Real-Time Operating Systems and Languages (1)

Real-Time and Embedded Systems (M)

Lecture 10

UNIVERSITY
of
GLASGOW

Lecture Outline

- Real-time operating systems and languages
  - Clocks and timing
    - Clocks and the concept of time
    - Delays and timeouts
  - Scheduling

- Informed by examples from:
  - C and POSIX
  - Real-time Java
  - Ada

The Concept of Time

- Real time systems must have concept of time – but what is time?
  - Measure of a time interval
    - Accuracy, stability and granularity of the clock source
      - Is "one second" a well defined measure?
      - Temperature dependencies
      - Relativistic effects
    - Skew and divergence between multiple clocks
      - Distributed systems and clock synchronisation
  - Measure of the time of day
    - How is the clock synchronised?
      - Step changes or gradual skew
      - NTP, GPS, etc.
    - How are corrections handled?
      - Leap seconds
      - Changes in daylight saving time rules

- Do any of these issues matter to your application?

Clocks in Programming Languages

- How to represent time in a programming language?
  - Different representations for time intervals versus time of day?
    - It there a lossless conversion between the two?
  - How to determine accuracy, stability, granularity of the clock?
  - How to calculate time differences?
  - How to compare times?
  - How to specify particular times?

- Recall:
  - Some minutes have 61 seconds
  - Some calendar times occur twice
  - Some calendar times never occur
  - Any two clocks likely disagree

POSIX Clock API (1)

- Example of a typical clock API – similar features in Real-Time Java and Ada

```
time_t time();
double difftime(time_t t1, time_t t2);
```

Low resolution clock time in seconds since 1970. Conversion to calendar time.

Inconsistent handling of leap seconds ⇒ accurate delays across leap second difficult

```
struct tm {
  int   tm_sec;     // seconds (0 - 60)
  int   tm_min;     // minutes (0 - 59)
  int   tm_hour;    // hours (0 - 23)
  int   tm_mday;    // day of month (1 - 31)
  int   tm_mon;     // month of year (0 - 11)
  int   tm_year;    // year - 1900
  int   tm_wday;    // day of week (Sunday = 0)
  int   tm_yday;    // day of year (0 - 365)
  int   tm_isdst;   // is summer time in effect?
  char *tm_zone;    // timezone name
  long  tm_gmtoff;  // offset from UTC
};
struct tm localtime(time_t t);
time_t    mktime(struct tm *t);
```

POSIX Clock API (2)

```
#include <sys/time.h>

struct timespec {
        time_t   tv_sec;
        long     tv_nsec;
};
int clock_gettime(CLOCK_REALTIME, struct timespec *t);
int clock_getres (CLOCK_REALTIME, struct timespec *r);
```

High resolution clock, counting seconds and nanoseconds since 1970.
Known clock resolution.

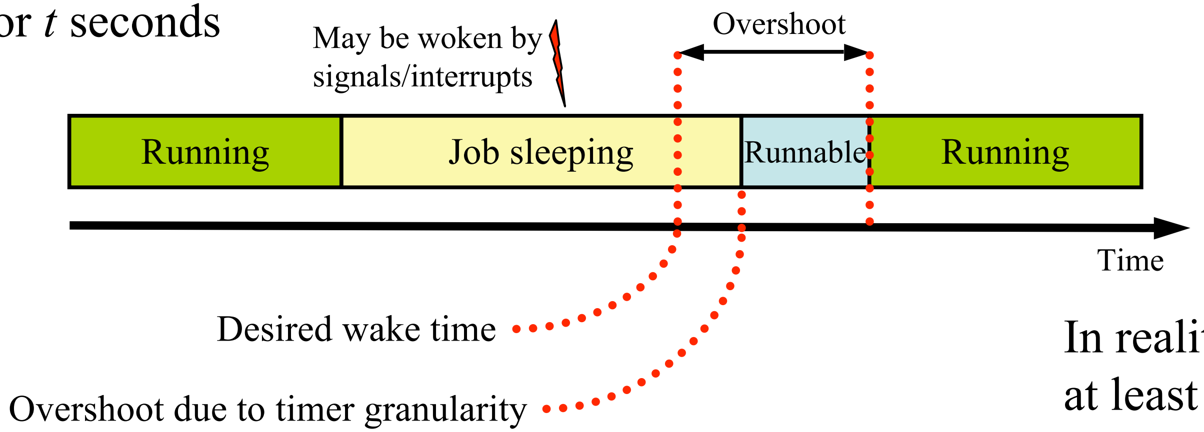
```
int nanosleep(struct timespec *delay, struct timespec *remaining);
```

Sleep for the interval specified. May return early due to signal (in which case **`remaining`** gives remaining delay). Otherwise will return after the specified delay.

