
)

// A timer is a potentially repeating trigger for calling t.f(t.arg, t.seq).
// Timers are allocated by client code, often as part of other data structures.
// Each P has a heap of pointers to timers that it manages.
//
// A timer is expected to be used by only one client goroutine at a time,
// but there will be concurrent access by the P managing that timer.
// The fundamental state about the timer is managed in the atomic state field,
// including a lock bit to manage access to the other fields.
// The lock bit supports a manual cas-based spin lock that handles
// contention by yielding the OS thread. The expectation is that critical
// sections are very short and contention on the lock bit is low.
//
// Package time knows the layout of this structure.
// If this struct changes, adjust ../time/sleep.go:/runtimeTimer.
type timer struct {
    ts *timers

    // Timer wakes up at when, and then at when+period, ... (period > 0 only)
    // each time calling f(arg, now) in the timer goroutine, so f must be
    // a well-behaved function and not block.
    //
    // when must be positive on an active timer.
    // Timers in heaps are ordered by when.
    when   int64
    period int64
    f      func(any, uintptr, int64)
    arg    any
    seq    uintptr

    // The state field holds state bits, defined below.
    state atomic.Uintptr

    // nextWhen is the next value for when,
    // set if state&timerNextWhen is true.
    // In that case, the actual update of when = nextWhen
    // must be delayed until the heap can be fixed at the same time.
    nextWhen int64
}

// A timers is a per-P set of timers.
type timers struct {
    // lock protects timers; timers are per-P, but the scheduler can
    // access the timers of another P, so we have to lock.
    lock mutex

    // heap is the set of timers, ordered by t.when.
    // Must hold lock to access.
    heap []*timer

    // len is an atomic copy of len(heap).
    len atomic.Uint32
```



```go
    // zombies is the number of timers in the heap
    // that are marked for removal.
    zombies atomic.Uint32

    // raceCtx is the race context used while executing timer functions.
    raceCtx uintptr

    // timer0When is an atomic copy of of heap[0].when.
    // If len(heap) == 0, timer0When is 0.
    timer0When atomic.Int64

    // timerModifiedEarliest holds the earliest known heap[i].nextWhen field
    // for the heap entries with a new nextWhen pending
    // (that is, with the timerNextWhen bit set in t.state).
    // Because timers can be modified multiple times,
    // timerModifiedEarliest can be set to a nextWhen that has since
    // been replaced with a later time.
    // If this is 0, it means there are no timerNextWhen timers in the heap.
    timerModifiedEarliest atomic.Int64
}

// Timer state field.
const (
    // timerLocked is set when the timer is locked,
    // meaning other goroutines cannot read or write mutable fields.
    // Goroutines can still read the state word atomically to see
    // what the state was before it was locked.
    // The lock is implemented as a cas on the state field with osyield on conten
    // the expectation is very short critical sections with little to no contenti
    timerLocked = 1 << iota

    // timerHeaped is set when the timer is stored in some P's heap.
    timerHeaped

    // timerNextWhen is set when a pending change to the timer's when
    // field has been stored in t.nextwhen. The change to t.when waits
    // until the heap in which the timer appears can also be updated.
    // Only set when timerHeaped is also set.
    timerNextWhen

    // timerZombie is set when the timer has been stopped
    // but is still present in some P's heap.
    // Only set when timerHeaped is also set.
    // It is possible for timerNextWhen and timerZombie to both
    // be set, meaning that the timer was modified and then stopped.
    // A timer sending to a channel may be placed in timerZombie
    // to take it out of the heap even though the timer is not stopped,
    // as long as nothing is reading from the channel.
    timerZombie

    // timerChan is set when a timer is recognized as only existing
    // to send to a channel. We take the timer out of the heap when
    // nothing is watching the channel, so that the channel and
    // the timer can be garbage collected if they become unreferenced,
    // even if the timer is still pending.
    timerChan

    // timerBlocked is a value added repeatedly to the state, once per
    // goroutine blocked on a timerChan timer.
    // (That is, the number of blocked goroutines is state/timerBlocked.)
    // Must be last, since it is not just a single bit.
    timerBlocked
)

// lock locks the timer, allowing reading or writing any of the timer fields.
```

