Embedded Software as Tasks

Static and Dynamic Aspects of Implementation of Embedded Software (Conceptualized as Tasks)

Rajesh Gupta
Computer Science and Engineering
University of California, San Diego.
Embedded software on a processor

• Typical implementation approaches
  • Synchronous
    • single program
  • Asynchronous
    • foreground/background system
    • multi-tasking
Consider the following example

- A process controller with following modules
  - a clock tick comes every 20 ms when a clock module must run
  - a control module must run every 40 ms
  - three modules with soft constraints
    - operator display update
    - operator input
    - management information logs
Single Program Approach

```c
while (1) {
    wait for clock;
    do clock module;
    if (time for control) do control;
    else if (time for display update) do display;
    else if (time for operator input) do operator input;
    else if (time for mgmnt. request) do mgmnt. output;
}
```

- Must have \( t_1 + \max(t_2, t_3, t_4, t_5) \leq 20 \text{ ms} \)
  - may require splitting tasks... gets complex!
int main(void) {
  Init_All();
  for (;;) {
    IO_Scan();
    IO_ProcessOutputs
    KBD_Scan();
    PRN_Print();
    LCD_Update();
    RS232_Receive();
    RS232_Send();
    TMR_Process();
  }

  // should never ever get here
  // can put some error handling here, just in case
  return (0); // will keep most compilers happy
}
Observations on the Single Program Approach

• Each function called in the infinite loop represents an independent task
• Each of these tasks must return in a reasonable time, no matter what ‘thread’ of code is being executed
• We have no idea at what frequency our main loop runs.
  – In fact, the frequency is not constant and can significantly change with the changes in system status
    • (as we are printing a long document or displaying a large bitmap, for example)
• Mix of periodic and event-driven tasks
  – Most tasks are event driven
    • e.g. IO_ProcessOutputs is an event-driven task
    • dedicated input event queue associated with them
      – e.g. IO_ProcessOutputs receives events from IO_Scan, RS232_Receive, and KBD_Scan when an output needs to be turned on
  – Others are periodic
    • No trigger event, but may have different periods, and may need to change their period over time
Observations on the Single Program Approach (contd.)

• Need some simple means of inter-task communications
  – e.g. may want to stop scanning the inputs after a particular keypad entry and restart the scanning after another entry
    • require a call from a keypad scanner to stop the I/O scanner task
  – e.g. may also want to slow down the execution of some tasks depending on the circumstances
    • say we detect an avalanche of input state changes, and our RS-232 link can no longer cope with sending all these messages
    • like to slow down the I/O scanner task from the RS-232 sending task

• May need to perform a variety of small but important duties
  – e.g. dim the LCD exactly one minute after the very last key was pressed, flash a cursor on the LCD at a periodic, fixed and exact frequency.
  – dedicating a separate task to each of these functions may be an overkill
Going Beyond Single Program Software

- Asynchronous implementation approaches
  - Foreground/background systems
  - Multitasking

**Foreground (interrupt)**

```plaintext
on interrupt {
    do clock module; if (time for control) do control;
}
```

**Background**

```plaintext
while (1) {
    if (time for display update) do display;
    else if (time for operator input) do operator;
    else if (time for mgmnt. request) do mgmnt.;
}
```

- Decoupling relaxes constraint: $t1 + t2 \leq 20$ ms
Multi-tasking Approach

- Single program approach: one “task”
- Foreground/background: two tasks
- Generalization: multiple tasks
  - also called processes, threads etc.
  - each task carried out in parallel
    - no assumption about # of processors
    - tasks simultaneously interact with external elements
      - monitor sensors, control actuators via DMA, interrupts, I/O etc.
    - often illusion of parallelism
  - requires
    - scheduling of these tasks
    - sharing data between concurrent tasks
Task Characteristics

• Tasks may have:
  – resource requirements
  – importance levels (priorities or criticalness)
  – precedence relationships
  – communication requirements
  – And, of course, timing constraints!
    – specify times at which action is to be performed, and is to be completed
    – e.g. period of a periodic task
    – or, deadline of an aperiodic task
Preemption

- **Non-preemptive:**
  - A task, once started, runs until it ends or has to do some I/O

- **Preemptive:** A task may be stopped to run another
  - incurs overhead and implementation complexity
  - but has better schedulability
  - with non-preemptive, quite restrictive constraints
    - e.g. N tasks, with task j getting ready every $T_j$, and needs $C_j$ time during interval $T_j$
      - then: $T_j \geq C_1 + C_2 + \ldots + C_N$ in the worst case
      - because all other tasks may already be ready

  i.e. period of every thread $\geq$ sum of computation times!
So, then how to organize multiple tasks?