Accuracy of delay not known (and not necessarily correlated to **`clock_getres()`** value)

Time Delays

- In addition to having access to a clock, need ability to:
  - Delay execution for a relative period of time
    - Delay for *t* seconds



In reality: delay at least *t* seconds

    - Delay for *t* seconds after event *e* begins

```
start = curr_time();
do_action1();
delay(10.0 - (curr_time() - start));
do_action2();
```

What if pre-empted between these? Oversleep unless system has a function **`delay_until(start+10.0)`**

  - Delay execution until an arbitrary calendar time
    - What does this mean during daylight saving time changeover?

Timeouts

- Synchronous blocking operations can include timeouts
  - Synchronisation primitives
    - Semaphores, condition variables, mutex locks, etc
  - Networking and other I/O calls
    - E.g. **`select()`** in POSIX

- May also provide an asynchronous timeout signal
  - Detect time overruns during execution of periodic task
  - In Ada:

```
select
    delay 0.1
then abort
    do_stuff();
end select;
```

Aborts call to **`do_stuff()`** if not complete after 0.1 seconds

  - Real-time Java also has overrun handlers

Scheduling

- Scheduling API typically doesn't support clock-driven scheduling
  - Limited to cyclic executives, not usually in full real-time operating systems

- Scheduling API should provide support for priority scheduling of:
  - Periodic tasks
    - At minimum should support setting thread priorities; time delays
    - Useful to allow specification of (ϕ, p, e, D) tuple
  - Aperiodic tasks
    - At minimum should support background server
    - May support sporadic or deferrable servers; consumption/replenishment rules
  - Sporadic tasks
    - Should support specification of deadlines, processor time requirements
    - Acceptance test, failure handler

Scheduler Case Studies

- Case studies in scheduler API design:
  - C and POSIX
  - Real-time Java

- Demonstrate the style of scheduler programming API available
- Provide *most* of the scheduling algorithms we have discussed

C and POSIX

- IEEE 1003 POSIX
  - "Portable Operating System Interface"
  - Defines a subset of Unix functionality, various (optional) extensions added to support real-time scheduling, signals, message queues, etc.
  - Widely implemented:
    - Unix variants and Linux
    - Dedicated real-time operating systems
    - Limited support in Windows

- Several POSIX standards for real-time scheduling
  - POSIX 1003.1b ("real-time extensions")
  - POSIX 1003.1c ("pthreads")
  - POSIX 1003.1d ("additional real-time extensions")
  - Support a sub-set of scheduler features we have discussed

Detecting POSIX Support

- If you need to write portable code, e.g. to run on Unix or Linux systems, you can check the presence of POSIX 1003.1b via pre-processor defines:

```
#include <stdio.h>
#include <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
        printf("POSIX Process scheduler\n");
#endif
#ifdef _POSIX_THREADS
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
        printf("POSIX thread schduler\n");
#endif
#endif
```

- Access to POSIX real-time extensions is usually privileged on general purpose systems (e.g. suid root on Unix)
  - Remember to drop privileges!

POSIX Scheduling API (Processes)

```
#include <unistd.h>
#include <sched.h>

struct sched_param {
        int               sched_priority;
        int               sched_ss_low_priority;
        struct timespec   sched_ss_repl_period;
        struct timespec   sched_ss_init_budget;
};

int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, struct sched_param *sp);
int sched_setparam(pid_t pid, struct sched_param *sp);

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

int sched_rr_get_interval(pid_t pid, struct timespec *t);

int sched_yield(void);
```

POSIX Scheduling API (Threads)

```
#include <unistd.h>
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_getschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *p);
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *p);

int pthread_create(pthread_t      *thread,
                   pthread_attr_t *attr,
                   void *(*thread_func)(void*),
                   void   *thread_arg);
int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
```

- Thread scheduling API mirrors process scheduling API
  - Same scheduling policies, priorities, etc.

POSIX Scheduling API

- Four scheduling policies:
  - **SCHED_FIFO** Fixed priority, pre-emptive, FIFO scheduler
  - **SCHED_RR** Fixed priority, pre-emptive, round robin scheduler
  - **SCHED_SPORADIC** Sporadic server
  - **SCHED_OTHER** Unspecified (often the default time-sharing scheduler)
  - Implementations can support alternative schedulers

- A process can **sched_yield()** or otherwise block at any time

POSIX Scheduling API: Priority Scheduler

- POSIX 1003.1b provides (largely) fixed priority scheduling
  - Priority can be changed using **`sched_set_param()`**, but this is high overhead and is intended for reconfiguration rather than for dynamic scheduling
  - No direct support for dynamic priority algorithms (e.g. EDF)

- Limited set of priorities:
  - Use **`sched_get_priority_min()`**, **`sched_get_priority_max()`** to determine the range
  - Guarantees at least 32 priority levels

Using POSIX Scheduling: Rate Monotonic

- Rate monotonic and deadline monotonic schedules can naturally be implemented using POSIX primitives
  1. Assign priorities to tasks in the usual way for RM/DM
  2. Query the range of allowed system priorities

     `sched_get_priority_min()`

     `sched_get_priority_max()`
  3. Map task set onto system priorities
     - Care needs to be taken if there are large numbers of tasks, since some systems only support a few priority levels
  4. Start tasks using assigned priorities and **`SCHED_FIFO`**