```go
// It returns the current m and the status prior to the lock.
// The caller must call unlock with the same m and an updated status.
func (t *timer) lock() (state uintptr, mp *m) {
    acquireLockRank(lockRankTimer)
    for {
        state := t.state.Load()
        if state&timerLocked != 0 {
            osyield()
            continue
        }
        // Prevent preemption while the timer is locked.
        // This could lead to a self-deadlock. See #38070.
        mp := acquirem()
        if t.state.CompareAndSwap(state, state|timerLocked) {
            return state, mp
        }
        releasem(mp)
    }
}

// unlock unlocks the timer.
// If mp == nil, the caller is responsible for calling
// releasem(mp) with the mp returned by t.lock.
func (t *timer) unlock(state uintptr, mp *m) {
    releaseLockRank(lockRankTimer)
    if t.state.Load()&timerLocked == 0 {
        badTimer()
    }
    if state&timerLocked != 0 {
        badTimer()
    }
    t.state.Store(state)
    if mp != nil {
        releasem(mp)
    }
}

// initChan checks to see if a timer exists to feed a channel.
// If so, it sets the timerChan bit in the state and also records
// the timer in the channel's c.timer field.
// initChan returns the updated state, to be passed to t.unlock.
func (t *timer) initChan(state uintptr) uintptr {
    if state&timerChan == 0 && t.arg != nil {
        if e := efaceOf(&t.arg); e._type.Kind() == kindChan {
            state |= timerChan
            t.hchan(state).timer = t
        }
    }
    return state
}

// hchan returns the channel associated with the timer t.
// It must only be called for timerChan timers.
func (t *timer) hchan(state uintptr) *hchan {
    if state&timerChan == 0 {
        badTimer()
    }
    return (*hchan)(efaceOf(&t.arg).data)
}

// updateHeap updates t.when as directed by state, returning the new state
// and a bool indicating whether the state (and t.when) changed.
// If ts != nil, then t must be ts.heap[0], and updateHeap takes care of
// moving t within the timers heap to preserve the heap invariants.
// If ts == nil, then t must not be in a heap (or is in a heap that is
// temporarily not maintaining its invariant, such as during timers.adjust).
```



```go
func (t *timer) updateHeap(state uintptr, ts *timers) (newState uintptr, updated
    if state&timerZombie != 0 {
        // Take timer out of heap, applying final t.when update first.
        state &^= timerHeaped | timerZombie
        if state&timerNextWhen != 0 {
            state &^= timerNextWhen
            t.when = t.nextWhen
        }
        if ts != nil {
            if t != ts.heap[0] {
                badTimer()
            }
            ts.zombies.Add(-1)
            ts.deleteMin()
        }
        return state, true
    }

    if state&timerNextWhen != 0 {
        // Apply t.when update and move within heap.
        state &^= timerNextWhen
        t.when = t.nextWhen
        // Move t to the right position.
        if ts != nil {
            if t != ts.heap[0] {
                badTimer()
            }
            ts.siftDown(0)
            ts.updateTimer0When()
        }
        return state, true
    }

    return state, false
}

// maxWhen is the maximum value for timer's when field.
const maxWhen = 1<<63 - 1

// verifyTimers can be set to true to add debugging checks that the
// timer heaps are valid.
const verifyTimers = false

// Package time APIs.
// Godoc uses the comments in package time, not these.

// time.now is implemented in assembly.

// timeSleep puts the current goroutine to sleep for at least ns nanoseconds.
//
//go:linkname timeSleep time.Sleep
func timeSleep(ns int64) {
    if ns <= 0 {
        return
    }

    gp := getg()
    t := gp.timer
    if t == nil {
        t = new(timer)
        gp.timer = t
    }
    t.f = goroutineReady
    t.arg = gp
    t.nextWhen = nanotime() + ns
    if t.nextWhen < 0 { // check for overflow.
```

```go
        t.nextWhen = maxWhen
    }
    gopark(resetForSleep, unsafe.Pointer(t), waitReasonSleep, traceBlockSleep, 1)
}

// resetForSleep is called after the goroutine is parked for timeSleep.
// We can't call resettimer in timeSleep itself because if this is a short
// sleep and there are many goroutines then the P can wind up running the
// timer function, goroutineReady, before the goroutine has been parked.
func resetForSleep(gp *g, ut unsafe.Pointer) bool {
    t := (*timer)(ut)
    t.reset(t.nextWhen)
    return true
}

// startTimer adds t to the timer heap.
//
//go:linkname startTimer time.startTimer
func startTimer(t *timer) {
    if raceenabled {
        racerelease(unsafe.Pointer(t))
    }
    if t.state.Load() != 0 {
        throw("startTimer called with initialized timer")
    }
    t.reset(t.when)
}

