# Real Time and Embedded Systems

by
Dr. Lesley Shannon
Email: lshannon@ensc.sfu.ca
Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc351

*Simon Fraser University*

Slide Set: 6
Date: October 27, 2011

# Slide Set Overview

- Clocks and Timers
  - Issues with Time
  - Types of Timers and Notification Schemes

- Using Timers

# Clocks and Timers

# Clocks and Timers

- Your applications may need to respond:
  - Periodically, or
  - To external events, or
  - After a specific period of time

# Clocks and Timers

- Historically one CPU was dedicated to one user
  - Programmers could use a function that loops and then wakes up after a specific time
  - A good example was the sleep() function, or high resolution nanosleep() function
  - You could calculate how fast your CPU was and the create your own sleep()

# Clocks and Timers

- Historically one CPU was dedicated to one user (cont'd)
  - Since nothing else was running on that machine, wasting processing time with an empty loop didn't matter
  - There was no other process trying to use the CPU so the sleep function could use it all
  - Multi-tasking was accomplished using interrupt routines that triggered off of system hardware or hardware timers

# Clocks and Timers

- The kernel reschedules threads due to
  - A hardware interrupt
  - A kernel call
  - A fault (exception)
- For this discussion interrupts and kernel calls are what matters

# Clocks and Timers

- Nowadays, when a thread calls sleep(x)/nanosleep()
  - The kernel puts the thread on hold for “x” seconds
  - The thread is removed from the running queue
  - The kernel starts a timer

# Clocks and Timers

- The kernel also typically receives regular hardware interrupts from the computer’s clock hardware
  - 10ms/1ms resolution
  - Every time one of these interrupts occurs, the kernel’s clock ISR increments its time-of-day variable  by 10ms (1ms)

# Clocks and Timers

- The kernel implements a 15-second timer by
    1. Setting a variable to current time plus 15 seconds
    2. Comparing this variable inside the ISR to the current time-of-day
    3. When the current time-of-day is the same or greater, restoring the thread to the ready queue

# Clocks and Timers

- Where does a clock interrupt come from?

# Errors in Time

- The high-speed clock is being divided by an integer divisor
  - The highspeed clock rate isn’t a multiple of 10ms
  - Therefore, the ISR rate isn’t exactly 10ms ( e.g. 9.999ms)
    - 8.64s off per day
    - 5.04 minutes off per year
  - Depending on the divisor , the error could be greater or smaller
  - The kernel knows about this error and corrects for it

Independent of the integer value shown, the real value is selected to be the next faster value.

# Timer Resolution

- If the clock tick (aka clock hardware ISR) is 10 ms, can a thread sleep for only 4 ms
  - No
  - Recall the kernel sets a variable in the ISR to some value, either
    - The current time of day (it's expired already and wakes up immediately)
    - The current time of day + 10ms (that's the next clock tick)

- Therefore, the timing resolution is only as good as the clock tick

# Timing Resolution Error

- Some software people call this “Clock Jitter”
  - Bad name for this concept
  - In hardware, clock jitter is unwanted variation in phase, frequency or amplitude (high frequency noise on the wire)
  - However, if the clock tick resolution is 10ms, what is the problem with requesting a 20ms timeout?
    - Put another way, will you get exactly 20ms of delay?

# Timing Resolution Error

- No!
  - Remember, when a thread (TA) is blocked, it is taken off the running queue
    - Another thread (TB) at the same priority may start using the CPU
    - After the 20ms expires, thread TA will be placed at the end of the READY queue for that priority
    - Depending on what thread is currently running, TA may not get to run on the processor
    - This also applies to interrupt handlers
  - Key point
    - Just because a thread is READY doesn't mean it runs on the CPU

# Timing Resolution Error

- No!
  - Reason two has to do with the resolution of the clock tick
  - The request is asynchronous to the clock source
    - Therefore the delay ranges from just over 20ms to just under 30ms

# Timing Resolution Error

- Timing resolution error is unavoidable
  - The only way to reduce the error is to reduce clock tick period, increasing the resolution to within the system's required tolerance
  - This error only happens on the first clock tick and thus the actual delay is the requested delay + some percentage of the clock tick period
    - For longer delays, this may not matter too much (i.e. a 10 ms error on a 3-hour delay is probably negligible)

# Types of Timers and Notification Schemes

# Types of timers

- Relative Timers
  - What we've been discussing so far
  - Delay for a specified time

- Absolute timers
  - "Time" started at January 1rst, 1970 00:00:00 GMT
  - Delay until a specified time

- When using timers be sure to pay attention to which one you are using

# Types of timers

- Periodic timers
  - Goes off after a set time period (e.g. the clock tick timer )
  - Keeps going until stopped
- One-shot timers
  - Goes off just once
  - Used to indicate a specific event
- Either way, the kernel stores the absolute time the timer is supposed to go off and the clock ISR compares it against the current time-of-day every time it fires

# Notification Schemes

- Instead of being blocked and waiting for the timer to go off, the thread can do something
  - It can keep running on the CPU
  - The kernel must somehow notify the thread when the desired
- Possible time out notification schemes are
  - Send a Signal
  - Notify a specific thread using a signal (Linux only)
  - Create a Thread – DON'T DO THIS!

