Contents

1 Introduction

Embedded System: Information processing systems embedded into a larger product. Goal is not information processing! Information processing is the way to reach a certain goal.

Hybrid systems: The analog and digital system together.

CPS: cyber-physical system. When a system connects the physical world with the digital.

Reactive system: A reactive system is one which is in continual interaction with its environment and executes at a pace determined by that environment.

Hard real-time constraint: real-time constraint is called hard, if not meeting that constraint could result in a catastrophe

Soft real-time constraint: All non-hard real-time constraints.

Guaranteed system response: A guaranteed system response has to be explained without statistical arguments.

1.1 Where used?

- Logistics
- Health Care
- Building Automation
- Factory Automation
- Cars
- Consumer Electronics (Microwave)

1.2 Characteristics of ES

- dependable (zuverlässig)
  - Reliability: $R(t)$ is the probability of system working correctly provided that it was working at $t = 0$
  - Maintainability: $M(d)$ is the probability of system working correctly d time units after error occurred
  - Availability: probability of system working at time t
  - Safety: no harm to be caused
  - Security: confidential and authentic communication
- efficient
  - Energy efficient

  - Code-size efficient (especially for systems on a chip)
  - Run-time efficient
  - Weight efficient
  - Cost efficient
- Dedicated towards a certain application. Makes no sense, to make the system faster than required
- Dedicated user interface
- real-time constraints: For real-time systems, right answers arriving too late are wrong.
- A guaranteed system response has to be explained without statistical arguments

1.3 ES vs. General Purpose System

Embedded Systems	General Purpose System
Few applications that are known at design-time	Broad class of applications.
Not programmable by end user	Programmable by end user.
Fixed run-time requirements (additional computing power not useful).	Faster is better.
Criteria: • cost • power consumption • predictability	Criteria: • cost • power consumption • average speed
Hard and Soft deadlines, periods...	Tasks do have timing constraints
Goal: Meet the individual timing requirements	Goal: Minimize the average response time, without any guarantees

2 Software Introduction

2.1 Real-Time Systems

Worst-case execution time - WCET: The upper bound on the execution times of all tasks statistically known.

Best-Case execution time - BCET: The lower bound on the execution times of all tasks statistically known.

Hard to determine execution time

- Use of caches, pipelines, branch prediction, speculation makes prediction impossible
- Measurements: determine execution time directly by observing the execution or a simulation on a set of inputs - does not guarantee an upper bound to all executions!
- Exhaustive execution in general is not possible!
- Compute timing along the structure of a program: check dependencies of instructions and sum up all worst-case-times
- In general not possible to compute execution time (halting problem)!

2.2 Time-Triggered Systems

Can be implemented as a set of methods which are called each after another. A timer is set to interrupt, when the period P ends.

2.2.1 Informations

- No interrupts except by timer
- Interaction with environment through polling - can be a problem if they are too long! Split them up in shorter processes
- Schedule computed off-line - complex algorithms can be used
- Deterministic behavior at run-time
- Inflexible, no adaption to environment

2.3 Example - Simple periodic time-triggered Scheduler

- Timer interrupts regularly with period P
- All processes have same period P
- Properties:
  - later processes (T_2, T_3) have unpredictable times

  - No problem with communication because static ordering and no parallelism

Theorem: $\sum_{(k)} WCET(T_k) < P$

2.4 Example - Time-triggered cyclic executive Scheduler

- Processes may have different periods
- The period P is partitioned into frames of length f
- Long processes must be partitioned into a sequence of small processes - must store the program state!

2.4.1 Conditions

- A process executes at most once within a frame: $f \leq p(k) \forall k$
- Period P is least common multiple of all periods $p(k)$
- Processes start and complete within a single frame: $f \geq WCET(k) \forall k$
- Between release time and deadline of every task there is at least one frame boundary: $2f - gcd(p(k), f) \leq D(k) \forall k$

2.4.2 Properties

- No RTOS needed
- No context-switch overhead because there is no preemption
- Just a few scheduling overhead $\rightarrow$ can use a cheap processor
- Development of a time triggered cyclic executive scheduler normally harder due to off-line scheduling
- Not flexible, if later on a new process is added - need to compute everything from beginning

2.5 Event triggered Systems

- The schedule of processes is determined by the occurence of external interrupts
- Dynamic and adaptive: there are possible problems with respect to timing - problem when using buffers or shared ressources!
- Guarantees can be given either off-line or during run-time

2.6 Example - Non-preemtive event triggerd scheduling

2.6.1 Principle

- To each event, there is accosicated a corresponding process that will be executed
- Events are emitted by a) external interrupts and b) by processes themselves
- Events are collected in a queue; depending on queueing discipline, an event is chosen for running
- Processes can not be preemted

2.6.2 Properties

- Communication simple; no parallelism - interrupts may cause problems with shared resources (for example the process queue)
- Buffer overflow if too many events are generated by environment or processes
- Long processes must be split into smaller ones since long running time may cause buffer overflow

2.7 Example - Preemtive event triggered scheduling

Similar to non-preemtive event triggered scheduling - but processes can be preempted by others!

2.7.1 Stack Policy

- Use usual stack-based context mechanism of function calls
- This implies a LIFO order of their instantiation - unknown waiting time for external events!
- Shared resources must be protected: use semaphores

2.8 Petri Nets

- Semantics of the calls is expressed using Petri Nets
- Bipartite graph consisting of places and transitions
- Data and control are represented by moving tokens
- Token are moved by transition according to rules: A transition can fire (is enabled) if there is at least one token in every input place. After firing, one token is removed form each input place and one is added to each output place.

2.9 Process

Process: A process is a unique execution of a program. Several copies of a program may run simultaneously or at different times. The process has its own state and consists of register values, memory stack and instruction pointer.

Activation record: Copy of the process state: registers and local data structures

Context switch: Current CPU context goes out, new CPU context goes in

2.9.1 Co-operative Multitasking

- Each process allows a context switch at `cswitch()` call
- Scheduler chooses which process to run next
- Advantages: predictable where context switch occurs. Less errors of shared resources.
- Problems: Programming error can keep other threads out. Real-time behavior at risk if it takes too long before context switch allowed

2.9.2 Preemtive Multitasking

- Most powerful form of multitasking - scheduler controls when context switches and determines what process runs next
- Use of timers to call OS and switch contexts

3 Real Time Models

3.1 Classification

Preemptive algorithm: With preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.