- No explicit support for indicating deadlines, periods

POSIX Scheduling API: Sporadic Server

- POSIX 1003.1d defines a hybrid sporadic/background server

```
struct sched_param {
  int               sched_priority;
  int               sched_ss_low_priority;
  struct timespec sched_ss_repl_period;
  struct timespec sched_ss_init_budget;
};
```

Additional **sched_ss_**… parameters added for the sporadic server

- When server has budget, runs at **sched_priority**, otherwise runs as a background server at **sched_ss_low_priority**
  - Set **sched_ss_low_priority** to be lower priority than real-time tasks, but possibly higher than other non-real-time tasks in the system
- Also defines the replenishment period and the initial budget after replenishment

POSIX Scheduling API: EDF

- EDF scheduling is not supported by POSIX
- Conceptually would be simple to add:
  - A new scheduling policy
  - A new parameter to specify the relative deadline of each task
  - But, requires the kernel to implement deadline scheduling
    - POSIX grew out of the Unix community
    - Unlike priority scheduling, difficult to retro-fit deadline scheduling onto a Unix kernel…

Summary of POSIX Scheduling

- Fixed priority scheduling
  - Rate monotonic algorithm
  - Widely supported
- Sporadic server can be used for aperiodic or sporadic tasks
  - Not widely supported on general purpose systems

- No support for earliest deadline scheduling
  - Some specialised RTOS support these
  - Earliest deadline scheduling more widely used to schedule network packets

Real-Time Java

- JSR-1: Real-Time Specification for Java
  - Version 1.0.1 (August 2004)
  - http://www.rtj.org/

- Extends Java with a `schedulable` interface and `RealtimeThread` class, and numerous supporting libraries
  - Definition of timing and scheduling parameters
    - Periodic tasks
    - Aperiodic and sporadic tasks
  - Definition of memory requirements
    - Extensions to the garbage collection model for real-time operation [see lecture 18 and 19]

- Requires a modified Java virtual machine
  - Due to changes to memory model, garbage collector, thread scheduling

Real-Time Java: Release Parameters

```
abstract class ReleaseParameters
{
    RelativeTime        cost
    RelativeTime        deadline
    AsyncEventHandler   overrunHandler
    AsyncEventHandler   missHandler
    ...
}
```

Extends

```
class PeriodicParameters
{
    HighResolutionTime start
    RelativeTime       period
    ...
}
```

```
class AperiodicParameters
{
    ...
}
```

```
class SporadicParameters
{
    RelativeTime minInterarrival
    ...
}
```

- Class hierarchy to express release timing parameters
- Supports deadline monitoring:
  - **`missHandler`** if deadline exceeded
- Supports execution time monitoring:
  - **`cost`** = needed CPU time
  - **`overrunHandler`** if execution time budget exceeded
- Unusual definition of aperiodic and sporadic tasks
  - Aperiodic tasks may have deadline; sporadic tasks differ because they also have minimum inter-arrival time

Real-Time Java: Scheduling Parameters

- Abstract `Scheduler` and `SchedulingParameters` classes defined
  - Allows a range of schedulers to be developed
    - Current standards only allow system-defined schedulers; cannot write a new scheduler without modifying the JVM
    - Likely to be extended to provide a pluggable scheduler API in future
  - Current standards provide only a pre-emptive priority scheduler
    - Conceptually similar to the POSIX priority scheduler
      - Presumably to make implementation simpler
    - Allows monitoring of execution times; missed deadlines; CPU budgets
    - Allows thread priority to be changed programmatically
      - Can be used to implement sporadic servers
    - Limited support for acceptance tests

Real-Time Java: Real time Threads

```
class RealtimeThread extends java.lang.Thread
{
    // ...adds additional constructors to specify
    // ReleaseParameters and SchedulingParameters
    ...

    // ...adds additional methods:
    public void    setScheduler(Scheduler s);
    public void    schedulePeriodic();
    public boolean waitForNextPeriod();
    ...
}
```

- The `RealtimeThread` class extends `Thread` with extra methods and parameters
  - Direct support for periodic threads
    - `run()` method will be a loop ending in a `waitForNextPeriod()` call
    - Contrast with POSIX APIs which require programmer to calculate explicit delay each period

Scheduling

- POSIX and Real-Time Java provide generally similar features
  - Pre-emptive priority scheduler for periodic tasks
    - Suitable for RM and DM algorithms
    - Real-Time Java also provides periodic threads
  - Limited support for sporadic and aperiodic tasks
    - Sporadic server included in POSIX standards; not widely implemented

- Both have scope for non-standard extensions
  - E.g. some RTOS extend POSIX scheduling

Summary

- Real-time operating systems and languages
  - Clocks and timing
    - Clocks and the concept of time
    - Delays and timeouts
  - Scheduling

- Additional reading:

  E. A. Lee, “Absolutely Positively on Time: What Would it Take?”, IEEE Computer, July 2005.