# Time

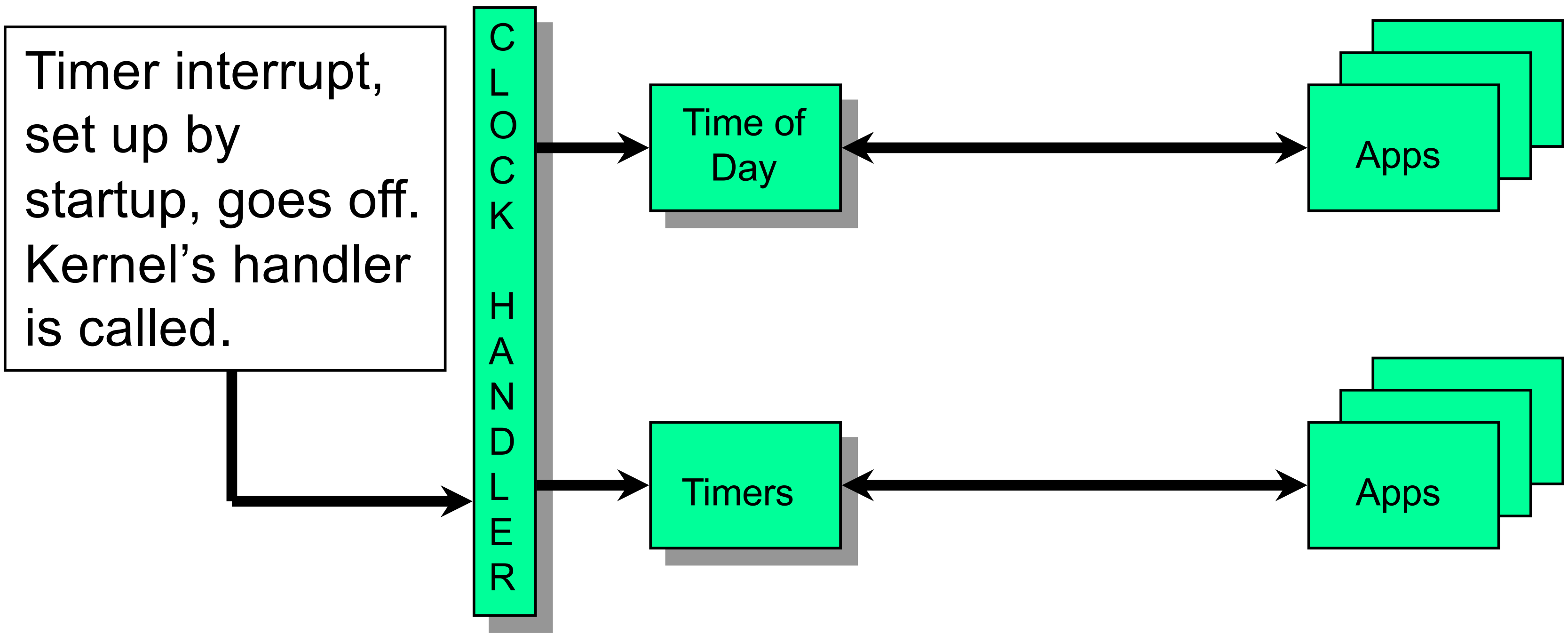
# You will learn how:

- QNX Neutrino handles time
- to read and update the system clock
- to use system timers and kernel timeouts

# Topics:

- → **Timing Architecture**
- **Getting and Setting the System Clock**
- **Timers**
- **Tolerant and High-Precision Timers**
- **Design Considerations**
- **Kernel Timeouts**
- **Conclusion**

# QNX® Neutrino®'s Concept of Time:



# Ticksize:

- on most systems, the default ticksize is 1ms
- this means most timing will be based on a resolution of 1ms
  - high-precision timers will allow for some timers to have a better resolution without decreasing ticksize

Note that the clock cannot usually be programmed for exactly 1ms in which case we round to the nearest available value that the clock can do (e.g. on legacy IBM PC hardware, you will actually get 0.999847ms).

# The time slice:

- is 4 times the ticksize so it defaults to 4ms
- the multiplier, 4, cannot be changed. If you change the ticksize then the time slice will also change

# Timing Examples

Let's examine how timing works by looking at some examples:

- we'll have two threads, t1 and t2, both READY at priority 10, both doing round-robin
- t2 will go to sleep for 10.5ms (a relative time)
- when t2 goes to sleep, the kernel figures out when to wake it up using this formula:

```
wake_up_time = now + requested_time + 1 tick
```

(where **now** is really the time as of the last tick)

# Timing Example 1

Keep in mind the following:

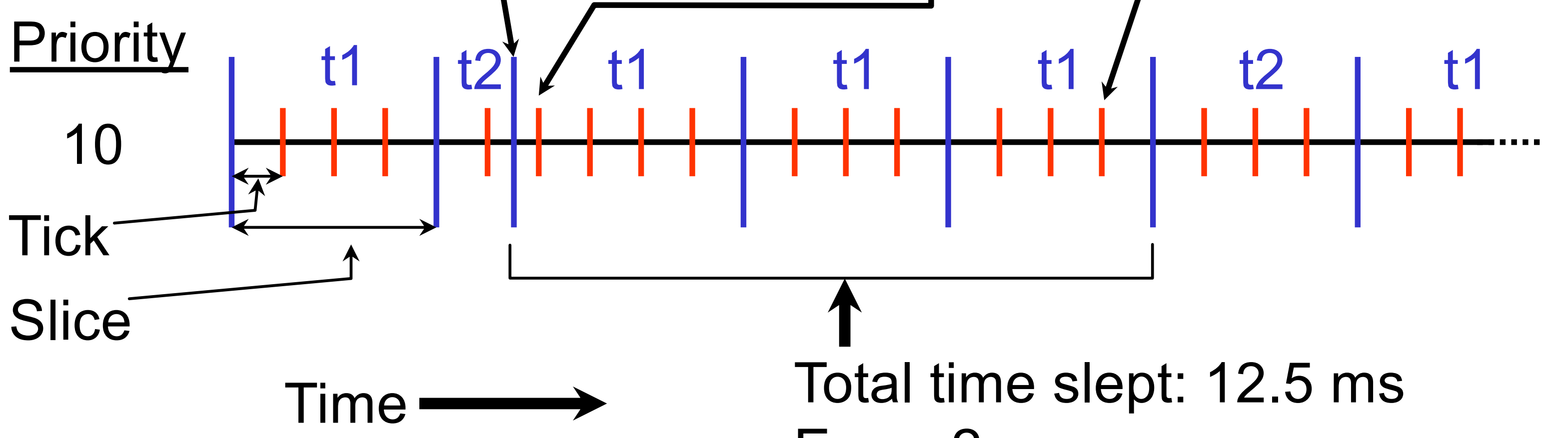
1. At every tick the kernel checks for expired timers.
2. If a thread's timer has expired then the kernel puts that thread at the end of the READY queue for that thread's priority.

For the expiry time, the kernel adds 10.5 ms to whatever time it will be here

t2 does:

```
t.tv_nsec = 10500000;
nanosleep(&t,NULL);
```

t2's timer has expired. t2 is put at the end of priority 10's READY queue, BUT t1 still runs




Total time slept: 12.5 ms

Error: 2 ms

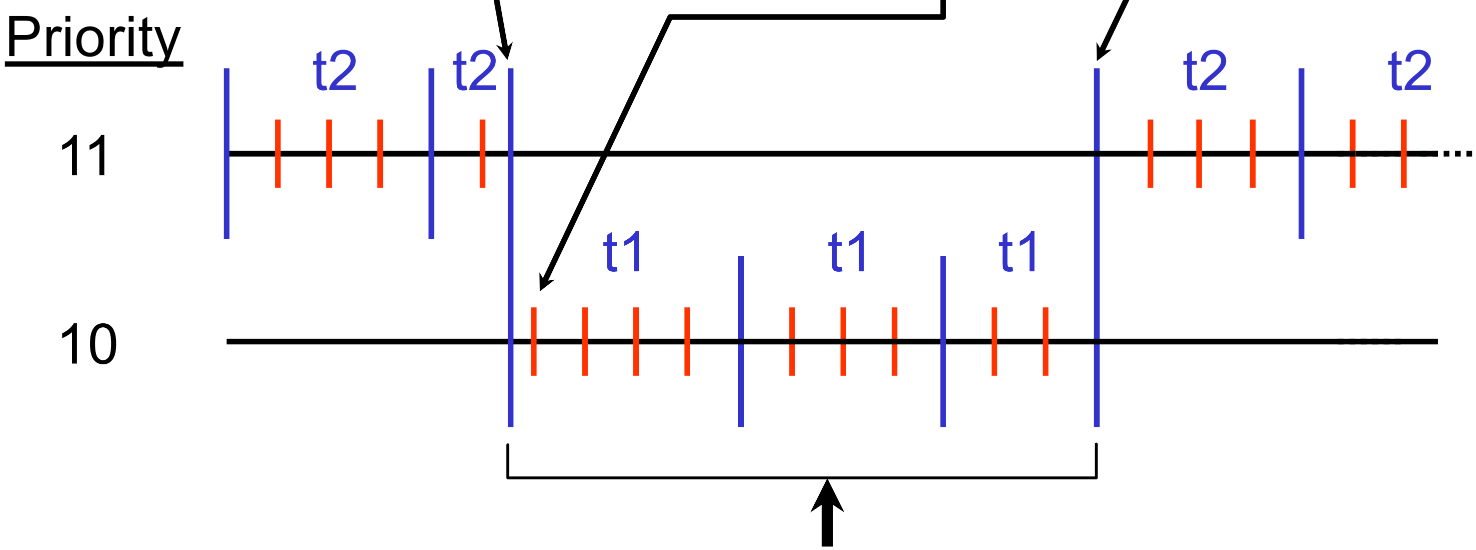
BUT, error <u>can be as bad as 5 ms</u>!

# Timing Example 2

For the expiry time, the kernel adds 10.5 ms to whatever time it will be here

t2 does:
t.tv_sec = 0; t.tv_nsec = 10500000;
**`nanosleep(&t);`**

t2's timer has expired. t2 is put at the end of priority 11's READY queue




Time

Total time slept: 11.5 ms
Error: 1 ms
Error can be as bad as 2 ms

Don't forget. A higher priority thread can preempt all of this and interrupt handlers can preempt even those.