Non-preemptive algorithm: With a non-preemptive algorithm, a task, once started, is executed by the processor until completion.

Static Algorithm: Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.

Dynamic algorithms: Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system execution

Optimal: An algorithm is said optimal if it minimizes some given cost function defined over the task set.

Heuristic: An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

3.2 General

Hard: A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control.

Soft: A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.

3.3 Schedule and Timing

Schedule: A schedule is an assignment of tasks to the processor, such that each task is executed until completion. It can be defined as an integer step function where $\sigma : \mathbb{R} \rightarrow \mathbb{N}$ denotes the task which is executed at time t.

Idle: If $\sigma(t) = 0$ then the processor is called idle. So no task is executed.

Context switch: If $\sigma(t)$ changes its value at some time.

Time slice: Interval, in which $\sigma(t)$ is constant.

Feasible: A schedule is said to be feasible, if all task can be completed according to a set of specified constraints.

schedulable: A set of tasks is said to be schedulable, if there exists at least one algorithm that can produce a feasible schedule.

3.4 Metrics

Arrival time a_i/release time r_i: Is the time at which a task becomes ready for execution (Waits in ready queue).

Computation time C_i: Is the time necessary to the processor for executing the task without interruption.

Deadline d_i: Is the absolute time at which a task should be completed.

Fair Schedule: (for the first exercise) a schedule is fair, if every task eventually gets a chance to execute on the processor.

Finishing time f_i: Is the time at which a task finishes its execution.

Lateness L_i: $L_i = f_i - d_i$, represents the delay of a task completion with respect to its deadline.

Laxity / slack time X_i: $X_i = d_i - a_i - C_i$, is the maximum time a task can be delayed on its activation to complete within its deadline.

Response time: The amount of time it takes to finish executing a task: $response_i = f_i - a_i$.

Start time s_i: Is the time at which a task starts its execution.

Tardiness / exceeding time E_i: $E_i = max(0, L_i)$, is the time a task stays active after its deadline.

Throughput: The measure of work done in a unit time interval.

Utilization: Ratio of busy time of the processor to the total time required for all tasks to finish.

Waiting time: Time spent by a task in the ready queue.

Average response time:

$$t_r = \frac{1}{n} \sum_{i=1}^{n} (f_i - r_i) \tag{1}$$

Total completition time:

$$t_c = \max_i(f_i) - \min_i(r_i) \tag{2}$$

Weighted sum of response time:

$$t_w = \frac{\sum_{i=1}^{n} w_i(f_i - r_i)}{\sum_{i=1}^{n} w_i} \tag{3}$$

Maximum lateness:

$$L_{max} = \max_i (f_i - d_i) \tag{4}$$

Number of late tasks:

$$N_{late} = \sum_{i=1}^{n} miss(f_i) \qquad miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

4 Aperiodic Tasks

event-driven, may have hard, soft, non- real-time requirements depending on the specific application

4.1 FIFO/FCFS - First in, first out/First come, first serve

4.1.1 Assumptions

- Independent tasks
- Non preemptive

4.1.2 Algorithm

Tasks are added to a FIFO data structure. Schedule only depended on the release/start time. If a task needs to execute repeatedly, each successive execution is treated as a new task.

4.2 SJF - Shortest Job First

4.2.1 Assumptions

- Independent tasks
- Non preemptive

4.2.2 Algorithm

Scheduler picks up the task with the shortest execution time from the ready queue.

4.2.3 Properties

Minimizes the average waiting time!

4.3 Shortest remaining time next

4.3.1 Assumptions

- Independent tasks
- Preemptive

4.3.2 Algorithm

As a new task arrives, the scheduler determines which of the ready tasks has the smallest execution time and executes it.

4.3.3 Properties

Attempts to minimize the average waiting time. Dynamic algorithm, so not strictly minimal.

4.4 RR - Round Robin

4.4.1 Assumptions

- Independent tasks
- Preemptive

4.4.2 Algorithm

Task preempted if it exceeds its time quantum, or when it gets done (task has finished its execution).

4.5 EDD - Earliest Deadline Due

4.5.1 Assumptions

- Independent tasks
- Equal arrival times
- Non preemptive

4.5.2 Algorithm

Jacksons rule: Given a set of n tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

4.5.3 Properties

- Minimize the maximum lateness
- Non preemptive

4.6 EDF - Earliest Deadline First

4.6.1 Assumptions

- Independent tasks
- Arbitrary arrival times
- Preemptive

4.6.2 Algorithm

Horns rule: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

Note: Worst case finishing time of task i: $f_i = t + \sum\limits_{k=1}^{i} c_k(t)$. Where $c_k(t)$ is the remaining worst-case execution time of task k.

4.6.3 Properties

- Minimize the maximum lateness
- Preemptive
- Any asynchronous task set feasible under EDD is also feasible under EDF
- For asynchronous task sets without precedence constraints, EDF* is identical to EDF

4.7 EDF* - Earliest Deadline First

4.7.1 Assumptions

- Dependent tasks
- Arbitrary arrival times
- preemptive
- Minimal development effort
- Big scheduling overhead
- RTOS required
- Context switch overhead $\rightarrow$ needs faster CPU than Time-triggered or RM
- Flexible, if later on a new process is added. High utilization possible!

4.7.2 Algorithm

For release times, start at the beginning (no predecessor):

1. For any initial node of the precedence graph set $r_i^* = r_i$
2. Elect a task j such that its release time has not been modified but the release times of all immediate predecessors i have been modified. If no such task exists, exit.
3. Set $r_j^* = max(r_j, max(r_i^* + C_i, if J_i \rightarrow J_j))$
4. Return to step 2

For modification of deadlines, start at the end (no successor)

1. For any terminal node of the precedence graph set $d_i^* = d_i$
2. Elect a task i such that its deadline has not been modified but the deadlines of all immediate successors j have been modified. If no such task exists, exit.
3. Set $d_i^* = min(d_i, min(d_j^* - C_j, if J_i \rightarrow J_j))$
4. Return to step 2

4.7.3 Properties

- The problem of scheduling a set of n tasks with precedence constraints (concurrent activation) can be solved in polynomial time complexity if tasks are preemptable.
- The EDF* algorithm determines a feasible schedule in the case of tasks with precedence constraints if there exists one.
- Minimize the maximum lateness

4.8 LDF - Latest Deadline First

4.8.1 Assumptions

- Tasks with precedence relations
- Synchronous arrival times
- Non preemptive

4.8.2 Algorithm

Build a dependency graph. Choose from all task which do not have a successor the task with the latest deadline. Extract this task from the dependency graph and add it to the scheduling stack. Repeat until the stack is empty. The schedule is according to the stack. This means the last task added to the stack is the first to be executed.