// stopTimer stops a timer.
// It reports whether t was stopped before being run.
//
//go:linkname stopTimer time.stopTimer
func stopTimer(t *timer) bool {
    return t.stop()
}

// resetTimer resets an inactive timer, adding it to the heap.
//
// Reports whether the timer was modified before it was run.
//
//go:linkname resetTimer time.resetTimer
func resetTimer(t *timer, when int64) bool {
    if raceenabled {
        racerelease(unsafe.Pointer(t))
    }
    return t.reset(when)
}

// modTimer modifies an existing timer.
//
//go:linkname modTimer time.modTimer
func modTimer(t *timer, when, period int64) {
    t.modify(when, period, t.f, t.arg, t.seq)
}

// Go runtime.

// Ready the goroutine arg.
func goroutineReady(arg any, _ uintptr, _ int64) {
    goready(arg.(*g), 0)
}

// add adds t to the timers.
// The caller must have set t.ts = t, unlocked t,
// and then locked ts.lock.
func (ts *timers) add(t *timer) {
```



```go
    // Timers rely on the network poller, so make sure the poller
    // has started.
    if netpollInited.Load() == 0 {
        netpollGenericInit()
    }

    if t.ts != ts {
        throw("timers.add: ts not set in timer")
    }
    ts.heap = append(ts.heap, t)
    ts.siftUp(len(ts.heap) - 1)
    if t == ts.heap[0] {
        ts.updateTimer0When()
    }
    ts.len.Store(uint32(len(ts.heap)))
}

// stop stops the timer t. It may be on some other P, so we can't
// actually remove it from the timers heap. We can only mark it as stopped.
// It will be removed in due course by the P whose heap it is on.
// Reports whether the timer was stopped before it was run.
func (t *timer) stop() bool {
    state, mp := t.lock()
Redo:
    pending := false
    switch {
    case state&timerHeaped != 0:
        // Timer is in some heap, but is possibly already stopped
        // (indicated by a nextWhen update to 0).
        if state&timerNextWhen == 0 || t.nextWhen > 0 {
            // Timer pending: stop it.
            t.nextWhen = 0
            state |= timerNextWhen
            pending = true
        }
        // Mark timer for removal unless already marked.
        // (A timerChan timer might be marked for removal but not yet stopped.)
        if state&timerZombie == 0 {
            state |= timerZombie
            t.ts.zombies.Add(1)
        }

    case state&timerChan != 0 && t.when != 0:
        // Active timer attached to channel but not in heap, because
        // nothing is waiting on the channel or timer is stopped.
        // If it should have triggered already (but nothing looked yet),
        // trigger now, so that a receive after the stop sees the "old"
        // value that should be there.
        if state >= timerBlocked { // state&timerHeaped == 0
            badTimer()
        }
        if now := nanotime(); t.when <= now {
            systemstack(func() {
                t.unlockAndRun(now, state, mp) // resets t.when
            })
            state, mp = t.lock()
            if state&timerHeaped != 0 {
                // While it was unlocked to run the channel send,
                // the timer moved into the heap.
                // Behave as though the send happened long ago
                // and stop was just called now.
                goto Redo
            }
        }
        pending = t.when > 0
        t.when = 0
```

```
    }

    t.unlock(state, mp)
    return pending
}

// deleteMin removes timer 0 from ts.
// ts must be locked.
func (ts *timers) deleteMin() {
    t := ts.heap[0]
    if t.ts != ts {
        throw("deleteMin: wrong timers")
    }
    t.ts = nil
    last := len(ts.heap) - 1
    if last > 0 {
        ts.heap[0] = ts.heap[last]
    }
    ts.heap[last] = nil
    ts.heap = ts.heap[:last]
    if last > 0 {
        ts.siftDown(0)
    }
    ts.updateTimer0When()
    ts.len.Store(uint32(last))
    if last == 0 {
        // If there are no timers, then clearly none are modified.
        ts.timerModifiedEarliest.Store(0)
    }
}