• Cyclic executive (Static table driven scheduling)
  – static schedulability analysis
  – resulting schedule or table used at run time
  – TDMA-like scheduling
• Event-driven non-preemptive
  – tasks are represented by functions that are handlers for events
  – next event processed after function for previous event finishes
• Static and dynamic priority preemptive scheduling
  – static schedulability analysis
  – no explicit schedule constructed: at run time tasks are executed “highest priority first”
  – Rate monotonic, deadline monotonic, earliest deadline first, least slack
Continued…

• Dynamic planning-based scheduling
  – Schedulability checked at run time for a dynamically arriving task
  – “admission control”
  – resulting schedule to decide when to execute
• Dynamic best-effort scheduling
  – no schedulability checking is done
  – system tries its best to meet deadlines
Performance Characteristics to Evaluate Scheduling Algorithms

• Static case: off-line schedule that meets all deadlines
  – secondary metric:
    • maximize average earliness
    • minimize average tardiness
• Dynamic case: no \textit{a priori} guarantees that deadline would be met
  – metric:
    – maximize \# of arrivals that meet deadline
Cyclic Executive, or Static Table-driven Scheduling

- Application consists of a fixed set of processes
- All processes are periodic, with known periods
  - aperiodic tasks can be converted into periodic by using worst case inter-arrival time
- Processes are completely independent of each other
- Zero overhead costs
- Processes have deadlines equal to their periods, i.e., each process must complete before it is next released
- All processes have fixed WCET
- A table is constructed and tasks are dispatched accordingly repeatedly
  - feasible schedule exists iff there is a feasible schedule for the LCM of the periods
  - heuristics such as earliest deadline first, or shortest period first
- Predictable, but inflexible
  - table is completely overhauled when tasks or their characteristics change
Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Period</th>
<th>Computation Time C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>50</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

Diagram showing the processes and computation times.
Cyclic Executive

• Minor cycle: 25ms
• Major cycle: 100ms
• A major cycle contains a number of minor cycles
  – During execution a clock interrupts every 25ms that enables the scheduler to loop through the four minor cycles

```plaintext
loop
  Wait_For_Interrupt;
  Procedure_For_A;
  Procedure_For_B;
  Procedure_For_C;
  Wait_For_Interrupt;
  Procedure_For_A;
  Procedure_For_B;
  Procedure_For_D;
  Procedure_For_E;
  Wait_For_Interrupt;
  Procedure_For_A;
  Procedure_For_B;
  Procedure_For_C;
  Wait_For_Interrupt;
  Procedure_For_A;
  Procedure_For_B;
  Procedure_For_D;
end loop;
```
Cyclic Executive

• So, no actual processes exist at run-time, each minor cycle is just a sequence of procedure calls

• The procedures share a common address space and can thus pass data between themselves.
  – This data does not need to be protected because concurrent access is not possible.

• It is difficult to incorporate sporadic processes, processes with long periods (may need to be split...)

• However, if it is possible to construct a CE, then no further schedulability test is needed
  – A bin packing problem
  – A typical system with 40 minor cycles and 400 entries
Priority based scheduling

• Non-preemptive scheduling
  – A lower priority task completes before the next available higher priority task executes

• Preemptive scheduling
  – Preempt executing task based on priority

• Deferred preemption, or cooperative dispatching
  – Allow a lower priority task to complete for a bounded time (but not necessarily to completion)
Priority-based Preemptive Scheduling

• Tasks assigned priorities, \textit{statically} or \textit{dynamically}
  – priority assignments relates to timing constraints
  – static priority attractive… no recalculation, inexpensive