4.8.3 Properties

- Minimize the maximum lateness
- non preemptive

5 Periodic Tasks

time-driven, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates

Periodic Tasks: An infinite sequence of identical activities, called instances or jobs that are regularly activated at constant rate with period T_i. The activation time of the first instance is called phase Φ_i.

5.1 Model

Definitions

- Γ: a set of periodic tasks
- τ_i: a generic periodic task
- $\tau_{i,j}$: the jth instance of task i
- $r_{i,j}$: release time of the jth instance of task i
- $s_{i,j}$: start time of the jth instance of task i
- $f_{i,j}$: finishing time of the jth instance of task i
- $d_{i,j}$: the absolute deadline of the jth instance of task i
- Φ_i: phase of task i (release time of its first instance)
- D_i: relative deadline of task i
- T_i: periode of task i

Note: Hypotheses are

- The instances of a periodic task are regularly activated at a constant rate. $r_{i,j} = \Phi_i + (j-1)T_i$. The first task has $j = 1$.
- All instances have the same worst case execution time C_i
- All instances of a periodic task have the same relative deadline D_i. Therefore: $d_{i,j} = \Phi_i + (j-1)T_i + D_i$
- Often, the relative deadline equals the period $D_i = T_i$ and therefore $d_{i,j} = \Phi_i + jT_i$
- All periodic tasks are independent.
- No task can suspend itself
- All tasks are released as soon as they arrive.
- All overheads in the OS kernel are assumed to be zero.

The meaning of sufficient and necessary

- sufficient condition C for proposition P: $C \Rightarrow P$
- necessary condition C for proposition P: $\neg C \Rightarrow \neg P$

5.2 RM - Rate Monotonic Scheduling

5.2.1 Assumptions

- Static priority assignment: Task priorities are assigned to tasks before execution and do not change over time.
- RM is intrinsically preemptive: the currently executing task is preempted by a task with higher priority - algorithm is preemptive!
- Deadlines equal the periods $D_i = T_i$

5.2.2 Algorithm

Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Tasks with higher priority interrupt tasks with lower priority

5.2.3 Properties

- Optimality: RM is optimal among all fixed-priority assignments in the sense that not other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM
- Just a few scheduling overhead
- RTOS required
- Context-switch-overhead
- Flexible, if later on a new process is added (update priority list)

5.2.4 Theorem

Theorem: A set of periodic tasks is schedulable with RM if $\sum\limits_{i=1}^{n} \frac{C_i}{T_i} = U \leq n(2^{1/n} - 1)$. This condition is sufficient, but not necessary.

Theorem: For the necessary and sufficient test, see at Deadline Monotonic Scheduling. Note, that in this case $D_i = T_i$.

5.3 DM - Deadline Monotonic Scheduling

5.3.1 Assumptions

- Static priority assignment: Task priorities are assigned to tasks before execution and do not change over time.
- DM is intrinsically preemptive: the currently executing task is preempted by a task with higher priority
- Deadlines may be smaller than the periodic. $C_i \leq D_i \leq T_i$

5.3.2 Algorithm

Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Tasks with higher priority interrupt tasks with lower priority

5.3.3 Theorem

Theorem: A set of periodic tasks is schedulable with DM if $\sum\limits_{i=1}^{n} \frac{C_i}{D_i} = U \leq n(2^{1/n} - 1)$. This condition is sufficient, but not necessary. So if true, then it works, but if not true then we do not know anything.

Critical instance: The worst case processor demand occurs when all tasks are released simultaneously; that is at their critical instances/A critical instant for any task occurs whenever the task is released with all higher priority tasks.

Theorem: A measure of the worst case interference for task i can be computed as the sum of the processing times of all higher priority tasks released before some time t where tasks are ordered according to $m \leq n \leftrightarrow D_m \leq D_n : I_i = \sum_{j=1}^{i-1} \lceil \frac{t}{T_j} \rceil C_j$

Response Time R_i: The longest response time of a periodic task i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks: $R_i = C_i + I_i$

Theorem: There is also a necessary and sufficient schedulability test. This test computes the biggest response time R_i. Sort first the tasks accordingly. Start with the biggest i first.

```
DM_guarantee( $\Gamma$ )
{
        //$\Gamma$ is sorted in descending order of priorities.
        //So we have $D_i \leq D_{i+1}$. Start with
        //the biggest $i$ (lowest priority, biggest $D$) first
        for each ($\tau_i \in \Gamma$)
        {
                I = 0;
                do
                {
                        R = I + $C_i$;
                //Biggest Response bigger
                // than Deadline?
                if( R > $D_i$)
                        return UNSCHEDULABLE;
                I = $\sum_{j=1}^{i-1} \lceil R/T_j \rceil C_j$ ;
        }while(I + $C_i$ > R);
    }
    return SCHEDULABLE;
}
```

5.4 EDF Scheduling - Earliest Deadline First

5.4.1 Assumptions

- Dynamic priority assignment
- Intrinsically preemptive
- $D_i \leq T_i$

5.4.2 Algorithm

The currently executing task is preempted whenever another periodic instance with earlier absolute deadline $d_{i,j}$ becomes active.

5.4.3 Properties

Optimality: No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.

5.4.4 Theorem

Theorem: A necessary and sufficient schedulability test if $D_i = T_i$: A set of periodic tasks is schedulable with EDF if and only if $\sum\limits_{i=1}^{n} \frac{C_i}{T_i} = U \leq 1$.
If the utilization satisfies $U > 1$, then there is no valid schedule.
If the utilization satisfies $U \leq 1$, then there is a valid schedule

Theorem: A sufficient schedulability test if $D_i \leq T_i$: A set of periodic tasks is schedulable with EDF if $\sum\limits_{i=1}^{n} \frac{C_i}{D_i} = U \leq 1$.

5.5 Background Scheduling

Simple solution for RM and EDF scheduling of periodic tasks.

- Processing of aperiodic tasks in the background, i.e. if there are no periodic request.
- Periodic tasks are not affected.
- Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.