# Ticksize

From code we can change the ticksize

– Let's change it to 100 us (.1 ms):

```
struct _clockperiod newval, oldval;

newval.nsec = 100*1000;         // 1e5 ns = .1 ms
newval.fract = 0;               // reserved, must be 0
ClockPeriod (CLOCK_REALTIME, &newval, &oldval, 0);
```

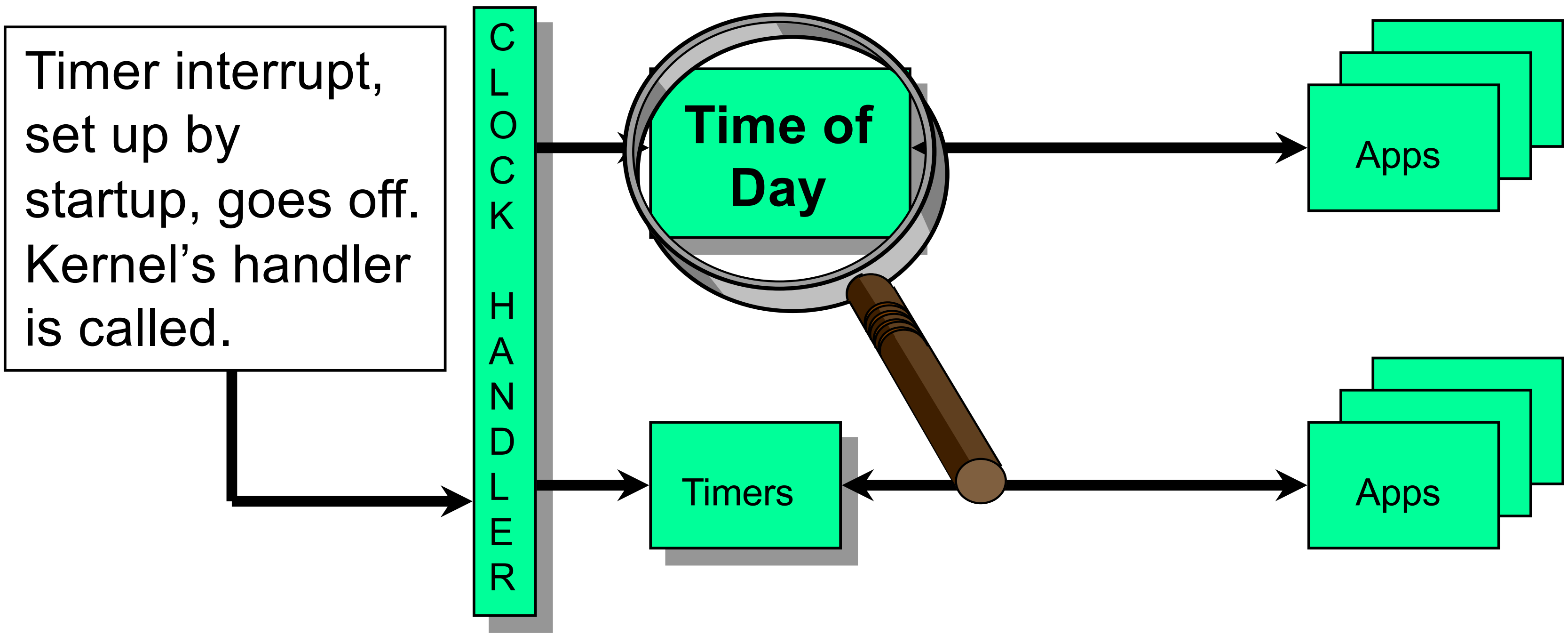
– the **`oldval`** parameter allows you to query the ticksize

☞ You must have **`PROCMGR_AID_CLOCKPERIOD`** to change the ticksize as it is a system wide setting.

# Topics:

- Timing Architecture
- → Getting and Setting the System Clock
- Timers
- Tolerant and High-Precision Timers
- Design Considerations
- Kernel Timeouts
- Conclusion

# Let's look at the time of day functions:



# QNX Neutrino time representation:

- internally stores time as 64-bit nanoseconds since 1970
  - this is actually stored as two 64-bit nanosecond values:
    - nanoseconds since boot
    - boot time since Jan 1, 1970
- POSIX functions uses a struct timespec
  - specified as seconds since Jan 1, 1970 and nanoseconds since last second
  - in 32-bit, QNX defines seconds as **`_Uint32t`**
  - in 64-bit, QNX defines seconds as **`_Int64t`**

# At bootup time:

- the kernel is given the current date and time from somewhere (battery backed up clock as on a PC, GPS, NTP, some atomic clock, ...)
- from then on, every tick, the kernel adds the ticksize to the current time
  - e.g. for a 1ms tick, every 1ms the kernel adds 1ms to the current time

# Read and/or Set the System Clock:

```
struct timespec tval;

clock_gettime( CLOCK_REALTIME, &tval );
tval.tv_sec += (60*60)*24L; /* add one day */
tval.tv_nsec = 0;
clock_settime( CLOCK_REALTIME, &tval );
```

☞ You must have **PROCMGR_AID_CLOCKSET** to change the system time.