• At any time, task with highest priority runs
  – if low priority task is running, and a higher priority task arrives, the former is preempted and the processor is given to the new arrival

• Appropriate assignment of priorities allow this to handle certain types of real-time cases

• Process “states”
  – Runnable
  – Suspended – waiting for a timing event
    • Useful for periodic processes
  – Suspended – waiting for a non-timing event
    • Useful for sporadic processes
  – (assume no IPC for now)
Scheduling and Schedulability Tests

- Scheduling is determination of ‘next task to run’
  - Can be as simple as determine task priorities
- Schedulability test
  - A test that determines whether a set of ready tasks can be scheduled such that each task meets its deadline
- Tests can be
  - Exact
  - Necessary
  - Sufficient

Complexity of the task set
Some Priority-based Preemptive Scheduling Approaches: RMS

- Rate Montonic Algorithm by Liu & Layland 73 / Serlin 72
  - static priorities based on periods
    - higher priorities to shorter periods
    - Priority is a monotonic function of rate
  - Slow generalization over a decade, leading to generalized RMS by Klein 93
- Primary focus on periodic processes, hard deadlines, statically characterizable task sets, fixed/bounded WCET
  - Not suitable of mixed media applications
- Used in several applications
  - IBM 1989 (Sonar Training); Use by IBM Federal Division in all its RT projects
  - Adopted by NASA for space station data management system in 1990
  - Influenced design of IEEE Futurebus, POSIX, Ada 95
Dynamic Prioritization

• Earliest-deadline First
  – dynamic priority assignment
  – closer a task’s deadline, higher is it priority
  – applicable to both periodic and aperiodic tasks
  – need to calculate priorities when new tasks arrives
    • more expensive in terms of run-time overheads
• Key results on schedulability bounds!
  – “When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set J of jobs with arbitrary release times and deadlines on a processor if and only if J has feasible schedules.”
    • Proof: based on the fact that any feasible schedule of J can be systematically transformed into an EDF schedule (i.e., produced by EDF algorithm).
Rate Monotonic Priority Assignment

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, $T$</th>
<th>Priority, $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>105</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>75</td>
<td>2</td>
</tr>
</tbody>
</table>
Schedulability Tests

- **Utilization Based**
  \[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(\sqrt{n} - 1\right) \]
  \[ \lim_{n \to \infty} n\left(\sqrt{n} - 1\right) = \ln 2 \approx 0.693 \]

- **Optimal:**
  - If any static priority algorithm can meet all the deadlines for a given task set then RMS can also.
  - Not so, if deadline \(<>\) period

- **Elegant but not exact**
  - In fact, a randomly generated task set will meet all deadlines when utilization is 85% or less.

\[ \sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) < N(2^{1/N} - 1) \]

<table>
<thead>
<tr>
<th>(N)</th>
<th>Utilization Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0%</td>
</tr>
<tr>
<td>2</td>
<td>82.8%</td>
</tr>
<tr>
<td>3</td>
<td>78.0%</td>
</tr>
<tr>
<td>4</td>
<td>75.7%</td>
</tr>
<tr>
<td>5</td>
<td>74.3%</td>
</tr>
<tr>
<td>10</td>
<td>71.8%</td>
</tr>
</tbody>
</table>

In the limit: 69.3%
### Example 1

<table>
<thead>
<tr>
<th></th>
<th>Period $T$</th>
<th>Computation Time, $C$</th>
<th>Priority $P$</th>
<th>Utilization $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>50</td>
<td>12</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>Task_2</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>Task_3</td>
<td>30</td>
<td>10</td>
<td>3</td>
<td>0.33</td>
</tr>
</tbody>
</table>

![Graph showing task execution over time]

- **Task_1**
- **Task_2**
- **Task_3**

- **Process Release Time**
- **Process Completion Time - Deadline met**
- **Deadline Missed**