5.6 RM - Polling Server

5.6.1 Assumption

- An aperiodic job J_a with computation time C_a and relative deadline D_a that needs to be scheduled.
- A periodic server task with period T_s and computation time C_s

5.6.2 Algorithm

Introduce an artificial periodic task whose purpose is to service aperiodic requests as soon as possible

- a server is characterized by a period T_s and a computation time C_s
- The server is scheduled with the same algorithm used for the periodic tasks and, once active, it serves the aperiodic requests within the limit of its server capacity.
- Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.

5.6.3 Properties

- Aperiodic guarantee of aperiodic activities
- Disadvantage: If an aperiodic requests arrives just after the server has suspended, it must wait until the beginning of the next polling period.

5.6.4 Theorem

Theorem: A set of periodic tasks and a server task can be executed within their deadlines if $\frac{C_s}{T_s} + \sum\limits_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1)(2^{1/(n+1)} - 1)$. This test is sufficient but not necessary.

Theorem: If we have the assumption that an aperiodic task is finished before a new aperiodic request arrives. Assume, we have the computation time C_a, deadline D_a. A sufficient schedulability test is $(1 + \left\lceil \frac{C_a}{C_s} \right\rceil)T_s \leq D_a$

Theorem: A necessary and sufficient test can be given by applying the algorithm at Deadline Monotonic Scheduling with $D_i = T_i$.

Note: To compute a smaller value D_a of the deadline of the aperiodic task, we can compute R_a with the necessary and sufficient test, and set $D_a = R + T_S$. (Because the task can arrive just after the start of the polling server and we have to add the response time to it).

5.7 EDF - Total Bandwidth Server

5.7.1 Assumptions

-

5.7.2 Algorithm

- When the kth aperiodic request arrives at time $t = r_k$, it receives a deadline $d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s}$ where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition, $d_0 = 0$.
- Once a deadline is assigned, the request is inserted into the ready queue of the system as any other periodic instance
- Use the EDF algorithm to schedule all tasks

5.7.3 Theorem

Theorem: Given a set of n periodic tasks with processor utilization U_p and a total bandwidth server with utilization U_s, the whole set is schedulable by EDF if and only if $U_p + U_s \leq 1$

6 Resource sharing

6.1 Terms

Mutual Exclusion: The requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions.

Critical section: A piece of code executed under mutual exclusion constraints.

Blocked: A task waiting for an exclusive resource.

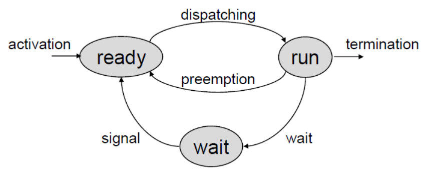


Figure 1: By preemption of the lock-holder, the blocked process goes into the ready pool. There it is decided, which thread to run next.

6.2 Solutions to Resource Sharing

- Non-preemptive tasks
- Disable interrupts
- Static scheduling
- Use of semaphores

6.3 Priority Inversion

6.3.1 Problem

A task with a middle priority J_2 can get the CPU for an arbitrary time, although a task with highest priority J_1 is waiting, caused by the lockholder J_3 with lowest priority!

6.3.2 Solution - Disallow preemption

Disallow preemption during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked. Deadlines now must become longer.

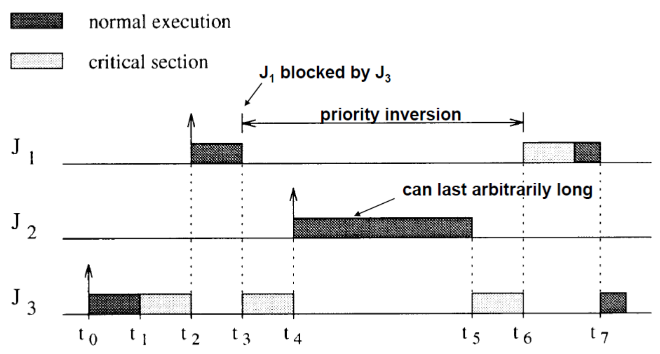


Figure 2: Example of priority inversion

6.3.3 Solution - Priority Inheritance Protocol (PIP)

Idea: Modify the priority of those tasks that cause blocking. When a task J_i blocks one or more higher priority tasks, it temporarily assumes a higher priority.

Assumption:

- n tasks which cooperate through m shared resources
- fixed priorities

Algorithm:

1. Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a FCFS discipline.
2. When a job J_i tries to enter a critical section and the resource is blocked by a lower priority job, the job J_i is blocked.
3. When a job J_i is blocked, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$
4. When J_k exits a critical section, it unlocks the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
5. Priority inheritance is transitive

Deadlock can still occur, for example when the following is executed with J_2 higher priority than J_1: J_1: wait(S_a); - J_2: wait(S_b); wait(S_a); - J_1: wait(S_b);

7 RTOS: Real-Time OS

Real-time Operating System: A real-time operating system is an operating system that supports the construction of real-time systems.

7.1 Desktop OS not suited

- Monolithic kernel too feature rich. Not modular, configurable, modifiable
- Too much space
- Not power optimized
- Timing uncertainty too large

7.2 Properties of RTOS/Embedded OS

- Configurability: no single RTOS will fit all needs. Reduce overhead. Examples: Remove unused functions at compile time
- Device drivers handled by tasks. Meaning everything goes through the scheduler (device drivers rely on the kernel)
- Interrupts employed by any process. Time to handle interrupts need to be considered
- Protection mechanism not always necessary. ES designed for a single purpose, rarely untested programs - SW considered reliable. Task do their own IO instructions instead of kernel call.

7.3 Requirements

7.3.1 Timing Behavior

The timing behavior of the OS must be predictable. For all services of the OS, there has to be an upper bound on the execution time. RTOS must be deterministic.

7.3.2 Timing

OS must manage the timing and scheduling. Must be aware of deadlines and provide precise time services with high resolution.

7.3.3 Fast

The OS must be fast. In general, practically important.

