Today’s Goals

- **Timing Faults**
  - Deadline miss detection
  - WCET overrun detection
  - Sporadic Overrun

- **Damage Confinement**
  - Execution time servers

- **Error Recovery**

- **Mode Change**
Timing Faults

How to spot a timing fault

Where do Timing Faults Come From? ... or: Theory Meets Practice
- WCET calculation incorrect
- Underestimated blocking times
- Invalid assumptions in schedulability analysis
- Errors in schedulability analysis
- Wrong schedulability theory
- System working outside design parameters

Detecting and Tolerating Timing Faults
- Deadline miss
- WCET overrun
- Sporadic overrun
- Overuse of resources
Timing Faults

How to spot a timing fault

Missed deadlines

Where do Timing Faults Come From? ... or: Theory Meets Practice

- WCET calculation incorrect
- Underestimated blocking times
- Invalid assumptions in schedulability analysis
- Errors in schedulability analysis
- Wrong schedulability theory
- System working outside design parameters

Detecting and Tolerating Timing Faults

- Deadline miss
- WCET overrun
- Sporadic overrun
- Overuse of resources
Detecting a deadline miss

**Ada**
- No built-in/explicit support for detecting missed deadlines
- Use separate watchdog thread
- Use ‘select delay until ... then abort ...’ (ATC)

**C/Real-Time POSIX**
- No built-in/explicit support for detecting missed deadlines
- Use separate watchdog thread

**Real-Time Java**
- Yes built-in/explicit support for detecting missed deadlines
- Throws exception when deadline miss is detected
- Handled by “normal” exception handler
```c
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer; /* shared btw. monitor and server */
struct timespec deadline = ...;
struct timespec zero = ...;
struct itimerspec alarm_time, old_alarm;
struct sigevent s;

void server(timer_t *watchdog) {
    /* perform service */
    TIMER_DELETE(*watchdog);
}

void watchdog_handler(int signum, siginfo_t *data, void *extra) {
    /* SIGALRM handler - server is late */
}
```
```c
void monitor() {
    pthread_attr_t attributes;
    pthread_t serve;

    sigset_t mask, omask;
    struct sigaction sa, osa;
    int local_mode;

    SIGEMPTYSET(&mask);
    SIGADDSET(&mask, SIGALRM);

    sa.sa_flags = SA_SIGINFO;
    sa.sa_mask = mask;
    sa.sa_sigaction = &watchdog_handler;

    SIGACTION(SIGALRM, &sa, &osa); /* assign handler */
}
alarm_time.it_value = deadline;
alarm_time.it_interval = zero; /* one shot timer */

s.sigev_notify = SIGEV_SIGNAL;
s.sigev_signo = SIGALRM;

TIMER_CREATE(CLOCK_REALTIME, &s, &timer);
TIMER_SETTIME(timer, TIMER_ABSTIME,
    &alarm_time, &old_alarm);

PTHREAD_ATTR_INIT(&attributes);
PTHREAD_CREATE(&serve,&attributes,
    (void *)server, &timer);
public class Periodic extends RealTimeThread {
    public Periodic(PeriodicParameters P) {...};

    public void run() {
        boolean deadlineMet = true;
        while (deadlineMet) {
            // task code
            ...
            deadlineMet = waitForNextPeriod();
        }
    }
}
Detecting a deadline miss: Real-Time Java

Handling a deadline miss

- Threads with a detected deadline miss are automatically de-scheduled
- Must be explicitly re-scheduled
- `waitForNextPeriod()` along with miss counter (`deadlineMiss`) indicates status

Class `RealTimeThread` support for de-/re-scheduling

```java
package javax.realtime;
public class RealTimeThread extends Thread {
    implements Schedulable {
        ...
        public boolean waitForNextPeriod();
        public void deschedulePeriodic();
        public void schedulePeriodic();
        ...
    }
}
Issues in Deadline Miss Detection

- No support for specifying temporal constraints for miss handlers
- **Block level** deadline miss detection possible in research languages
Worst-Case Execution Time Overrun

Why bother?

