Concurrent Programming and Real-Time Scheduling (MF2008)

Lei FENG
2008.11.10

Embedded Control Systems Research Group
KTH (Royal Institute of Technology)
Stockholm, Sweden
Outline

• Brief review
• Concurrent programming (Chp. 3)
• Real-time computing & scheduling (Chp. 8)
• Real-time OS (Sec. 3.3)
• Rubus RTOS (Sec. 3.4)
Brief Review

- Popular devices for ECS: μc
- Control implementation on μc
  - Direct management of GPIO, interrupts, timers
  - Sampling period, discretization, quantization
  - Various time delays and program structure
  - Error handling, ......
Concurrent Programming

- Definition, true and pseudo parallelism
  1. Multiprogramming: on a single processor
  2. Multiprocessing: on multiple processors with shared resources
  3. Distributed processing: independent memories
Application Examples

- Recall your programs on the labs
- Main function: while(1)
- Human-machine interface: keyboard, switches, LEDs
- Sensing, actuating, and control computing
- Communication via network
- These concurrent programs might be
  - Time-triggered (PIT)
  - Event-triggered (Switch handler)
Processes vs. Threads

• A **process** is an executable instance of a program stored and operating in its own protected memory space. (Unit of allocation)

• A **thread** is a stream of execution that may share memory space with others. (Unit of execution)

• Different methods of communication
A Process Has

- Code
- Data
- Stack
- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
A Thread Has

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- 1 process has at least 1 thread
- 1 thread must belong to 1 process
An Illustrating Example

Cruise Control App.

Threads

Cruise Cntr

Sensor of Velocity

Actuator of Gas

Processes of 1 Thread

Process of 2 Threads

ref_speed

Speed Control
Program vs. Thread/Process

- Creation by function call, RTOS cfg.
- Initializing contexts
- Specifying triggering condition (when & how)
- Can T1 and T2 execute together?
  - **Reentrant** code, uses local variables only
  - Non-reentrant code, uses shared resources

```c
float ctrl(float p, ref, cur)
{
    ...
    return u;
}
```
Thread States (Thread Life Cycle)

- Pre-emption owing to scheduling method
- Blocked by shared resource or to wait for a time period
Context Switch

Thread A
- creation
- running
- ready
- preemption

Thread B
- dispatch
- running
- blocked by var.
- termination

KTH Royal Institute of Technology
Outline

• Brief review
• Concurrent programming
  – What is it?
  – Why is it used in ECS?
  – What are its characteristics?
• How to use it effectively? (Real-time Scheduling)
• Real-time OS
• Rubus RTOS
Effects of Schedules

- Task A: starts 0, deadline 3, exec. time 2
- Task B: starts 0, deadline 2, exec. time 1
Real-Time Computing Basics

- **Time**: the correctness of the system depends on both time and logic
- **Real**: the internal time scale must match the external time scale
- **Real-time**: (≠ real fast) predictable in deterministic or probabilistic timing requirements, i.e., the deadline
  - **Hard**: missing deadline may cause catastrophic or critical consequence
  - **Soft**: meeting deadline is desirable, but not critical
Examples of RT Tasks

• Hard RT tasks:
  – Sensory data acquisition
  – Detection of critical conditions
  – Control output to actuator
  – Control of safety critical systems

• Soft RT tasks:
  – HMI
  – Handling input data from the keyboard
  – Displaying messages on the screen
  – GUI
  – Logging
Categories of RT Tasks

- A **task** is a thread or process
- A **job** is an execution instance of a task
- **Periodic, aperiodic, and sporadic** (with minimal interval time)
Timing Parameters of A Task

- Relative Deadline $D_i = d_i - r_i$
- Response time $R_i = f_i - r_i$
- Lateness $L_i = f_i - d_i$ (Might be negative)
- Tardiness (exceeding time) $E_i = \max(0, L_i)$
- Laxity (Slack time) $X_i = D_i - C_i$
Jitter of the Periodic Task