- **Executing**
- **Preempted**
Example 2

<table>
<thead>
<tr>
<th></th>
<th>Period $T$</th>
<th>Computation Time, $C$</th>
<th>Priority $P$</th>
<th>Utilization $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>80</td>
<td>32</td>
<td>1</td>
<td>0.400</td>
</tr>
<tr>
<td>Task_2</td>
<td>40</td>
<td>5</td>
<td>2</td>
<td>0.125</td>
</tr>
<tr>
<td>Task_3</td>
<td>16</td>
<td>4</td>
<td>3</td>
<td>0.250</td>
</tr>
</tbody>
</table>
### Example 3

<table>
<thead>
<tr>
<th>Period $T$</th>
<th>Computation Time, $C$</th>
<th>Priority $P$</th>
<th>Utilization $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>80</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>Task_2</td>
<td>40</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Task_3</td>
<td>20</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

How far can we continue this visual test?
Schedulability Tests

- Utilization based tests are not exact and can not be generalized easily
- Response Time Analysis
  - Response Time of process $i$, $R_i$

\[
R_i = C_i + I_i
\]

- For the highest priority process, its worst-case response time is equal to its own computation time.
- For other processes, however, it is a function of the interference from other processes.
- The maximum interference is bounded.
Response Time Analysis

\[ Number\_Of\_Releases = \left\lfloor \frac{R_i}{T_j} \right\rfloor \]

\[ Maximum\_Interference = \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j \]

\[ I_i = \sum_{j \in h\_p(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j \]

where \( h\_p(i) \) is the set of higher priority processes (than \( i \))
Response Time Analysis

• Therefore,

\[ R_i = C_i + \sum_{j \in hp(i)} \left( \frac{R_i}{T_j} \right) C_j \]

• This can be solved by a recurrent relation.

\[ w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left( \frac{w_i^n}{T_j} \right) C_j \]

for i in 1..N loop -- for each process in turn
  n := 0
  w_i^n := C_i
  loop
    calculate new \( w_i^{n+1} \) from Equation
    if \( w_i^{n+1} = w_i^n \) then
      \( R_i := w_i^n \)
      exit \{value found\}
    end if
    if \( w_i^{n+1} > T_i \) then
      exit \{value not found\}
    end if
    n := n + 1
  end loop
end loop
Example

<table>
<thead>
<tr>
<th></th>
<th>Period</th>
<th>Computation Time, $C$</th>
<th>Priority $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Task_2</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Task_3</td>
<td>20</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

- **T1** has a response time of 3 OK
- **For T2**
  - Response time $= 3 + \lceil \frac{3}{7} \rceil \times 3 = 6$
  - Response time $= 3 + \lceil \frac{6}{7} \rceil \times 3 = 6$ OK
- **For T3**
  - Response time $= 5 + \lceil \frac{5}{7} \rceil \times 3 + \lceil \frac{5}{12} \rceil \times 3 = 11$
  - Response time $= 5 + \lceil \frac{11}{7} \rceil \times 3 + \lceil \frac{11}{12} \rceil \times 3 = 14$
  - Response time $= 5 + \lceil \frac{14}{7} \rceil \times 3 + \lceil \frac{14}{12} \rceil \times 3 = 17$
  - Response time $= 5 + \lceil \frac{17}{7} \rceil \times 3 + \lceil \frac{17}{12} \rceil \times 3 = 20$
  - Response time $= 5 + \lceil \frac{20}{7} \rceil \times 3 + \lceil \frac{20}{12} \rceil \times 3 = 20$ OK
Exercise

- Try response time analysis for the process set

<table>
<thead>
<tr>
<th></th>
<th>Period $T$</th>
<th>Computation Time, $C$</th>
<th>Priority $P$</th>
<th>Utilization $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>80</td>
<td>40</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>Task_2</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>Task_3</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Not priority Driven: Latest Release Time (LRT)

• If the goal is only to meet deadlines, there is no advantage to completing any job sooner than necessary.
  – Sometimes we may want to postpone execution of a job for other reasons (power, response time)

• LRT or reverse EDF
  – Treat release times as deadlines and deadlines as release times
  – Schedule jobs backwards starting from the latest deadline of all jobs to the current time
    • Later the release time, the higher the “priority”
    • Can leave processor idle when there are jobs ready for execution (hence not a priority-driven algorithm)
  – LRT is optimal under the same conditions as EDF.
Least Slack-Time First (LST) or Maximum Laxity First (MLF)

- At time, $t$, laxity of a job = deadline $d - t -$ time to complete remaining portion of job.