// modify modifies an existing timer.
// This is called by the netpoll code or time.Ticker.Reset or time.Timer.Reset.
// Reports whether the timer was modified before it was run.
func (t *timer) modify(when, period int64, f func(any, uintptr, int64), arg any,
    if when <= 0 {
        throw("timer when must be positive")
    }
    if period < 0 {
        throw("timer period must be non-negative")
    }

    state, mp := t.lock()
Redo:
    t.period = period
    t.f = f
    t.arg = arg
    t.seq = seq

    if state&timerHeaped == 0 {
        // Timer not in any heap, so either stopped/new
        // or a timer for a currently unused channel.
        // If this is a timer for a channel, initialize but leave out of heap,
        // so that GC can collect it. The channel code will add the timer
        // to the heap as needed to serve blocked channel ops.
        // See enqueueTimerChan, dequeueTimerChan.
        if state = t.initChan(state); state&timerChan != 0 && state < timerBlocke
            pending := false
            if t.when != 0 {
                if now := nanotime(); t.when <= now {
                    systemstack(func() {
                        t.unlockAndRun(now, state, mp) // resets t.when
                    })
                    state, mp = t.lock()
                    if state&timerHeaped != 0 {
                        // While it as unlocked to run the channel send,
```



```
                        // the timer moved into the heap. Behave as though
                        // the channel send happened long ago and the
                        // modify call just started at this instant.
                        goto Redo
                    }
                }
            }
            pending = t.when > 0
            t.when = when
            t.unlock(state, mp)
            return pending
        }

        // Not a timer for a channel, so needs to go into heap.
        // Assigning to when is permitted because the timer
        // is not in any heap, so the assignment cannot
        // break heap invariants.
        t.when = when
        t.unlockAndQueue(state, mp)
        return false
    }

    pending := true // in the heap

    if state&timerZombie != 0 {
        // In the heap but marked for removal.
        // Therefore not pending; unmark it.
        pending = false
        t.ts.zombies.Add(-1)
        state &^= timerZombie
    }

    // The timer is in some P's heap (perhaps another P),
    // so we can't change the when field.
    // If we did, the other P's heap would be out of order.
    // So we put the new when value in the nextWhen field
    // and set timerNextWhen, leaving the other P set the when
    // field when it is prepared to maintain the heap invariant.
    t.nextWhen = when
    state |= timerNextWhen
    earlier := when < t.when
    if earlier {
        t.ts.updateTimerModifiedEarliest(when)
    }
    t.unlock(state, mp)

    // If the new status is earlier, wake up the poller.
    if earlier {
        wakeNetPoller(when)
    }

    return pending
}

// unlockAndQueue unlocks the timer and adds it to the
// local P's timer heap.
func (t *timer) unlockAndQueue(state uintptr, mp *m) {
    // Set up t for insertion but unlock first,
    // to avoid lock inversion with timers lock.
    // We set t.ts = ts so that any other concurrent
    // updates to t after the unlock update the
    // various atomic state in ts correctly,
    // as if t were already in ts.
    ts := &getg().m.p.ptr().timers
    state |= timerHeaped
    t.ts = ts
```

```
	when := t.when
	t.unlock(state, nil)

	lock(&ts.lock)
	ts.add(t)
	unlock(&ts.lock)
	releasem(mp)
	wakeNetPoller(when)
}

// reset resets the time when a timer should fire.
// If used for an inactive timer, the timer will become active.
// This should be called instead of addtimer if the timer value has been,
// or may have been, used previously.
// Reports whether the timer was active and was stopped.
func (t *timer) reset(when int64) bool {
	return t.modify(when, t.period, t.f, t.arg, t.seq)
}

// cleanHead cleans up the head of the timer queue. This speeds up
// programs that create and delete timers; leaving them in the heap
// slows down addtimer.
// The caller must have locked ts.
func (ts *timers) cleanHead() {
	gp := getg()
	for {
		if len(ts.heap) == 0 {
			return
		}

		// This loop can theoretically run for a while, and because
		// it is holding timersLock it cannot be preempted.
		// If someone is trying to preempt us, just return.
		// We can clean the timers later.
		if gp.preemptStop {
			return
		}

		t := ts.heap[0]
		if t.ts != ts {
			throw("timers.cleanHead: bad ts")
		}

		if t.state.Load()&(timerNextWhen|timerZombie) == 0 {
			// Fast path: head of timers does not need adjustment.
			return
		}

		state, mp := t.lock()
		state, updated := t.updateHeap(state, ts)
		t.unlock(state, mp)
		if !updated {
			// Head of timers does not need adjustment.
			return
		}
	}
}