- Localise and confine faults
- Deadline miss may be caused by other tasks

Execution Time Clocks in C/Real-Time POSIX

- Create watchdog for execution time
- Two special, execution time, clocks (per thread/process) supported:
  - CLOCK_PROCESS_CPUTIME_ID
  - CLOCK_THREAD_CPUTIME_ID
- Standard clock functions supported
  - clock_settime(CLOCK_PROCESS_CPUTIME_ID,...)
  - clock_gettime(CLOCK_PROCESS_CPUTIME_ID,...)
  - clock_getres(CLOCK_PROCESS_CPUTIME_ID,...)
- Also supports monitoring of other threads/processes
### Execution Time Monitoring in Ada

- Supported by the `Execution_Time` package (and sub-packages)
- Defines execution time clocks at the task level
- Timers based on execution time clocks can fire events
- Handled by “standard” event handling
- Requires manipulation of ceiling priorities

### Execution Time Monitoring in Real-Time Java

- Execution time clocks **not** supported (in general)
- Monitoring of ‘cost’ is supported through exceptions
  - Similar to deadline miss detection:
  - Implementation dependent: support for execution time/cost mapping
  - Uses asynchronous events (interrupts) to communicate
  - Co-operates with scheduler
- Some support for execution time/cost statistics
Worst-Case Execution Time Overrun: Summary

- Attempt to localise and confine timing faults
- Highly language dependent
  - C/Real-Time POSIX: low level primitives with execution time clocks
  - Ada: low/medium level primitives with execution time clocks and timing events
  - Real-Time Java: high-level primitives with abstract/vague notion of execution cost
- Execution time clocks: requires hardware and OS support
Sporadic Overrun

Definition (Sporadic Overrun)

A sporadic event firing more frequently than anticipated, i.e., the minimal inter-arrival time (MIT) was overestimated.

Example (Classic)

First landing on the moon: CPU on Lunar Landing Module flooded with radar data interrupts.

Solutions

- Reduce firing rate of sporadic event to comply with MIT
  - Hardware supported solutions (rare, except interrupt disable/enable)
  - Implement sporadic interrupt controller
- Bound CPU time used for handling a given sporadic event
  - Execution time server

Consider (at least) two event types: hardware interrupts, software events.
Handling Sporadic Overrun in Ada

- Receive interrupt from device
- Disable interrupts from device
- Set timer to MIT
- Re-enable interrupts when timer expires
- **Device-dependent** what happens when interrupts are ignored

### Sporadic Interrupt Controller for Hardware Interrupts

```ada
protected Sporadic_Interrupt_Controller is
    procedure Interrupt; -- mapped onto real interrupt
    entry Wait_For_Next_Interrupt;
private
    procedure Timer(Event: in out Timing_Event);
    Call_Outstanding : Boolean := False;
    MIT : Time_Span := Milliseconds(...);
end Sporadic_Interrupt_Controller;

Event : Timing_Event;
```
protected body Sporadic_Interrupt_Controller is

procedure Interrupt is
begin
  -- disable interrupts
  Set_Handler(Event, MIT, Timer’Access);
  Call_Outstanding := True;
end Interrupt;

entry Wait_For_Next Interrupt when Call_Outstanding is
begin
  Call_Outstanding := False;
end Wait_For_Next Interrupt;

procedure Timer(Event: in out Timing_Event) is
begin
  -- enable interrupts
end Timer;
end Sporadic_Interrupt_Controller;
Sporadic Interrupt Controller for Software Events

- Similar to solution for HW interrupts
- Monitor task release instead of interrupts
- Throw exception if MIT is violated (no interrupt disable)
MIT violation detected directly by the run-time system (JVM)

Programmer specified policy for handling violations
- mitViolationIgnore: the release event is ignored
- mitViolationExcept: throw an exception (in the releasing thread)
- mitViolationReplace: the last release event is overwritten with the current event
- mitViolationSave: the release event is delayed to comply with MIT

Interrupt handlers in Real-Time Java are second level
- Cannot be used to directly control interrupts
- Must adopt sporadic interrupt controller similar to Ada
## Overuse of Resource(s)

### Sources of resource abuse
- A task may monopolise resource longer than anticipated
- Resource contention not taken into account during design/analysis
  - In particular: large systems, many libraries, ...