- LST/MLF assigns priority based on least slack

- It is also optimal in the same sense as EDF and LRT

- The difference is that LRT needs to know execution time of tasks.

- Finally, optimality vanishes if the tasks are non-preemptive – Or multiple processors.
Deadline Monotonic Priority Assignment

• Sporadic processes: period now provides a minimum (or average) bound on their arrival
  – $T = 20$ ms is guaranteed not to arrive more than once in any 20 ms interval
  – Example: a error handling routine, with timing derived from a fault model
  – Assuming Deadline same as Period is no longer reasonable.

• DMPO: Fixed priority of a process is inversely proportional to its deadline

<table>
<thead>
<tr>
<th>Task</th>
<th>Period $T$</th>
<th>Deadline $D$</th>
<th>Comp Time, $C$</th>
<th>Priority $P$</th>
<th>Response Time, $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_1</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Task_2</td>
<td>15</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Task_3</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Task_4</td>
<td>20</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>
Analysis

- **Response time analysis**
  - *Works perfectly for values of $D < T$*
    - as long as the stopping criterion is changed to $w_i > D_i$ (instead of equality)
    - *Determine response time for a given priority ordering*

- **Deadline Monotonic Priority Ordering or DMPO is optimal**
  - *That is, if for any process set that is schedulable by priority scheme, it is also schedulable by DMPO*
Process Interactions and Blocking

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Execution Seq</th>
<th>Release Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_4$</td>
<td>4</td>
<td>EEQVE</td>
<td>4</td>
</tr>
<tr>
<td>$L_3$</td>
<td>3</td>
<td>EVVE</td>
<td>2</td>
</tr>
<tr>
<td>$L_2$</td>
<td>2</td>
<td>EE</td>
<td>2</td>
</tr>
<tr>
<td>$L_1$</td>
<td>1</td>
<td>EQQQQE</td>
<td>0</td>
</tr>
</tbody>
</table>

Diagram showing the execution timeline and process interactions with different states: Executing with Q locked, Executing with V locked, Preempted, Executing, Blocked.
Priority Inversion and Inheritance

- Priority inversion
  - A higher priority process L4 waits for a lower priority process (because of resource locks)
  - Result of fixed priority scheme
- Priority inheritance
  - If a process p is suspended waiting for process q then the priority of q becomes that of p
    - L1 will have priority of L4
Response Time Calculations with Blocking

- Very similar to the non-blocking case

\[ R = C + B + I \]

\[ R_i = C_i + B_i + \sum_{j \in hp(i)} \left( \frac{R_i}{T_j} \right) C_j \]

- With priority inheritance

\[ B_i = \sum_{k=1}^{K} usage(k, i) CS(k) \]

Where usage is a 0/1 function: if resource k is used by at least one process with priority less than i and at least one process with priority greater than or equal to i.
Dynamic Planning-Based Approaches

- Flexibility of dynamic approaches + predictability of approaches that check for feasibility
- On task arrival, before execution begins
  - attempt made to create schedule that contains previously admitted tasks and the new arrival
  - if attempt fails, alternative actions
Dynamic Best Effort Approaches

- Task could be preempted any time during execution
- Don’t know whether timing constraint is met until the deadline arrives, or the task finishes
Other Scheduling Issues

- Scheduling with fault-tolerance constraints
- Scheduling with resource reclaiming
- Imprecise computations
Scheduling with Fault-tolerance Constraints

• Example: deadline mechanism to guarantee that a primary task will make its deadline if there is no failure, and an alternative task (of less precision) will run by its deadline in case of failure
  – if no failure, time set aside for alternative task is reused

• Another approach: contingency schedules embedded in primary schedule, and triggered when there is a failure
Scheduling with Resource Reclaiming

- Variation in tasks’ execution times
  - some tasks may finish sooner than expected
- Task dispatcher can reclaim this time and utilize it for other tasks
  - e.g. non-real-time tasks can be run in idle slots
  - even better: use it to improve guarantees on tasks that have timing constraints

Next Lecture: Implementing RTOS