# Clocks

## We can adjust the time:

– Bring it forward by one second:

```
struct _clockadjust new, old;

new.tick_nsec_inc = 10000; // 1e4 ns == 10 µs
new.tick_count = 100000;   // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

– Bring it backward by one second:

```
new.tick_nsec_inc = -10000;// 1e4 ns == 10 µs
new.tick_count = 100000;    // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

In both above examples the total adjustment will usually take 100 seconds (100k ticks (for a 1 msec tick size) = 100 seconds).

☞ As this changes the system time, it also requires **`PROCMGR_AID_CLOCKSET`**.

# QNX provides a free running counter:

```
uint64_t count;
count = ClockCycles ();
```

- It returns increments of the `cycles_per_sec` available from the system page (see below).
- On a system with supporting hardware we use it
  - e.g. `rdtsc` op code on x86 machines
  - these often increment at processor clock speed, e.g on a 500 MHz processor it returns increments of 2 ns (1/2 ns on 2 GHz, etc.)
- On a processor that does not have a free running counter, we fake it
  - these mostly don't get used anymore (e.g. 80486)

To find out how many cycles per second this clock is running at:

```
#include <sys/syspage.h>
uint64_t  cycles_per_sec;
cycles_per_sec = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
```

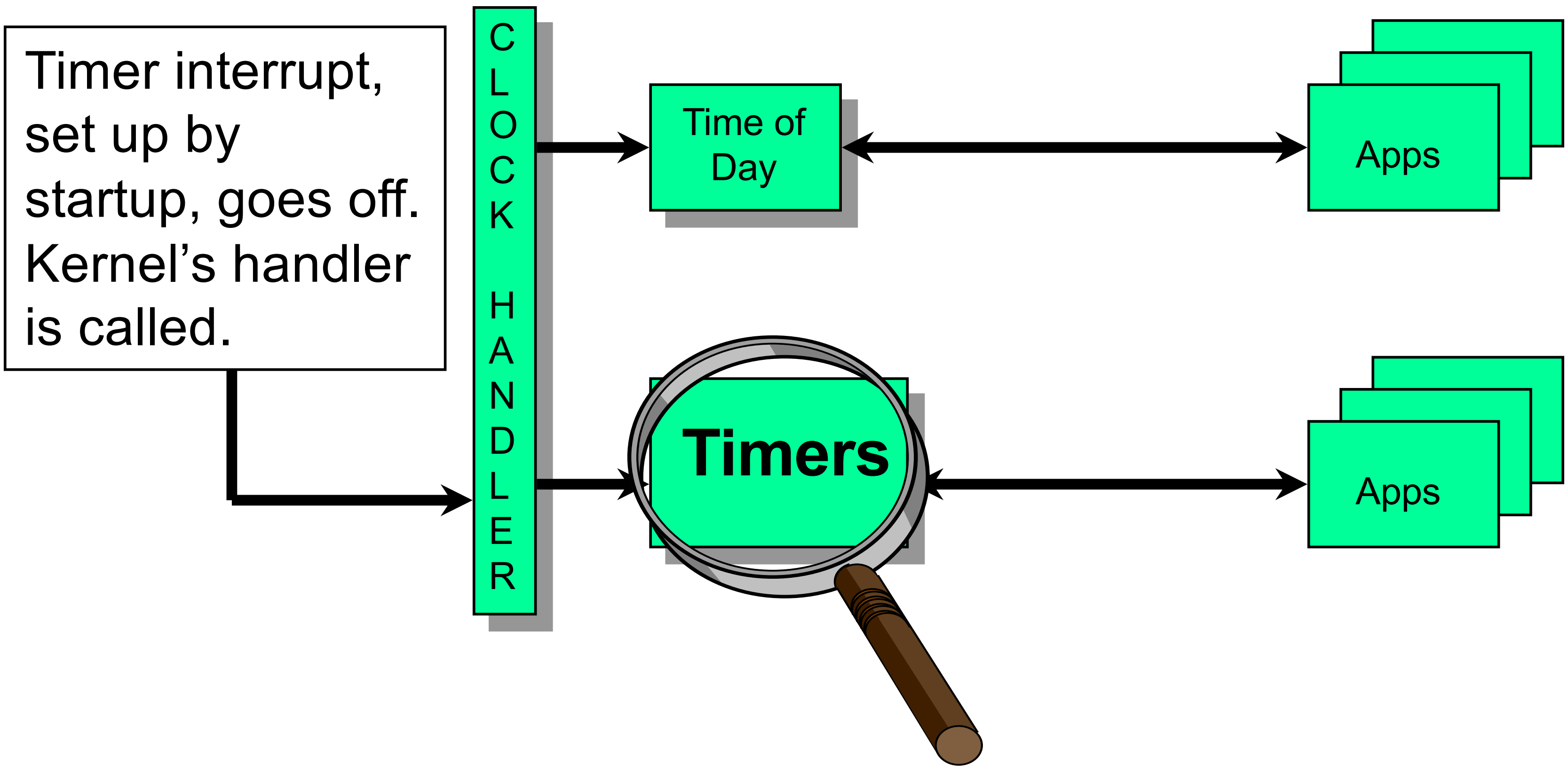
# Exercise:

- in your **`time`** project
- look at **`calctime1.c`**, and run it
  - what do the first set of delta values represent?
  - what would you do to figure out how long each iteration of the second **`for`** loop takes in microseconds?
- modify **`calctime2.c`** to adjust the ticksize to be ½ millisecond (500 us)

# Topics:

- Timing Architecture
- Getting and Setting the System Clock
- → Timers
- Tolerant and High-Precision Timers
- Design Considerations
- Kernel Timeouts
- Conclusion

# Let's look at timers:



## Setting a Timer

To set a timer, the process chooses:

- what kind of timer
  - periodic
  - one shot
- timer anchor
  - absolute
  - relative
- event to deliver upon trigger
  - fill in an EVENT structure

# POSIX timer functions:

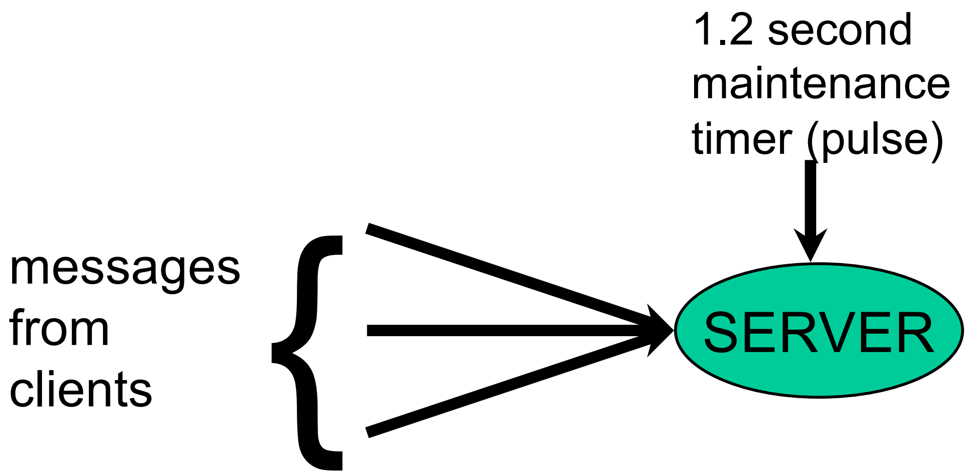
- administration

```
timer_create (clockID, &event, &timerID);
timer_delete (timerID);
```

- configuration

```
timer_settime (timerID, flags, &newtime, &oldtime);
```

# Timer example:




We want to have the server receive maintenance timer messages every 1.2 seconds, so that it can go out and perform housekeeping duties / integrity checks, etcetera.

*continued...*

# Timer example (continued):

```
#define TIMER_PULSE_CODE        (_PULSE_CODE_MINAVAIL+2)
struct  sigevent                sigevent;
struct  itimerspec              itime;
timer_t                         timerID;
int                             coid;

coid = ConnectAttach (..., chid, ...);
SIGEV_PULSE_INIT (&sigevent, coid, MAINTENANCE_PRIORITY,
    TIMER_PULSE_CODE, 0);
timer_create (CLOCK_REALTIME, &sigevent, &timerID);

itime.it_value.tv_sec = 1;
itime.it_value.tv_nsec = 500000000; // 500 million nsecs=.5 secs
itime.it_interval.tv_sec = 1;
itime.it_interval.tv_nsec = 200000000; // .2 secs
timer_settime (timerID, 0, &itime, NULL);
```

Fill in the sigevent to request a PULSE when the timer expires

Specify an expiry of 1.5 seconds

Repeating every 1.2 seconds thereafter

Relative, not absolute

*continued...*

# Timer example (continued):

```
typedef union {
   struct _pulse   pulse;
   // other message types you will receive
} myMessage_t;

myMessage_t   msg;
    ... // the setup code from the previous page goes here
    while (1) {
      rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
      if (rcvid == 0) {
        // it's a pulse, check what type...
        switch (msg.pulse.code) {
         case  TIMER_PULSE_CODE:
               periodic_maintenance();
               break;
        }
      }
    }
```

## Cancelling a Timer

You can cancel a timer without destroying it:

- useful if you will frequently be cancelling then restarting it
  - e.g. going into/coming out of low-power modes
- to cancel a timer:

```
struct itimerspec itime;
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = 0;
timer_settime (timeID, 0, &itime, NULL);
```

- to restart it simply fill in the timing and call *timer_settime()* again

# Exercise:

- in you **`time`** project is a file called **`reptimer.c`**
- when finished, it will wake up 5 seconds from the time it runs and then every 1500 milliseconds after that
  - it will wake up by receiving a pulse.
- all of the code is in *main()* for setting up the pulse event structure and for receiving the pulse
- however, the code for creating the timer and starting it ticking is missing
  - add it
- to test it do:

```
reptimer
```

- you should see a message displayed to the screen 5 seconds from the time you ran it, and then every 1500 milliseconds (1.5 seconds) after

# Topics:

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- → Tolerant and High-Precision Timers
- Design Considerations
- Kernel Timeouts
- Conclusion