7.4 Functionality of RTOS-Kernels

- Execution of quasi-parallel tasks on a processor using processes or threads (lightweight process). Providing and maintaining process states, process queuing, preemptive tasks, fast context switching, quick interrupt handling
- CPU scheduling (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- Process synchronization (critical sections, semaphores, monitors, mutual exclusion)
- Inter-process communication (buffering)
- Support of a real-time clock as an internal time reference

7.5 Context-Switch

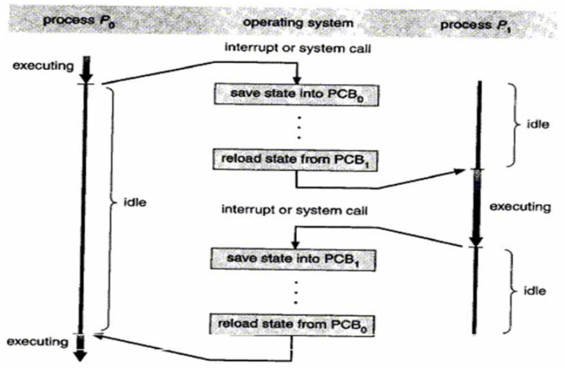


Figure 3: Context switch from process 0 to process 1 and back

7.6 Process states

- Run: A task enters this state as it starts executing on the processor
- Ready: State of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task
- Wait: A task enters this state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head is resumed when the semaphore is unlocked by a signal primitive.
- Idle: A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period

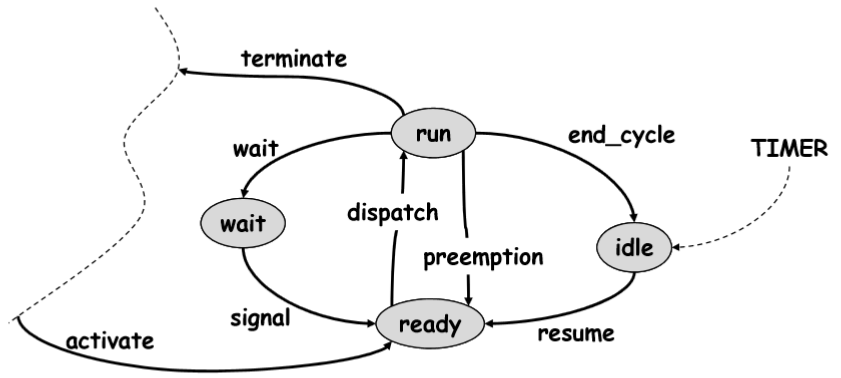


Figure 4: State diagram of process states.

7.7 Threads

Thread: A thread is an execution stream within the context of a thread state; e.g., a thread is a basic unit of CPU utilization

- key difference between processes and threads: multiple threads share parts of their state (shared memory, own register and stack)
- Faster to switch between threads (user-level threads)
- an application will have a separate thread for each distinct activity
- Thread Control Block (TCB) stores information needed to manage and schedule a thread

7.7.1 Process Management

- Typical os: synchronization and mutual exclusion is performed via semaphores and monitors
- In real-time OS, special semaphores and a deep integration into scheduling is necessary (priority inheritance protocols)
- Initializations of internal data structures (tables, queues, task description blocks, semaphores)

7.8 Synchronous Communication

- rendez-vous

- Have to wait for each other
- Problem in dynamic RTS: Estimating the maximum blocking time for a process rendez-vous
- In static RTS: the problem can be solved off-line by transforming all synchronous interactions into precedence constraints

7.9 Asynchronous Communication

- More suited for real-time systems
- Tasks do not have to wait for each other.
- sender just deposits its message into a channel and continues its execution
- receiver can directly access the message if at least a message has been deposited into the channel
- Shared memory buffer, FIFO-queue. Often has fixed capacity
- Problem: Blocking behavior if channel is full or empty. Solved by cyclic (overriding) buffers

7.10 Class of Kernels

7.10.1 Class 1 - Fast Proprietary Kernels

For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect. Examples: VxWORKS

7.10.2 Class 2 - Extensions to Standard OSs

Real-time extensions to standard OS: Attempt to exploit comfortable main stream OS. RT-kernel running all RT-tasks. Standard-OS executed as one task. Pro: Crash of standard-OS does not affect RT-tasks. Con: RT-tasks cannot use Standard-OS services; less comfortable than expected

7.10.3 Class 3 - Research Systems

- trying to avoid limitations of overhead memory protection
- temporal protection of computing resources
- quality of service (QoS) control
- formally verified kernel properties

8 System Components

8.1 General Purpose Processors

8.1.1 Properties

- High Performance: Use of parallelism, optimized circuits and technology
- Complex memory architecture
- Not suited for real-time applications: times are highly unpredictable due to intensive resource sharing and dynamic decisions
- Good average performance
- High power consumption

8.1.2 Multicore Processors

Properties

- Higher execution performance by parallelism
- useful in high-performance embedded systems (autonomous driving)
- Disadvantages and problems for embedded systems:
  - Increased interference on shared resources (buses, shared caches)
  - Increased timing uncertainty
  - Often no parallelism in embedded applications

Examples

- Xeon Phi
- Oracle Sparc

Domains

- Image and Audio processing
- Signal Processing
- Scientific computing
- Control

8.2 System Specialization

Main difference between general purpose highest volume microprocessors and embedded systems is specialization.

8.2.1 Flexibility

- application domain specific systems shall cover a class of applications
- some flexibility is required to account for late changes, debugging

8.2.2 Examples

- Code-size efficiency: RISC designed for run-time efficiency. Many instructions for simple job
- Heterogeneous registers: Different functionalities for each register vs. general purpose register
- Specified memory banks

8.2.3 Microcontroller

- for control-dominant applications
- Short latency times
- Low power consumption
- Suited for real-time applications

8.3 Digital Signal Processors/VLIW

8.3.1 General

- Streaming oriented systems with mostly periodic behavior
- Application examples: signal processing, control engineering
- use of parallel hardware units (VLIW)
- high data throughput
- specialized memory
- suited for real-time applications

8.3.2 Very Long Instruction Word - VLIW

Key idea: detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient). VLIW: parallel operations (instructions) encoded in one long word (instruction packet), each instruction controlling one functional unit.

8.4 Programmable Hardware - FPGA

- Granularity of logic units: Gate, tables, memory, functional blocks (ALU, control, data path, processor)
- Communication network: Crossbar, hierarchical mesh, tree
- Reconfiguration: fixed at production time, once at design time, dynamic during run-time

8.5 Application Specific Circuits - ASICs

- Custom-designed circuits necessary if ultimate speed or energy efficiency is the goal and large numbers can be sold
- Problems: long design times, lack of flexibility (changing standards) and high costs

8.6 System on a Chip

A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all sniper components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functionsall on a single chip substrate. SoCs are very common in the mobile electronics market because of their low power consumption. A typical application is in the area of embedded systems.

