So far in CS4271...

- Functionality analysis
- Modeling, Model Checking
- Timing Analysis
- Software level – WCET analysis
- System level – Scheduling methods – Today!

Real-time Embedded Systems

- Embedded systems that monitor, respond to, or control external environment under timing constraints
- Examples:
  - Vehicles (car, aircraft, …)
  - Traffic control (highway, air, railway, …)
  - Process control (power plant, chemical plant, …)
  - Medical systems (radiation therapy, …)
  - Telephone, radio, satellite communication
  - Computer games

Characteristics

- Timing constraints / deadline
- Functional and temporal correctness
- Hard deadline
  - Must always meet deadline
  - Air traffic controller
- Soft deadline
  - Must frequently meet deadline
  - MPEG decoder

Tasks

- A task is a block of code executed in a CPU in a sequential fashion.
- Several independent tasks may be executing on the same CPU
- How to schedule them?
- Today’s lecture
- There might also be dependences among tasks, captured by a task graph
- Task mapping – which task on which CPU?
- Task scheduling – in what order to run the tasks mapped to same CPU?

Why study scheduling?

- Increase CPU utilization or other metrics
- For real-time systems requiring hard guarantees
  - Study in advance whether all tasks can be scheduled without missing any deadlines.
  - Need computation time of each task
    - Typically given as a worst-case bound, called the Worst-case Execution Time (WCET)
    - How to derive these WCET bounds?
- Discussed in earlier 2 lectures.
Informally, scheduling is

- Assume that we are given a task graph $G=(V,E)$.
- **Def.** A **schedule** $s$ of $G$ is a mapping $V \rightarrow T$ of a set of tasks $V$ to start times from domain $T$.

![Graph of G=(V,E)](image)

Typically, schedules have to respect a number of constraints, incl. resource constraints, dependency constraints, deadlines.

**Scheduling** = finding such a mapping.

Scheduling to be performed several times during ES design (early rough scheduling as well as late precise scheduling).

More precisely,

- **Schedule**
  - An assignment of tasks to the processor (assuming 1 processor) over time.
- **Feasible schedule**
  - All tasks can be completed and all constraints (precedence, resource, deadline) can be respected.
- **Scheduling Algorithm**
  - A recipe for producing schedules
- **Schedulability**
  - If at least one scheduling algorithm producing a feasible schedule exists.

![Diagram of schedule and tasks](image)

Periodic and Aperiodic

- **Def.** Tasks which must be executed once every $p$ units of time are called **periodic** tasks. $p$ is called their period. Each execution of a periodic task is called a **job**.
- All other tasks are called **aperiodic**.
- **Def.** Tasks requesting the processor at unpredictable times are called **sporadic**, if there is a minimum separation between the times at which they request the processor.

![Diagram of periodic task](image)

Periodic Task

- Activated on a regular basis between fixed interval
  - scan the airspace every 3 sec
  - $P = (s, c, p, d)$
- $s$ = start time or arrival time
- $c$ = worst case execution time (WCET)
- $p$ = period or cycle time
- $d$ = deadline
- $c \leq d \leq p$

![Diagram of periodic task parameters](image)

Periodic Task (Contd.)

- **Period**: interval between task activations.
- **Computation time**: Typically the WCET.
- **Initiation time**: time at which task becomes ready.
- **Deadline**: time at which process must finish.
- In most cases, $d = p$
Preemptive and non-preemptive scheduling

- Non-preemptive schedulers: Tasks are executed until they are done. Response time for external events may be quite long.
- Preemptive schedulers: To be used if some tasks have long execution times or if the response time for external events to be short.

Dynamic/online scheduling

- Dynamic/online scheduling: Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.

Static/offline scheduling

- Static/offline scheduling: Scheduling taking a priori knowledge about arrival times, execution times, and deadlines into account. Dispatcher allocates processor when interrupted by timer. Timer controlled by a table generated at design time.

Schedulability

- Set of tasks is schedulable under a set of constraints, if a schedule exists for that set of tasks & constraints.
- Exact tests cannot be efficiently computed in many situations.
- Sufficient tests: sufficient conditions for schedule checked. (Hopefully) small probability of indicating that no schedule exists even though one exists.
- Necessary tests: checking necessary conditions. Used to show no schedule exists. There may be cases in which no schedule exists & we cannot prove it.

To summarize

- Input to Scheduling Algorithm
  - One or more tasks
  - Activation time, execution time, deadline for each process
- Scheduling algorithm: a policy to allocate tasks to the processor(s)
- Feasible schedule if the scheduling algorithm can meet all the constraints
- Optimal algorithm: A scheduling algorithm that produces a feasible schedule if it exists
Organization of Scheduling slides

- Real-time Systems
- Basics of Scheduling
- Periodic Scheduling Methods
  - RMS
  - EDF

Recap: Why study scheduling?

- Increase CPU utilization or other metrics
- For real-time systems requiring hard guarantees
  - Study in advance whether all tasks can be scheduled without missing any deadlines.
- Need computation time of each task
  - Typically given as a worst-case bound, called the Worst-case Execution Time (WCET)
  - How to derive these bounds?