# Notification Schemes

- All of the notification schemes require use of the sigevent structure
- The sigev_notify member determines the notification type
  - SIGEV_NONE: Don't asynchronously notify when the timer expires
  - SIGEV_SIGNAL* : Generate the signal sigev_signo when the timer expires
  - SIGEV_THREAD_ID: Like SIGEV_SIGNAL, but sends a signal to a specific thread
  - SIGEV_THREAD: Creates a thread
- Check out:

http://kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html

# Notification Schemes

- Thread notification can be dangerous!!
  - Every time the timer fires, a new thread is created!!
  - If the timer fires too often and this could chew up all the available system resources
  - If there are higher priority threads waiting to run (use this resource), you could effectively be blocking (starving them)
- Note there are macros designed to fill in the notification structures

# Notification Schemes

- Signal notification
  - Working on a task, but don't want to do it forever (e.g. calculating pi)
  - If you don't know how long you can wait without slowing up the system, use a signal/signal handler combination
- Sigwait() is the cheapest solution if there is no channel and the application can block

# Using Timers

# Using timers

- To use a timer, you must:
  1. Decide how you wish to be notified (signal/signal to specific thread/thread)
  2. Create the notification structure (sig_event)
  3. Create the timer object
  4. Set the timer to be relative/absolute and one-shot/periodic
  5. Start the timer

# Using timers

- To create a timer, use:

  int timer_create (clockid_t *clock_id*, struct sigevent **event*, timer_t **timerid*);

  – Set clock_id to CLOCK_REALTIME

  – The timerid acts as the handle to that specific timer object (an index to the kernel's timer table)

  – The sigevent structure tells the kernel about the type of event that occurs when it "fires"

# Using timers

- To set the type of timer, use:

  int timer_settime (timer_t *timerid*, int *flags*, struct itimerspec **value* , struct itimerspec *old*value*);

  - The timerid is from timer_create()
  - The flags specify an absolute versus relative timer
    - TIMER_ABSTIME = absolute
    - Pass in zero to use a relative timer

# Using timers

- Recall the itimerspec structure from the lab:

```
struct itimerspec
{
    struct timespec it_value;        //The one-shot value
    struct timespec it_interval;     //The periodic reload value
}
```

  – struct timespec has two values tv_sec, and tv_nsec;

# Using timers

- An example:

```
it_value.tv_sec = 1;
it_value.tv_nsec = 500000000;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

  - Periodic or one-shot?
  - Absolute or relative?

# Getting and setting the time

- clock_getres() POSIX
- clock_gettime() POSIX
- clock_settime() POSIX

Rule of Thumb: Don’t mess with time!!

# Getting and setting the time

- clock_gettime() and clock_settime() are based on kernel functions
- clock_settime() is a hard adjustment
  - The clock's current time gets changed immediately to the given value
- This can have severe consequences, especially when you move backwards in time (sometimes good/sometimes bad)
- *QNX has a function called ClockAdjust() allows you to change the time slowly
  - Over N clock ticks, increase/reduce the advancement by M nsec_inc
  - Note you never move backwards, but you may slow down

# Getting and setting the time

- Some systems let you set the resolution of the clock:
  - You can try to set the time resolution to something ridiculously small, but the kernel will stop you
  - Typically the range is 1ms to hundreds of us

- One possible exception is a high-frequency counter built into some processors
  - This high accuracy counter is particularly useful for determining how long a piece of code takes to execute (aka ***software profiling***)
  - No direct support in POSIX; you need an API

# Getting and setting the time

- If you use an SMP/CMP machine, be careful when profiling
  - The “start time” could be on one CPU and the “finish time” could be on another CPU giving you inconsistent results
  - Remember Clocks are often local to a CPU and not synchronized between CPUs
  - The solution is to force the thread to run on only one specific CPU

- Soon we’ll look at Signals, Interrupts and Device Drivers

# WARNING: Different "types" of Time

What if you adjust the clock while using a timer?

- CLOCK_REALTIME:
  - Fine with relative events: change the "real time", but the elapsed time is correct (e.g. sleep(50) )

# Different “types” of Time

## What if you adjust the clock while using a timer?

- TIMER_ABSTIME:
  - Absolute time will result in the timer going off at the absolute time in the new time base (aka the “new” real time)
  - Problem for mutex timeouts:
    - pthread_mutex_timedlock() uses an absolute time out value (therefore, if the time gets adjusted, relative timeouts will be wrong)

# Different “types” of Time

What if you adjust the clock while using a timer?

- CLOCK_MONOTONIC:
  - Always increasing count
  - Based on real time
  - Starts at zero
  - Not interchangeable with CLOCK_REALTIME
  - Will ensure that timer elapses after the required delay even if CLOCK_REALTIME changes

# Questions?

- What function starts the timer?

- What is the difference between CLOCK_REALTIME and CLOCK_MONOTONIC?

# Questions?

- Why would we use CLOCK_MONOTONIC?
- What is the maximum error in timing resolution for a clock?

# Questions?

- How does QNX's ClockAdjust() and POSIX's clock_settime differ?

- What function would you use to profile software at runtime in POSIX? What's the problem? What's the solution?

# Questions?

- What are the possible notification schemes when a timer goes off?

- What type of structure do you use as part of the notification scheme for a timer?