- Latency $L_s^i = s_i - r_i$
- (Absolute) Jitter: $J_s = \max L_s^i - \min L_s^i$
Real-Time Scheduling

- Def: assigning (temporal & spatial) processors and resources (memory, I/O devices) to tasks to complete all tasks under constraints
- **Feasible** schedule completes all tasks under constraints
- **Schedulable** task set has at least 1 feasible schedule
- Intractable, exponential time in the number of processors and tasks (NP-hard)
- Efficient algorithms (polynomial time) exist under certain simplified assumptions
Classification of Scheduling Algorithms

• Preemptive: running tasks can be interrupted
• Non-preemptive: (More difficult)
• Static: based on fixed parameters
• Dynamic: based on dynamic parameters
• Off line: the execution of all tasks are determined before activation (as a table)
• On line: decisions are made at runtime when the task set varies
• Optimal: minimize a “cost”, or no one is better than it
• Heuristic: no guarantee of the optimality
Cyclic Executive

- Old fashion
- Off line, static construction of an execution table
- With or without OS support
- Manual or optimization-based
- Example: A (exe=0.5, per.=2), B (2, 6)
- Analysis within a hyperperiod (LCM)
Pros and cons of Cyclic Executive

+ Very efficient implementation using periodic timer interrupts and a dispatch table
+ Simple and relatively deterministic
- Manual partition of long tasks
- Changes (e.g., new tasks, different execution time) can cause complete reschedule
- Difficult to find a feasible schedule
- Assumes a closed environment
- Periodic tasks only
Fixed-Priority Scheduling

- On line, preemptive, static
- A task has a fixed priority (may allow “mode switch”), and all its jobs inherit it
- The scheduler dispatches the highest priority job (and may preempt a low priority job)
- Predominant in RTOS
- Priorities are represented by integer values
- Different systems use different ordering schemes. Rubus RTOS is different
Rate Monotonic (RM) Scheduling (Liu & Layland)

- Priority is proportional to rate
- Optimal fixed-priority policy, for periodic tasks that
  - are released at the beginning of a period and must complete within the period
  - are independent
  - have fixed computation time (or upperbound, WCET), less than the period
- Example A (0.5, 1), B (1, 2)
Feasibility Test of RM Policy

- CPU utilization (rate) of a task: $C_i/T_i$
- A set of $n$ task is schedulable if
  $U = C_1/T_1 + \ldots + C_n/T_n \leq n(2^{1/n} - 1)$
- Limit for 2 is 0.8284
- When $n \rightarrow \infty$, limit $\rightarrow \ln(2) = 0.6931$
- Sufficient but NOT necessary
- In general, limit is around 0.88
- 1 can be reached if periods are **harmonic**

Q: What is the $U$ For A and B?
Further Developments of FP

- Exact feasibility analysis (iff)
- Analysis of task interaction (Next lecture)
- Inclusion of aperiodic tasks (organize them as periodic tasks using *servers*)
- Overload management: relax the usage of WCET. Properly set task priority, skip some jobs, ......
- Multiprocessors and distributed systems
Dynamic-Priority Scheduling

- On line, preemptive, dynamic
- Priorities are assigned to individual jobs dynamically
- The scheduler dispatches the highest priority job
- Slightly higher computation overhead
- Not available in most commercial RTOS (like Rubus RTOS)
Earliest Deadline First (EDF) Scheduling (Liu & Layland)

- Priorities are assigned inversely proportional to the absolute deadlines of the active jobs.
- Under the same assumption for RM, n periodic tasks are schedulable if and only if
  \[ U = \frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq 1 \]
- Optimal among **ALL** preemptive scheduling algorithms.
Related Results