9 Communication

9.1 Requirements

- Performance: bandwith, latency, guaranteed behaviour (real-time)
- efficient: cost (material, installation), low power
- Robustness: fault tolerance, security, safety, maintainability

9.2 Classification

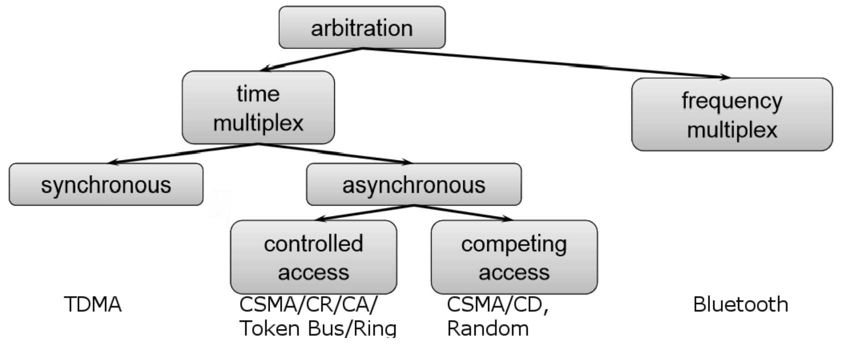


Figure 5: Overview over the communication classifications

9.3 Random Access

9.3.1 Fully randomized

no access control; requires low medium utilization, high access conflicts

9.3.2 Slotted random access

Start and end of access to the channel at given intervals

Theorem: Given sending rate p and n stations: probability that a slot is not taken by others is $(1-p)^{n-1}$

Theorem: Given sending rate p and n stations: probability that a station transmits successfully is $p(1-p)^{n-1}$

Theorem: Optimal sending rate p in case of n stations is given by $p = 1/n$

9.4 Time Division Multiple Access - TDMA

- Time multiplex, synchronous
- Communication in statically allocated time slots
- Synchronization necessary: master node sends out a synchronization frame or periodic repetition of communication frame
- Examples: satellite networks

9.5 Carrier Sense Multiple Access / Collision Detection – CSMA/CD

- Computers connected using a single cable. Only one computer can transmit at a time
- before starting to transmit, check whether the channel is idle
- if a collision is detected (several nodes started almost simultaneously), wait for some time (backoff timer)
- Collision detected by invalid CRC or unexpected voltages and currents
- repeated collisions result in increasing backoff times
- Examples: Ethernet, IEEE 802.3

Maximum Distance: The distance L [m] two nodes are separated from.

Propagation Speed: Speed σ [m/s] the signal travels

Slot-time: The minimum time $t_w = \frac{2L}{\sigma}$ [s] a node has to wait to reattempt transmission. Also this is the worst case time needed to detect a collision.

Minimum Packet size: TODO (wohl data-rate*slot-time)

data-rate: The rate the data is transmitted from one station to the other: B [bytes/sec].

Theorem: Given, that as exponential back-off algorithm in collision m a random value from $K \in \{0, ..., 2^m - 1\}$ is choosen. The probability of s successive collisions is given by: $p_s = \frac{1}{2} * \frac{1}{2^2} * ... * \frac{1}{2^{s-1}} = \frac{1}{2^{\sum_{i=1}^{s-1} i}}$, where $\sum_{i=1}^{s-1} i = \frac{(s-1)*s}{2}$

9.6 Token Protocols

- Token value determines which node is transmitting and/or should transmit next
- Only the token holder may transmit

- Null messages with tokens must be passed to prevent network from going idle
- IEEE 802.4, TokenRing

9.7 Token Ring

- Token owner can send data. Waits for acknowledgement from the receiver
- Sends null token to the next owner

9.8 Carrier Sense Multiple Access / Collision Avoidance - CSMA/CA - Flexible TDMA (FTDMA)

- reserve s slots for n nodes
- nodes keep track of global communication state by sensing
- nodes start transmitting a message only during the assigned slot
- if $s = n$, no collisions. if $s < n$, statistical collision avoidance
- examples: 802.11

9.9 Carrier Sense Multiple Access / Collision Resolution – CSMA/CR

- Before any message transmission, there is a global arbitration
- Each node is assigned a unique identification number
- All nodes wishing to transmit compete by transmitting a binary signal based on their identification value
- A node drops out the competition if it detects a dominant state while transmitting a passive state
- The node with the lowest identification value wins

9.10 FlexRay

9.10.1 General

- By FlexRay consortium (BMW, Ford, Bosch, DaimlerChrysler, General Motors, Motorola, Philips)
- High data rates can be achieved: 10Mbit/sec, but even much higher data rates possible

9.10.2 Principle

- Cycle is subdivided into a static and a dynamic segment

- Static segment is based on a fixed allocation of time slots to nodes
- Dynamic segment for transmission of ad-hoc communication with variable bandwidth requirements
- Two independent channels to eliminate single-point failures
- Star and Bus topologies possible

9.10.3 TDMA

- all static slots are the same length whether used or not
- all slots are repeated in order every communication cycle
- slots are lock-stepped in order on both channels

9.10.4 Flexible TDMA

- each minislot is an opportunity to send a message
- if message isnt sent, minislot elapses unused (short idle period)
- all nodes watch whether a message is sent so they can count minislots

9.11 Bluetooth

9.11.1 Design Goals

- small size
- low cost
- low energy
- secure transmission, robust transmission

9.11.2 Technical Data

- 2.4 GHz Band
- 10-100 m transmission range, 1 Mbit/s bandwidth for each connection
- simultaneous transmission of multimedia streams (synchronous) and data (asynchronous)
- ad hoc network, no centralized coordination, multi-hop communication

9.11.3 Frequency Hopping

- Transmitter jumps from one frequency to another with a fixed rate (1600 hops/s). The ordering (channel sequence) is determined by a pseudo random sequence of length $2^{27} - 1$
- Frequency range (2402 + k) MHz, k = 0 ...78

- The data transmission is partitioned into time windows of length 0.625 ms; each packet is transmitted by means of a different frequency
- For security reasons.

9.11.4 Network Topologies

Hierarchical structure (scatternet) of small nets (piconet)
Piconet:

- A piconet contains 1 master and maximally 7 slaves
- All nodes in a piconet use the same frequency hopping scheme (channel sequence) which is determined by the device address of the master BD_ADDR and phase which is determined by the system clock of the master.
- Connections are either one-to-one or between the master and all slaves (broadcast).
- A slave can never become active by themselves. Master has to poll!