// take moves any timers from src into ts
// and then clears the timer state from src,
// because src is being destroyed.
// The caller must not have locked either timers.
func (ts *timers) take(src *timers) {
	if len(src.heap) > 0 {
		// The world is stopped, but we acquire timersLock to
		// protect against sysmon calling timeSleepUntil.
```



```
		// This is the only case where we hold the timersLock of
		// more than one P, so there are no deadlock concerns.
		lock(&src.lock)
		lock(&ts.lock)
		ts.move(src.heap)
		src.heap = nil
		src.len.Store(0)
		src.zombies.Store(0)
		src.timer0When.Store(0)
		src.timerModifiedEarliest.Store(0)
		unlock(&ts.lock)
		unlock(&src.lock)
	}
}

// moveTimers moves a slice of timers to pp. The slice has been taken
// from a different P.
// This is currently called when the world is stopped, but the caller
// is expected to have locked the timers for pp.
func (ts *timers) move(timers []*timer) {
	for _, t := range timers {
		state, mp := t.lock()
		t.ts = nil
		state, _ = t.updateHeap(state, nil)
		// Unlock before add, to avoid append (allocation)
		// while holding lock. This would be correct even if the world wasn't
		// stopped (but it is), and it makes staticlockranking happy.
		if state&timerHeaped != 0 {
			t.ts = ts
		}
		t.unlock(state, mp)
		if state&timerHeaped != 0 {
			ts.add(t)
		}
	}
}

// adjust looks through the timers in the current P's heap for
// any timers that have been modified to run earlier, and puts them in
// the correct place in the heap. While looking for those timers,
// it also moves timers that have been modified to run later,
// and removes deleted timers. The caller must have locked the timers for pp.
func (ts *timers) adjust(now int64, force bool) {
	// If we haven't yet reached the time of the earliest timerModified
	// timer, don't do anything. This speeds up programs that adjust
	// a lot of timers back and forth if the timers rarely expire.
	// We'll postpone looking through all the adjusted timers until
	// one would actually expire.
	if !force {
		first := ts.timerModifiedEarliest.Load()
		if first == 0 || first > now {
			if verifyTimers {
				ts.verify()
			}
			return
		}
	}

	// We are going to clear all timerModified timers.
	ts.timerModifiedEarliest.Store(0)

	changed := false
	for i := 0; i < len(ts.heap); i++ {
		t := ts.heap[i]
		if t.ts != ts {
			throw("timers.adjust: bad ts")
```

```go
        }

        state, mp := t.lock()
        if state&timerHeaped == 0 {
            badTimer()
        }
        state, updated := t.updateHeap(state, nil)
        if updated {
            changed = true
            if state&timerHeaped == 0 {
                n := len(ts.heap)
                ts.heap[i] = ts.heap[n-1]
                ts.heap[n-1] = nil
                ts.heap = ts.heap[:n-1]
                t.ts = nil
                ts.zombies.Add(-1)
                i--
            }
        }
        t.unlock(state, mp)
    }

    if changed {
        ts.initHeap()
        ts.updateTimer0When()
    }

    if verifyTimers {
        ts.verify()
    }
}

// wakeTime looks at ts's timers and returns the time when we
// should wake up the netpoller. It returns 0 if there are no timers.
// This function is invoked when dropping a P, so it must run without
// any write barriers.
//
//go:nowritebarrierrec
func (ts *timers) wakeTime() int64 {
    next := ts.timer0When.Load()
    nextAdj := ts.timerModifiedEarliest.Load()
    if next == 0 || (nextAdj != 0 && nextAdj < next) {
        next = nextAdj
    }
    return next
}

// check runs any timers for the P that are ready.
// If now is not 0 it is the current time.
// It returns the passed time or the current time if now was passed as 0.
// and the time when the next timer should run or 0 if there is no next timer,
// and reports whether it ran any timers.
// If the time when the next timer should run is not 0,
// it is always larger than the returned time.
// We pass now in and out to avoid extra calls of nanotime.
//
//go:yeswritebarrierrec
func (ts *timers) check(now int64) (rnow, pollUntil int64, ran bool) {
    // If it's not yet time for the first timer, or the first adjusted
    // timer, then there is nothing to do.
    next := ts.wakeTime()
    if next == 0 {
        // No timers to run or adjust.
        return now, 0, false
    }
```