- Deadline Monotonic (DM) for aperiodic tasks
- RM vs. EDF: RM easy implementation, but EDF outperforms it in most (soft) applications [1]
- Server based scheduling for Aperiodic tasks
- Transient overloads (U<1)
  - Activation overrun: admission control
  - Execution overrun: resource reservation
- Permanent overloads (U>1)
  - Job skipping, quality of service (QoS),
  - Elastic scheduling (varying rates)
  - Feedback control
References for Real-Time Scheduling

Outline

• Brief review
• Concurrent programming
  – What is it?
  – Why is it used in ECS?
  – What are its characteristics?
• Real-time Scheduling
• Real-time OS (How to implement it?)
• Rubus RTOS
Implementation without OS

• Background/foreground programming
• Foreground:
  – Interrupt programming provided by the microcontroller
  – Fixed priority provided by hardware
• Background: main function, while loop
• Advantages: simple, efficient, small size
• Drawbacks:
  – Low level programming (interrupts, ports, timers)
  – Difficult to debug and re-allocate
  – Complex
Concurrent Programming using (Real-Time) OS

- Scheduling of multi-tasks, and
- Management of memory, I/O, files, etc.

Application programs

OS provides services

Hardware is transparent
Scheduling Threads on one CPU

- **Queue of ready threads sorted by scheduling policy**
- **Waiting queues sorted by scheduling policy**

- creation
- dispatch
- pre-empt
- termination
Real-Time Kernel

- Also called Microkernel
- Core part of an RTOS
- Supports concurrent threads and scheduling
- Task communication and protection of shared resources
- Error detection and handling
Insufficiency of General Purpose Computer Systems

- GP computers optimized for high-throughput on the average, but *not predictable*
- Direct memory access (DMA) steals CPU time
- Nondeterministic access time of cache
- Interrupts of I/O devices
- Unknown blocking of shared resources
- Memory paging causes significant delay
- Uncontrollable queues in the kernel (FIFO)
Real-Time OS

- Improvement on the limitations of GP computers
- No DMA on the microcontroller or better control method
- No cache on the microcontroller or consider cache effects on WCET
- Direct access to I/O devices
- Time bounded access of shared resources
- Static memory allocation
- Flexible scheduling policies
Introduction to the Rubus RTOS

- Aimed for safety-critical real-time applications
- Rubus RTOS includes 3 parts:
  1. “Blue kernel” supports preemptive FP scheduling
  2. “Red kernel” supports non-preemptive off line
  3. “Green kernel” supports interrupt handling
- Tasks are defined through a configuration file at design time
- Rubus does not support dynamic task creation
Structure of the Rubus OS

Applications utilising red & blue services

"System calls" - application programmer interface

Rubus OS

Red kernel | Green kernel | Blue kernel

Basic Services

Hardware Adaption Layer

Hardware
Rubus Programming Structure

'RED' THREADS:
- Start (periodically)
- Execute once
- Terminate
  "Delay until next tick"

'DIFFERENCES:
- Non blocking, predictable timing
- Shared stack

'BLUE' THREADS
- Initialization
- While (condition) { 
  do something
  possibly wait for something
  do something else
}

- Can block, less predictable
- One stack per task
Discussion

• **Red kernel** for highly deterministic implementation of periodic activities (hard)
  – Off line, cyclic executive
  – Mode switch to increase flexibility

• **Blue kernel** for less time-critical and event-triggered activities (soft)
  – FP, preemptive, priorities (0-15), 0 the smallest
  – FIFO for tasks with identical priority

• Error detection and handling
  – Monitor timing requirements of system calls
  – `redError`, `blueError` (deadline overrun interrupt)
Summary of Lecture 3

- What are concurrent programming, threads, tasks, jobs?
- Real-time scheduling methods: FP, RM, EDF
- Sufficient (not necessary) condition of RM: 
  \[ \frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \]
- Sufficient and necessary condition of EDF: 
  \[ \frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq 1 \]
- How to realize concurrent programming?
- Foreground/background programming
- RTOS, and why is it more predictable?