Recap: Periodic tasks

- A task is a program!
- Activated on a regular basis between fixed interval
  - scan the airspace every 3 sec
- \(P = (s, c, p, d)\)
  - \(s\) = start time or arrival time
  - \(c\) = worst case execution time (WCET)
  - \(p\) = period or cycle time
  - \(d\) = deadline
  - \(c \leq d \leq p\)

Task Execution

Recall: Periodic tasks

- A task is a program!
- Activated on a regular basis between fixed interval
  - scan the airspace every 3 sec
- \(P = (s, c, p, d)\)
  - \(s\) = start time or arrival time
  - \(c\) = worst case execution time (WCET)
  - \(p\) = period or cycle time
  - \(d\) = deadline
  - \(c \leq d \leq p\)

Priority driven scheduling

- Each task has a priority.
- CPU goes to highest-priority task that is ready.
- Priorities determine scheduling policy:
  - fixed priority;
  - time-varying priorities.
- Rules:
  - Each task has a fixed priority (1 highest);
  - highest-priority ready task gets CPU
  - task continues until done (non pre-emptive) OR
  - Task can be pre-empted by a later arriving higher priority task.

Example (preemptive)

- \(P1\): priority 1, execution time 10
- \(P2\): priority 2, execution time 30
- \(P3\): priority 3, execution time 20

\(P3\) ready \(t=18\)
\(P2\) ready \(t=0\) \(P1\) ready \(t=15\)

\(P2\) | \(P1\) | \(P2\) | \(P3\)
-----|-----|-----|-----
0 | 10 | 20 | 30 | 40 | 50 | 60 | time

Copyright 2009 by Abhik Roychoudhury
Rate-monotonic scheduling

- RMS (Liu and Layland 1973)
  - widely-used, analyzable scheduling policy.
  - Analysis is known as Rate Monotonic Analysis
  - RMS is an optimal fixed priority assignment method
    - If there exists a schedule that meets all the deadlines with fixed priority, then RMS will produce a feasible schedule.
    - Fixed-priority, pre-emptive scheduling.

Assumptions in RMS

- All tasks run on single CPU.
- Zero context switch time.
  - If not, the context switch time needs to be added in response time computation.
- No data dependencies between tasks.
- Task execution time is constant.
- Deadline is at end of period (p = d)
- Highest-priority ready task runs.

Priority assignment in RMS.

- Optimal (fixed) priority assignment:
  - shortest-period task gets highest priority;
  - priority inversely proportional to period;
  - breaks ties arbitrarily.
- Intuition: Tasks requiring frequent attention (smaller period) should receive higher priority.

RMS Example

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Period</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Rate monotonic analysis

- Response time: time required to finish task.
- Critical instant: scheduling state that gives worst response time.
- Critical instant occurs when all higher priority tasks are ready to execute.
- Worst case response time
  - Solved by iterative computation
  - \( w_i = c_i + \sum_{j<i} c_j \cdot \left\lfloor \frac{w_i}{p_j} \right\rfloor \)

Critical Instant
Informal argument about optimality

- P1 = (c1, p1, d1) with p1 = d1
- P2 = (c2, p2, d2) with p2 = d2
- p1 < p2
- Suppose P1 and P2 can be scheduled with non-RM priority assignment, i.e., P2 has highest priority
- At critical instant, with non-RM priorities
  - c1 + c2 ≤ p1; [1]
- With RM priority
  - \[ \lceil p2/p1 \rceil \cdot c1 + c2 ≤ p2; [2]\]
- If [1] is satisfied, then [2] is also satisfied

RMS optimality

- RMS CPU utilization
  - Utilization for n processes is
    - U = Σ ci / pi
  - U ≤ 1 is a necessary condition for feasibility regardless of scheduling policy
  - Scheduling with fixed priorities is feasible if U ≤ n \( \frac{1}{m-1} \)
  - The bound is sufficient but not necessary.
  - As number of tasks approaches infinity, maximum utilization approaches 69%.
  - RMS cannot use 100% of CPU even with zero context switch overhead.
  - Must keep idle cycles available to handle worst-case scenario.

Earliest-Deadline-First

- EDF: dynamic priority scheduling scheme.
- Task closest to its deadline has highest priority.
- Requires recalculating tasks at every timer interrupt.
- EDF can use 100% of CPU.
- Implementation
  - On each timer interrupt:
    - compute time to deadline;
    - choose task closest to deadline.
  - Generally considered too expensive to use in practice.

EDF Example

- P1 = (2, 4, 4) P2 = (5, 10, 10)

EDF Properties

- EDF is optimal
  - If a feasible schedule exists using dynamic priorities, then EDF will produce a feasible schedule
  - EDF can always produce a feasible schedule if U ≤ 1
- Scheduling with dynamic priority is feasible if and only if U ≤ 1
Fixing scheduling problems …

- ... in practice.
- What if your set of tasks is not schedulable?
  - Change deadlines in requirements.
  - Reduce execution times of processes.
  - Get a faster CPU.