```go
    if now == 0 {
        now = nanotime()
    }

    // If this is the local P, and there are a lot of deleted timers,
    // clear them out. We only do this for the local P to reduce
    // lock contention on timersLock.
    force := ts == &getg().m.p.ptr().timers && int(ts.zombies.Load()) > int(ts.le

    if now < next && !force {
        // Next timer is not ready to run, and we don't need to clear deleted tim
        return now, next, false
    }

    lock(&ts.lock)
    if len(ts.heap) > 0 {
        ts.adjust(now, force)
        for len(ts.heap) > 0 {
            // Note that runtimer may temporarily unlock
            // pp.timersLock.
            if tw := ts.run(now); tw != 0 {
                if tw > 0 {
                    pollUntil = tw
                }
                break
            }
            ran = true
        }
    }

    unlock(&ts.lock)

    return now, pollUntil, ran
}

// run examines the first timer in timers. If it is ready based on now,
// it runs the timer and removes or updates it.
// Returns 0 if it ran a timer, -1 if there are no more timers, or the time
// when the first timer should run.
// The caller must have locked the timers for pp.
// If a timer is run, this will temporarily unlock the timers.
//
//go:systemstack
func (ts *timers) run(now int64) int64 {
Redo:
    if len(ts.heap) == 0 {
        return -1
    }
    t := ts.heap[0]
    if t.ts != ts {
        throw("timers.run: bad ts")
    }

    if t.state.Load()&(timerNextWhen|timerZombie) == 0 && t.when > now {
        // Fast path: not ready to run.
        // The access of t.when is protected by the caller holding
        // pp.timersLock, even though t itself is unlocked.
        return t.when
    }

    state, mp := t.lock()
    state, updated := t.updateHeap(state, ts)
    if updated {
        t.unlock(state, mp)
        goto Redo
    }
```

```
	if state&timerHeaped == 0 {
		badTimer()
	}

	if t.when > now {
		// Not ready to run.
		t.unlock(state, mp)
		return t.when
	}

	t.unlockAndRun(now, state, mp)
	return 0
}

// unlockAndRun unlocks and runs the timer t.
// If t is in a timer set (t.ts != nil), the caller must have locked the timer se
// and this call will temporarily unlock the timer set while running the timer fu
//
//go:systemstack
func (t *timer) unlockAndRun(now int64, state uintptr, mp *m) {
	if raceenabled {
		// Note that we are running on a system stack,
		// so there is no chance of getg().m being reassigned
		// out from under us while this function executes.
		tsLocal := &getg().m.p.ptr().timers
		if tsLocal.raceCtx == 0 {
			tsLocal.raceCtx = racegostart(abi.FuncPCABIInternal((*timers).run) +
		}
		raceacquirectx(tsLocal.raceCtx, unsafe.Pointer(t))
	}

	if state&(timerNextWhen|timerZombie) != 0 {
		badTimer()
	}

	f := t.f
	arg := t.arg
	seq := t.seq
	var next int64
	delay := now - t.when
	if t.period > 0 {
		// Leave in heap but adjust next time to fire.
		next = t.when + t.period*(1+delay/t.period)
		if next < 0 { // check for overflow.
			next = maxWhen
		}
	} else {
		next = 0
	}
	if state&timerHeaped != 0 {
		t.nextWhen = next
		state |= timerNextWhen
		if next == 0 {
			state |= timerZombie
		}
	} else {
		t.when = next
	}
	ts := t.ts
	state, _ = t.updateHeap(state, ts)
	t.unlock(state, mp)

	if raceenabled {
		// Temporarily use the current P's racectx for g0.
		gp := getg()
```



```
		if gp.racectx != 0 {
			throw("timers.run: unexpected racectx")
		}
		gp.racectx = gp.m.p.ptr().timers.raceCtx
	}

	if ts != nil {
		unlock(&ts.lock)
	}
	f(arg, seq, delay)
	if ts != nil {
		lock(&ts.lock)
	}