Scatternet

- Several piconets with overlapping nodes form a scatternet. Up to 10 piconets per scatternet
- A node can simultaneously have the roles of slaves in several piconets and the role of a master in at most one piconet.
- The channel sequences of the different piconets are not synchronized.
- As a result, large network structures can emerge and multi-hop communication is possible

9.11.5 Packet Format

- The (channel) access code identifies all packets between Bluetooth devices.
- Packet Header identifies and characterizes connection between master and slave
- 126bit MAC-header (Channel Access Code and Packet Header)
- 24bit used for Payload-Header and CRC
- A typical packet has therfore a payload of 216 bits (DH1)

9.11.6 Addressing

- Bluetooth Device Address: BD_ADDR: 48 Bit, unique address for each device
- Active Member Address AM_ADDR: 3 Bit for maximally 7 active Slaves in a piconet, address Null is a broadcast to all slaves.
- Parked Member Address PM_ADDR: 8 Bit for parked slaves.

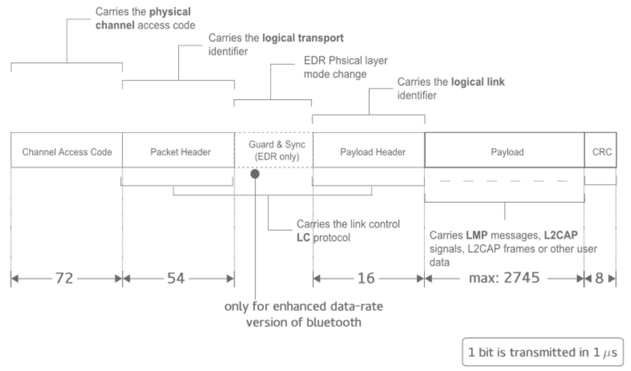

Carries the **physical channel** access code
Carries the **logical transport** identifier
EDR Phsical layer mode change
Carries the **logical link** identifier
Channel Access Code
Packet Header
Guard & Sync (EDR only)
Payload Header
Payload
CRC
Carries the link control **LC** protocol
Carries **LMP** messages, **L2CAP** signals, L2CAP frames or other user data
72
54
16
max: 2745
8
only for enhanced data-rate version of bluetooth
1 bit is transmitted in 1 μs

Figure 6: How a bluetooth packet is built up. Numbers in bit

9.11.7 Connection Types

- Synchronous Connection-Oriented (SCO)
  - Point to point full duplex connection between master and slaves
  - Reservation-based: Master reserves slots to allow transmission of packets in regular intervals.
- Asynchronous Connection-Less (ACL)
  - Asynchronous service
  - No reservation of slots
  - The master transmits spontaneously, the addressed slave answers in the following interval
  - Polling required

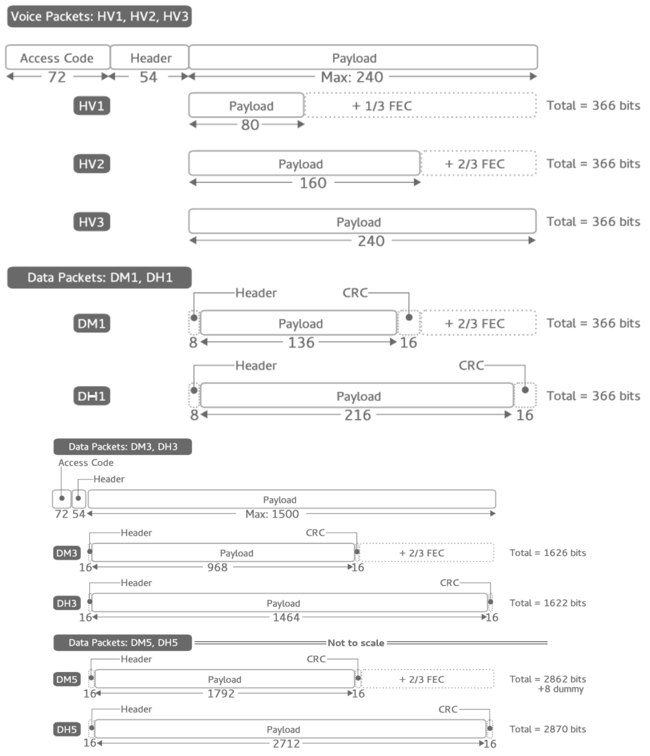
9.11.8 Frequency Hopping Time Multiplex

- A packet of the master is followed by a slave packet
- After each packet, the channel (frequency) is switched

9.11.9 Multi-Slot Communication

- Master can only start sending in even slot numbers.
- Packets from master or slave have length of 1, 3 or 5 slots
- In total, 2, 4 or 6 slots are used (for answering an additional slot is used)

- When the slave sends data across multiple slots, the frequency of the tramission is kept the same. The missed frequencies are skipped



9.11.10 Modes of Operation

- Inquiry (master identifies addresses of neighboring nodes)
- Page (master attempts connection to a slave whose address BD_ADDR is known)
- Connected (connection between master and slave is established)

9.11.11 States in Connection Mode

- active (active in a connection to a master)
- hold (does not process data packets)

- sniff (awakens in regular time intervals)
- park (passive, in no connection with master but still synchronized)

9.11.12 Synchronization in Connection Mode

- channel sequence of a piconet is determined by the BD_ADDR of the master
- The phase within the sequence is also determined by the master; all slaves follow

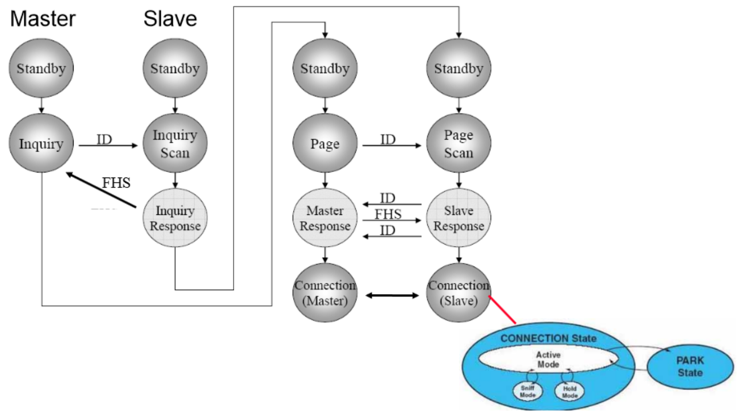


Figure 7: Overview how Bluetooth establishes a connection between Master and Slave