Organization of Timing Analysis

- Software timing analysis - Completed!
- WCET analysis
- System level analysis – Completed!
- Scheduling methods
- Design issues to improve timing predictability
- Scratchpad memories

Scratchpad Memory

- Compiler controlled memory.
  - Unlike cache, compiler has control over its contents.
  - We can statically decide what to put in the scratchpad and lock it in.

- So, is it simply a statically locked cache?
  - No, it is a bit more!

Cache locking and scratchpad

How scratchpad memory works?

Combination with cache is also possible.
Scratchpad allocation strategy

- Assume a scratchpad memory for data variables.
- We need to statically decide which variables to allocate in scratchpad memory.
  - Variables = \{v_1, \ldots, v_n\}
  - Say \(n_i\) is # of times \(v_i\) is executed in a given execution path.
  - Define \(\text{gain}_i = n_i \cdot (N - 1)\)
  - \(\text{gain}_i\) is gain from allocating \(v_i\) to scratchpad.
  - \(N\) = number of cycles needed to access main memory.
  - \(\text{gain}_i\) is constant.

Variables = \{v_1, \ldots, v_n\}

\(n\) = # of times \(v\) is executed in a given execution path

\(\text{gain}_i\) = gain from allocating \(v_i\) to scratchpad.

\(N\) = number of cycles needed to access main memory.

\(\text{gain}_i\) is constant.

\(\sum_{1 \leq i \leq n} \text{choice}_i \cdot \text{gain}_i\)

Subject to

\(\sum_{1 \leq i \leq n} \text{choice}_i \cdot w_i \leq \text{Capacity}\)

\(\text{Gain} = \text{gain}_i\)

\(\text{choice}_i\) is 0 or 1

0 if \(v_i\) is not allocated

1 if \(v_i\) is allocated.

The gain is not constant

- Define \(\text{gain}_i = n_i \cdot (N - 1)\)
- \(\text{gain}_i\) = gain from allocating \(v_i\) to scratchpad.
- \(N\) = number of cycles needed to access main memory.
- Say \(n_i\) is # of times \(v_i\) is executed in a given execution path.
- \(n_i\) is constant.

Which execution path?

What if we want to allocate to scratchpad memory for reducing the program’s WCET?

Knapsack problem

- Given \(n\) objects and a knapsack
  - Capacity of knapsack \(W\)
  - Object \(i\) has weight \(w_i\) and value \(\text{gain}_i\)
  - Fill up the knapsack so as to maximize value

Perfect fit for our allocation problem if the gain by allocating a variable to scratchpad memory is a constant

Holds for ACET based allocation

Not true for WCET based allocation

Knapsack problem can be easily solved by dynamic programming.

Knapsack

- Array \(V[0..n,0..W]\)
- \(V[i,j] = \text{maximum value if we are restricted to objects 1..i, and the weight limit is } j\)

Define

- \(V[i,j] = \max(V[i-1,j], V[i-1,j-w_i] + v_i)\) for \(i > 0\)
- \(V[0,j] = 0\)

Final Answer \(V[n,W]\)

This solution is useful only for ACET based allocation.

ACET vs WCET based

WCET (Worst-case) = Maximum exec. time of a pgm. for all possible inputs

In this simple example assume

- \(U\) and \(V\) have same size
- Only one of them can be allocated
- ACET savings depends on the execution counts of Path1 and Path2 (whichever is the more frequent path forms our profile)
- WCET savings depend on which path is longer
Difficulty in WCET based allocation

Allocate V

Path1 Path2
Before 90 100
After 90 80

Path2 is now the WCET path which has different var. access frequencies from Path2

Before After
90 100
90 80

Contribution of V to WCET path = 20
Reduction in WCET by allocating V = 10

Sub-optimal allocation

V, U both appear in WCET path. Suppose \#V > \# U

Before After
90 100
90 80

Optimal allocation = {U}

Before After
90 100
75 85

WCET based allocation

× gain in WCET reduction due to a variable v
  • not a constant, depends on
    • current WCET path
    • Diff in exec. Times of WCET path and other paths
    • # Occurrence of v in WCET path and other paths

WCET-based allocation

× gain in WCET reduction due to a variable allocation:
  • not a constant
  • not cumulative
    • \( \text{gain}_v \leq \text{gain}_u + \text{gain}_v' \)

Summary: WCET-based allocation

Optimal WCET based allocation
  • Must not be profile-guided (WCET path).
  • Cannot take WCET contribution of variables as constant.
  • Rules out Knapsack like solutions.
  • ACET-based solution (where gain, is constant) is a well-known knapsack problem with known algorithmic solutions via dynamic programming.

Overall Summary of Timing Issues

• Correctness of many embedded systems depend on their timing behavior.
• Analysis /Validation Methods
  • Software level
    • WCET analysis – fine-grained
  • System level
    • Scheduling methods – use WCET estimates.
• Make the system more time predictable and easier to analyze
  • Scratchpad memories are one such solution.