	if raceenabled {
		gp := getg()
		gp.racectx = 0
	}
}

// updateTimerPMask clears pp's timer mask if it has no timers on its heap.
//
// Ideally, the timer mask would be kept immediately consistent on any timer
// operations. Unfortunately, updating a shared global data structure in the
// timer hot path adds too much overhead in applications frequently switching
// between no timers and some timers.
//
// As a compromise, the timer mask is updated only on pidleget / pidleput. A
// running P (returned by pidleget) may add a timer at any time, so its mask
// must be set. An idle P (passed to pidleput) cannot add new timers while
// idle, so if it has no timers at that time, its mask may be cleared.
//
// Thus, we get the following effects on timer-stealing in findrunnable:
//
//   - Idle Ps with no timers when they go idle are never checked in findrunnable
//     (for work- or timer-stealing; this is the ideal case).
//   - Running Ps must always be checked.
//   - Idle Ps whose timers are stolen must continue to be checked until they run
//     again, even after timer expiration.
//
// When the P starts running again, the mask should be set, as a timer may be
// added at any time.
//
// TODO(prattmic): Additional targeted updates may improve the above cases.
// e.g., updating the mask when stealing a timer.
func updateTimerPMask(pp *p) {
	if pp.timers.len.Load() > 0 {
		return
	}

	// Looks like there are no timers, however another P may transiently
	// decrement numTimers when handling a timerModified timer in
	// checkTimers. We must take timersLock to serialize with these changes.
	lock(&pp.timers.lock)
	if pp.timers.len.Load() == 0 {
		timerpMask.clear(pp.id)
	}
	unlock(&pp.timers.lock)
}

// verifyTimerHeap verifies that the timers is in a valid state.
// This is only for debugging, and is only called if verifyTimers is true.
// The caller must have locked the timers.
func (ts *timers) verify() {
	for i, t := range ts.heap {
		if i == 0 {
```

```
            // First timer has no parent.
            continue
        }

        // The heap is 4-ary. See siftupTimer and siftdownTimer.
        p := (i - 1) / 4
        if t.when < ts.heap[p].when {
            print("bad timer heap at ", i, ": ", p, ": ", ts.heap[p].when, ", ",
            throw("bad timer heap")
        }
    }
    if n := int(ts.len.Load()); len(ts.heap) != n {
        println("timer heap len", len(ts.heap), "!= atomic len", n)
        throw("bad timer heap len")
    }
}

// updateTimer0When sets the P's timer0When field.
// The caller must have locked the timers for pp.
func (ts *timers) updateTimer0When() {
    if len(ts.heap) == 0 {
        ts.timer0When.Store(0)
    } else {
        ts.timer0When.Store(ts.heap[0].when)
    }
}

// updateTimerModifiedEarliest updates the recorded nextwhen field of the
// earlier timerModifiedEarier value.
// The timers for pp will not be locked.
func (ts *timers) updateTimerModifiedEarliest(nextwhen int64) {
    // The low bit of timerModifiedEarliest tracks how the value was set.
    // Low bit 1 means it was set by updateTimerModifiedEarliest
    // (without holding ts.lock).
    nextwhen |= 1
    for {
        old := ts.timerModifiedEarliest.Load()
        if old != 0 && old < nextwhen {
            return
        }

        if ts.timerModifiedEarliest.CompareAndSwap(old, nextwhen) {
            return
        }
    }
}

// timeSleepUntil returns the time when the next timer should fire. Returns
// maxWhen if there are no timers.
// This is only called by sysmon and checkdead.
func timeSleepUntil() int64 {
    next := int64(maxWhen)

    // Prevent allp slice changes. This is like retake.
    lock(&allpLock)
    for _, pp := range allp {
        if pp == nil {
            // This can happen if procresize has grown
            // allp but not yet created new Ps.
            continue
        }

        if w := pp.timers.wakeTime(); w != 0 {
            next = min(next, w)
        }
    }
```



```
    unlock(&allpLock)

    return next
}

// Heap maintenance algorithms.
// These algorithms check for slice index errors manually.
// Slice index error can happen if the program is using racy
// access to timers. We don't want to panic here, because
// it will cause the program to crash with a mysterious
// "panic holding locks" message. Instead, we panic while not
// holding a lock.

// siftUp puts the timer at position i in the right place
// in the heap by moving it up toward the top of the heap.
func (ts *timers) siftUp(i int) {
    t := ts.heap
    if i >= len(t) {
        badTimer()
    }
    when := t[i].when
    if when <= 0 {
        badTimer()
    }
    tmp := t[i]
    for i > 0 {
        p := (i - 1) / 4 // parent
        if when >= t[p].when {
            break
        }
        t[i] = t[p]
        i = p
    }
    if tmp != t[i] {
        t[i] = tmp
    }
}