# The kernel may operate in “tickless” mode:

- this is intended to help with managing power consumption
- to enter “tickless” mode:
  - it must be enabled by the `-Z` option to `startup-*`
  - no clock adjustment may be in process (*ClockAdjust()*)
  - all CPU cores must be idle
- in tickless mode, the kernel:
  - reprograms the timer chip to generate an interrupt when the next timer should expire
  - sets some internal flags
  - resets to regular mode as soon as any non-idle thread is scheduled

# Tolerant Timers

To help enable the kernel to enter “tickless” mode, or other low-power modes:

- a timer may have a tolerance value associated with it
- only affects the timer in “tickless” mode
- the tolerance is added to the regular expiry time to determine when the timer actually needs to next expire
- a `CLOCK_SOFTTIME` timer is the equivalent of a tolerant timer with an infinite tolerance

# To set the tolerance on a timer:

- make an **additional** call to *timer_settime()* or *TimerSettime()*
- pass **`TIMER_TOLERANCE`** in the flags field
- specify the tolerance in the **`itimerspec.it_value`** or **`_itimer.nsec`** fields, respectively
  - the tolerance value must be greater than the current ticksize
- if using *TimerSettime(),* an infinite tolerance may be specified by passing **`~0ULL`** in the **`nsec`** field

- a default tolerance for the timers in a process may be set with *procmgr_timer_tolerance()*

# Tolerant Timer example:

```
struct  sigevent                     sigevent;
struct  itimerspec                   itime;
struct _itimer                       ker_itime;
timer_t                              timerID;
coid = ConnectAttach (..., chid, ...);
...
timer_create (CLOCK_REALTIME, &sigevent, &timerID);

itime.it_value.tv_sec = 1;
itime.it_value.tv_nsec = 500000000; // 500 million nsecs=.5 secs
itime.it_interval.tv_sec = 1;
itime.it_interval.tv_nsec = 200000000; // .2 secs
timer_settime (timerID, 0, &itime, NULL);// set timer

ker_itime.nsec = 10 * 1000 * 1000 * 1000; // 10 second tolerance
TimerSettime( timerID, TIMER_TOLERANCE, &ker_itime, NULL );
```

# To make a timer high-precision:

– first make a call to *timer_settime()* or *TimerSettime()* with **`TIMER_TOLERANCE`** in the flags field
  - for high precision timers, set the timer tolerance prior to starting your timer
  - specify the resolution in the **`it_value.tv_nsec`** field
    – the value must fall between 0 and the ticksize
  - the kernel will adjust the hardware timer appropriately
    – doesn't change ticksize
    – extra calculations during the last timer interrupt before expiry

  ☞ will add jitter to all regular timers in the system

– then set the timer as per usual

☞ this is a privileged operation, requires **`PROCMGR_AID_HIGH_RESOLUTION_TIMER`** ability

# High-Precision Timer example:

– want  a timer that fires in 5.4ms, accurate to 20 usec

```
struct  itimerspec     timerTol, itime;
...
timer_create (CLOCK_REALTIME, &sigevent, &timerID);

memset( &timerTol, 0, sizeof timerTol );
timerTol.it_value.tv_nsec = 20 * 1000; // 20 usec resolution
timer_settime( timerID, TIMER_TOLERANCE, &timerTol, NULL );
  // Kernel will adjust HW timer as needed to give 20000 nsec
  // resolution for this timer

memset( &itime, 0, sizeof itime );
itime.it_value.tv_nsec = 5400 * 1000;
timer_settime ( timerID, 0, &itime, NULL ); // set timer
```

# Topics:

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- Tolerant and High-Precision Timers
- → Design Considerations
- Kernel Timeouts
- Conclusion