### Timeouts not enough
- For priority inheritance, blocking is cumulative
- With ICPP/OCPP blocking starts before task execution
- Timeout not generally supported for critical sections, e.g., Java synchronised

### Solutions?
- Move control to block level
- Careful, explicit monitoring of all resource access
Damage Confinement

**Definition (Damage confinement)**
To prevent the propagation of errors to other components in the system

**In particular...**
- Protect the system from unbounded sporadic and aperiodic activity
- Support composability and temporal isolations

**Confining Sporadic and Aperiodic Activity**
- **Solution:** execution time servers
- Group sporadic/aperiodic tasks together
- Use periodic task to schedule group(s) of sporadic/aperiodic tasks
- Enables schedulability analysis on sporadic/aperiodic tasks
- Impact of unbounded sporadic/aperiodic activity is **confined**
Implementing Sporadic Servers in C/Real-Time POSIX

- Sporadic servers directly supported as a scheduling policy
- Applicable for both threads and processes

Sporadic Server

- Two priorities: “high” and “low”
- Has execution time **budget** to spend on sporadic events
- Executes at “high” priority when spending budget on handling sporadic events
- Executes at “low” priority when **replenishing** budget
- Can be analysed as a periodic task

Aperiodic Server?

- Cannot use Sporadic Server directly
- Use Sporadic Server **process**: all aperiodic threads collected in one process
Implementing Sporadic Servers in C/Real-Time POSIX

#define SCHED_SPORADIC ...
define PTHREAD_SCOPE_SYSTEM ...
define PTHREAD_SCOPE_PROCESS ...

typedef ... pid_t;
struct sched_param {
    ...
timespec sched_ss_repl_period;
timespec sched_ss_init_budget;
int sched_ss_max_repl
    ...
};

int sched_setparam(pid_t pid,
    const struct sched_param *param);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int pthread_attr_setscope(pthread_attr_t *attr,
    int contentionscope);
int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);
Built-in Support

- Cost monitoring and enforcement (optional)
- Sporadic release parameters
- Processing group parameters

Class `ProcessingGroupParameters`

```java
package javax.realtime;

public class ProcessingGroupParameters implements Cloneable {

    public ProcessingGroupParameters(
        HighResolutionTime start, RelativeTime period,
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler)

}
```
Error Recovery

**Strategies for WCET Overrun**

- For hard real time tasks: plan with plenty of slack and do nothing(!), active monitoring and graceful degradation, dedicated recovery task(s)
- For soft/firm real-time tasks: ignore (if isolation works), lower task priority, skip/abort current release

**Strategies for Sporadic Overrun**

Like Real-Time Java: Ignore, throw exception, overwrite, or delay.

**Strategies for Deadline Miss**

- For hard real-time tasks: active monitoring/two deadlines and graceful degradation
- For soft real-time tasks: count misses and otherwise ignore... until miss threshold is reached, then inform
- For firm real-time tasks: terminate since result is useless anyway
Mode Change

- Systems may (deliberately) enter situations with high degree of expected deadline misses
- Adapt, dynamically, by re-configuring

Example (Missions in space)

Space vehicles often have different modes corresponding to different phases of the overall mission: take-off, in-flight, landing.
Summary

- **Timing Faults**
  - Deadline miss detection
  - WCET overrun detection
  - Sporadic Overrun

- **Damage Confinement**
  - Execution time servers

- **Error Recovery**

- **Mode Change**