// siftDown puts the timer at position i in the right place
// in the heap by moving it down toward the bottom of the heap.
func (ts *timers) siftDown(i int) {
    t := ts.heap
    n := len(t)
    if i >= n {
        badTimer()
    }
    when := t[i].when
    if when <= 0 {
        badTimer()
    }
    tmp := t[i]
    for {
        c := i*4 + 1 // left child
        c3 := c + 2  // mid child
        if c >= n {
            break
        }
        w := t[c].when
        if c+1 < n && t[c+1].when < w {
            w = t[c+1].when
            c++
        }
        if c3 < n {
            w3 := t[c3].when
            if c3+1 < n && t[c3+1].when < w3 {
                w3 = t[c3+1].when
```

```go
				c3++
			}
			if w3 < w {
				w = w3
				c = c3
			}
		}
		if w >= when {
			break
		}
		t[i] = t[c]
		i = c
	}
	if tmp != t[i] {
		t[i] = tmp
	}
}

// initHeap reestablishes the heap order in the slice ts.heap.
// It takes O(n) time for n=len(ts.heap), not the O(n log n) of n repeated add op
func (ts *timers) initHeap() {
	// Last possible element that needs sifting down is parent of last element;
	// last element is len(t)-1; parent of last element is (len(t)-1-1)/4.
	if len(ts.heap) <= 1 {
		return
	}
	for i := (len(ts.heap) - 1 - 1) / 4; i >= 0; i-- {
		ts.siftDown(i)
	}
}

// badTimer is called if the timer data structures have been corrupted,
// presumably due to racy use by the program. We panic here rather than
// panicking due to invalid slice access while holding locks.
// See issue #25686.
func badTimer() {
	throw("timer data corruption")
}

// Timer channels.

// maybeRunChan checks whether the timer needs to run
// to send a value to its associated channel. If so, it does.
// The timer must not be locked.
func (t *timer) maybeRunChan() {
	if t.state.Load()&timerHeaped != 0 {
		// If the timer is in the heap, the ordinary timer code
		// is in charge of sending when appropriate.
		return
	}

	state, mp := t.lock()
	now := nanotime()
	if state&timerHeaped != 0 || t.when == 0 || t.when > now {
		// Timer in the heap, or not running at all, or not triggered.
		t.unlock(state, mp)
		return
	}
	systemstack(func() {
		t.unlockAndRun(now, state, mp)
	})
}

// enqueueTimerChan is called when a channel op has decided to block on c.
// The caller holds the channel lock for c and possibly other channels.
// enqueueTimerChan makes sure that c is in the timer heap,
```



```go
// adding it if needed.
func enqueueTimerChan(c *hchan) {
	t := c.timer
	state, mp := t.lock()
	if state&timerChan == 0 {
		state = t.initChan(state)
		if state&timerChan == 0 {
			badTimer()
		}
	}
	state += timerBlocked
	if state >= 2*timerBlocked {
		// Already blocked and therefore in heap if running.
		if t.when > 0 && state&timerHeaped == 0 {
			badTimer()
		}
		t.unlock(state, mp)
		return
	}

	if state&timerHeaped != 0 {
		// Already in heap, but if this the first enqueue after a recent dequeue,
		// it may be marked for removal. Unmark it if so, but don't unmark
		// if the removal is because the timer is not running at all.
		if state&timerNextWhen == 0 || t.nextWhen != 0 {
			state &^= timerZombie
			t.ts.zombies.Add(-1)
		}
		t.unlock(state, mp)
		return
	}

	if t.when == 0 {
		// Timer not running. Skip adding to heap.
		t.unlock(state, mp)
		return
	}

	// Not in heap, but timer is running. Need to add to heap now.
	t.unlockAndQueue(state, mp)
}

// dequeueTimerChan is called when a channel op that was blocked on c
// is no longer blocked. Every call to enqueueTimerChan must be paired with
// a call to dequeueTimerChan.
// The caller holds the channel lock for c and possibly other channels.
// dequeueTimerChan removes c from the timer heap when nothing is
// blocked on it anymore.
func dequeueTimerChan(c *hchan) {
	t := c.timer
	state, mp := t.lock()
	if state&timerChan == 0 || state < timerBlocked {
		badTimer()
	}
	state -= timerBlocked
	if state < timerBlocked && state&timerHeaped != 0 && state&timerZombie == 0 {
		// Last goroutine that was blocked on this timer.
		// Mark for removal from heap but do not clear t.when,
		// so that we know what time it is still meant to trigger.
		state |= timerZombie
		t.ts.zombies.Add(1)
	}
	t.unlock(state, mp)
}
```