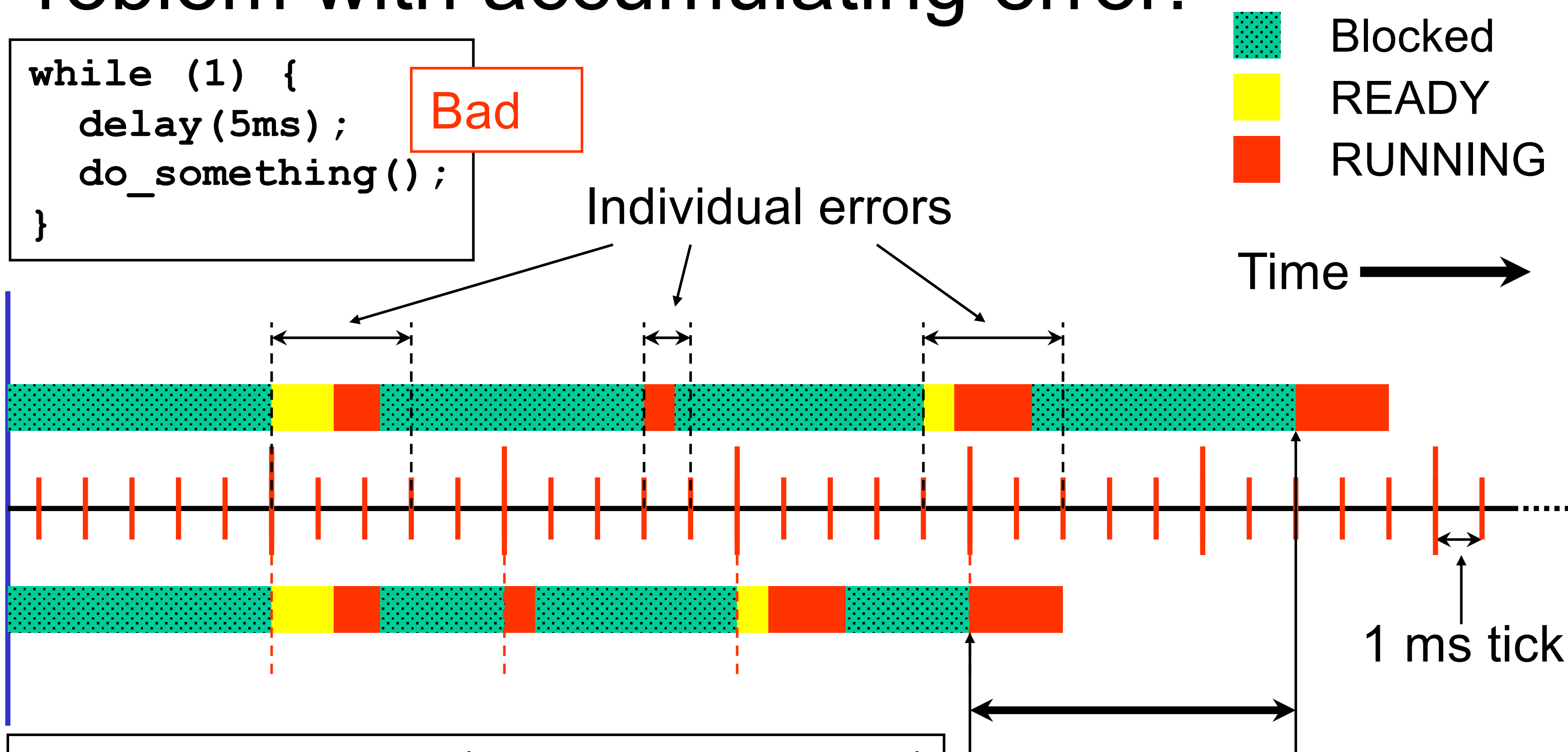
# Two timing design issues:

- how to run periodically without accumulating error
- timer frequency issues which can make timer expiry erratic

# Problem with accumulating error:

```
while (1) {
   delay(5ms);
   do_something();
}
```

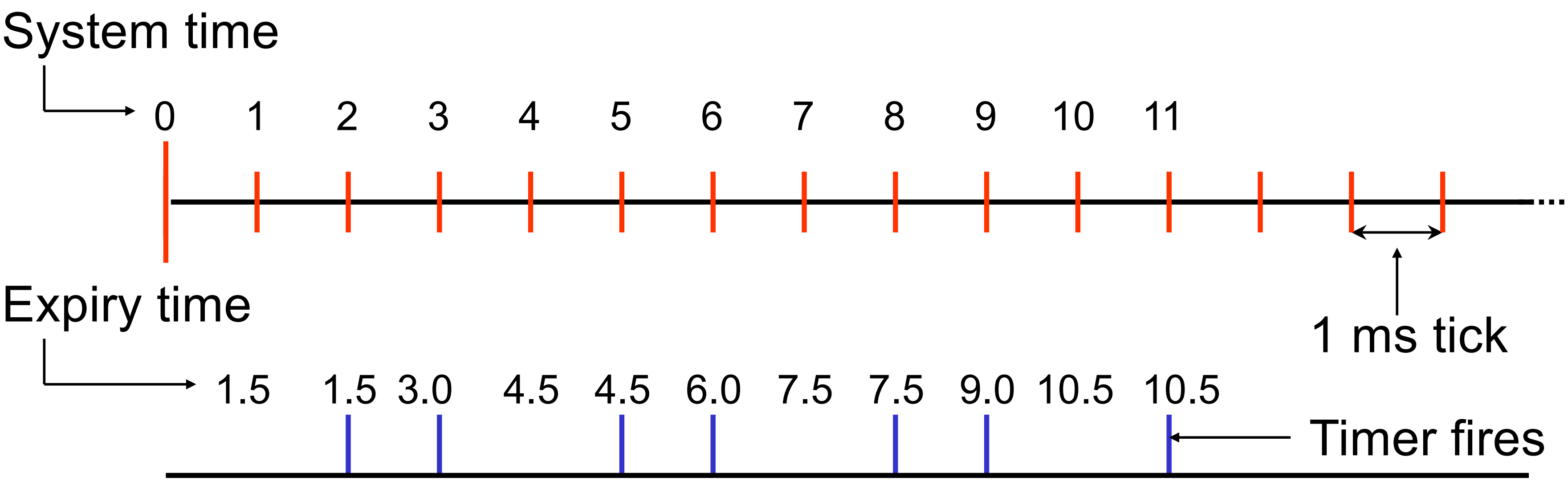
Bad




```
timer_settime(); /* repeating 5ms */
while(1) {
   wait_for_timer();
   do_something();
}
```

Good

# What happens if we ask for a repeating 1.5ms timer?



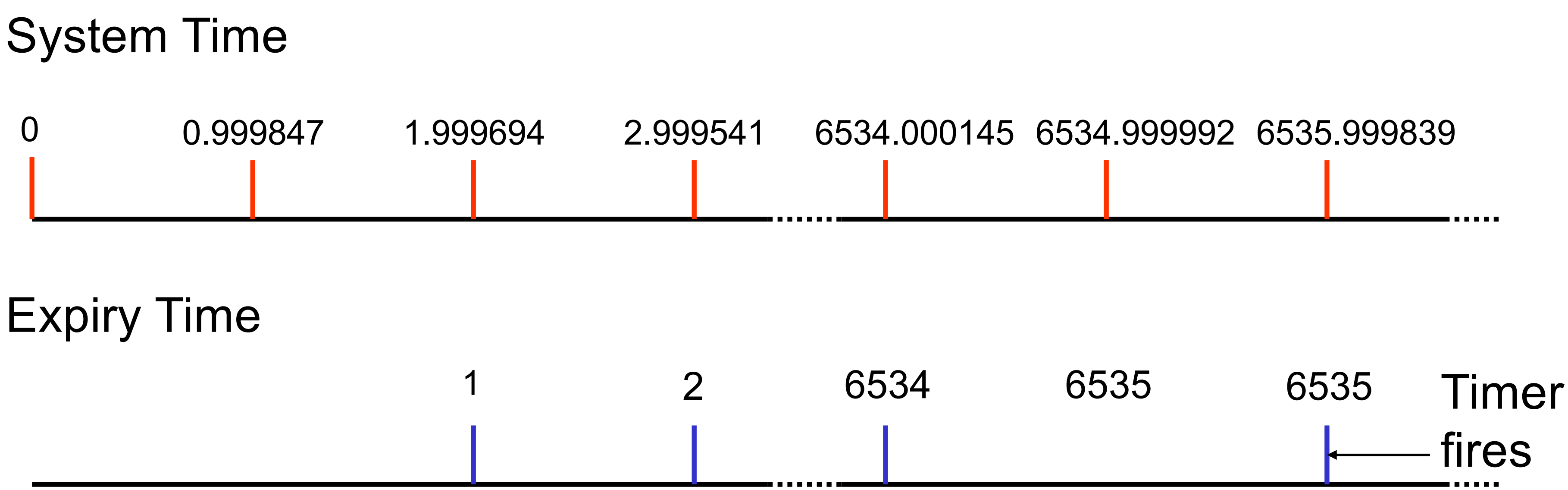

– we see an actual expiry pattern of : 2ms, 1ms, 2ms,1ms, 2ms,…

– the average is correct, and is the best our clock granularity can give.

## But, what if we ask for a repeating 1.0 ms timer...

A 1 ms ticksize is really the highest value less than 1ms which the hardware can do:

- e.g. on a legacy x86 it would be .999847 ms
- how would this behave?



# Timer Frequency Issues - What to do

So what do you do? There are some choices:

- use a high-precision timer
  - dangerous as it introduces jitter to everyone else
- make sure your tick size is quite a bit smaller than the smallest timer period you need
  - smaller ticksizes will impose more system overhead from the kernel handling the timer interrupt more frequently
- make your timer expiry an exact multiple of the actual tick size
  - use *ClockPeriod()* to get this value
  - use **`CLOCK_MONOTONIC`** so that time changes and *ClockAdjust()* don't affect your timing

# Topics:

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- Tolerant and High-Precision Timers
- Design Considerations
- → Kernel Timeouts
- Conclusion

# The kernel provides a timeout mechanism:

```
#define BILLION         1000000000LL
#define MILLION         1000000
struct sigevent         event;
uint64_t                timeout;

event.sigev_notify = SIGEV_UNBLOCK;

timeout = (2 * BILLION) + 500 * MILLION;

flags = _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY;

TimerTimeout (CLOCK_REALTIME, flags, &event, &timeout, NULL);
MsgSend (…);   // will time out in 2.5 seconds
```

Specify the event

Length of time (2.5 seconds)

Which blocking states

# Some notes on timeouts:

- timeout is relative to when *TimerTimeout()* is called
- the timeout is automatically cancelled when the next kernel call returns
  - therefore you should not do anything else between the call to *TimerTimeout()* and the function that you are trying to timeout
  - but what if a signal handler is called? Might there be kernel calls made there? The same does not apply if the kernel call is made from within a signal handler that had preempted

# It can be used for checking/cleanup:

No time means "do not block"

```
event.sigev_notify = SIGEV_UNBLOCK;
flags = _NTO_TIMEOUT_RECEIVE;
/* loop, receiving (cleaning up) all pulses in receive queue */
do {
    /* MsgReceivePulse() wont block, if there's a pulse it
     * will return 0, otherwise it will timeout immediately
    */
    TimerTimeout (CLOCK_REALTIME, flags, &event, NULL, NULL);
    rcvid = MsgReceivePulse (chid, &pulse, ...);
} while (rcvid != -1);
/* if errno is ETIMEDOUT, then we got all the pulses */
```

This practice is not recommended for implementing polling since polling in general is wasteful of CPU. Usage like above is fine.

# Topics:

# Conclusion

You learned:

– ticksize is the fundamental quantum of time

– how to set or gradually adjust the system time

– how to get periodic notification

– how to customize timers for power-management or higher accuracy

– how to timeout kernel calls