9.11.13 Page Mode

1. Page: Master transmits its own and slave address to slave (it uses a special channel sequence)
2. page scan: Slave listens, whether its own address is sent from a master
3. Slave answers the master with its own address
4. Master sends FHS-packet (frequency hop synchronization) to slave. It contains the channel sequence and the phase of the piconet

9.11.14 Protocol Hierarchy

- baseband specification: defines the packet formats, the physical and logical channels, the error correction, the synchronization between receiver and

transmitter, and the different modes of operation and states that allow the transmission of data and audio
- audio specification: defines the transmission of audio signals, in particular the coding and decoding methods
- link manager: covers the authentication of a connection and the encryption, the management of a piconet (synchronous/asynchronous connection), the initiation of a connection (asynchronous/synchronous packet types, exchange of name and ID) and the transition between different modes of operation and states
- host controller interface (HCI): defines a common standardized interface between a host and a bluetooth node; it is specified for several physical interconnections
- link layer control and adaptation layer (L2CAP): provides an abstract interface for data communication. It segments packets (up to 64kByte) and assembles them again, it allows the multiplexing of connections (simultaneous use of several protocols and connections) and allows the exchange of quality of service information between two nodes (packet rate, packet size, latency, delay variations, maximal rate).
- RFCOMM is a simple transport protocol that simulates a serial connection

10 Low Power Design

10.1 General

Performance-Power Efficiency list (Operations/Watt). From bad (low) to good (high performance while using low energy).

1. General-purpose processors
2. Application-specific instruction set processors (ASIPs)
3. Programmable hardware: FPGA
4. Application-specific integrated circuits (ASICs)

10.2 Power and Energy

Theorem (Energy): $E = \int P(t)dt$. To save energy, we can decrease the execution time or use less power.

10.2.1 Low power consumption

Important for:

- the design of the power supply
- the design of voltage regulators
- the dimensioning of interconnect
- cooling

10.2.2 Low energy consumption

Important due to

- restricted availability of energy
- limited battery capacities
- high costs of energy
- long term: low temperatures

10.2.3 Power Consumption of CMOS Gate

- leakage current: caused by electronic devices attached to the capacitors, such as transistors or diodes, which conduct a small amount of current even when they are turned off
- short circuit current: For a short time, both are conducting.
- switching current: Current used for switching the gate

Note: Without switching, there is no short circuiting current and no switching current!

10.2.4 Types of Power Consumption/ Consumption of CMOS Processors

- Dynamic power consumption: charging and discharging capacitors
- Short circuit power consumption due to switching
- Leakage: diodes, translators. The smaller the technology, the more important the leakage current

10.2.5 Dynamic Voltage Scaling - DVS

Supply Voltage: V_{dd}

Switching activity: α

Load Capacity: C_L

Clock frequency: f

threshold voltage: V_T where $V_T \ll V_{dd}$

Theorem (Power consumption of CMOS circuits (ignoring leakage)): $P \sim \alpha C_L V_{dd}^2 f$

Theorem (Delay for CMOS circuits): $\tau \sim C_L \frac{V_{dd}}{(V_{dd}-V_T)^2} \approx C_L \frac{1}{V_{dd}}$ with $V_T \ll V_{dd}$

- Decreasing V_{dd} reduces P quadratically (f constant)
- The gate delay increases reciprocally with decreasing V_{dd}
- Maximal frequency f_{max} decreases linearly with decreasing V_{dd}
- Assuming α, C_L constant, then $P \sim V_{dd}^3$

Theorem (Energy): $E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 * (\#cycles)$

- Reduce the supply voltage V_{dd}
- Reduce switching activity α
- Reduce the load capacitance C_L
- Reduce the number of cycles $\#cycles$

10.2.6 Power Supply Gating

Power gating is one of the most effective ways of minimizing static power consumption (leakage). Cut-off power supply to inactive units/components

10.3 Parallelism

A given task with V_{dd} and f_{max} which uses E_1 energy is split to two units with $V_{dd}/2$ and $f_{max}/2$. Both work in parallel. This results in a quarter of the energy used before: $E_2 = \frac{1}{4}E_1$. Calculated by inserting in the formulas.

10.4 Pipelining

A given task with V_{dd} and f_{max} which uses E_1 energy is split to two units with $V_{dd}/2$ and $f_{max}/2$. The units are in series, so the first does a pre-processing and the second unit finishes the task. This results in a quarter of the energy used before: $E_2 = \frac{1}{4}E_1$. Calculated by inserting in the formulas.

10.5 VLIW

Use the degree of parallelism to develop a chip with many computational units, (deeply) pipelined to make use of Parallelism and Pipelining to save energy. Explicit parallelism by a parallel instruction set or parallelization is done offline (compiler). Problem is that chip takes now different code: needs a chip on the chip which does the translation or re-compile the code or use a software compiler at run time to translate the instructions.

10.6 Dynamic Voltage Scaling

We know that: a) Not all components require same performance. and b) Required performance may change over time. Make now use of these two facts

Theorem (DVS): If the power-consumption is a convex function of the supply voltage, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling.

10.7 YDS Algorithm for Offline Scheduling

Solves the problem of scheduling tasks such that all these tasks can be finished no later than their deadlines and the energy consumption is minimized.

Intensity: Intensity $G([z, z'])$ in some time interval $[z, z']$: average accumulated execution time of all tasks that have arrival and deadline in $[z, z']$ relative to the length of the interval $z' - z$.
$V'([z, z']) = \{v_i \in V : z \leq a_i \leq d_i \leq z'\}$
$G([z, z']) = \sum_{v_i \in V'([z,z'])} c_i/(z' - z)$

1. Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.
2. Adjust the arrival times and deadlines by excluding the possibility to execute at the previous critical intervals.
3. Run the algorithm for the revised input again.
4. Put pieces together. Run the tasks with EDF with the computed frequencies. Output is a diagram which shows at what time which threads is scheduled with what frequency.

Theorem: The algorithm guarantees the minimal energy consumption while satisfying the timing constraints

Theorem: Time complexity is $O(N^3)$ where N is the number of tasks in V. Number of iterations is at most N.

Theorem: For periodic real-time tasks with deadline=period, running at constant speed with 100% utilization under EDF has minimum energy consumption while satisfying the timing constraints.

10.8 DVS - Online Scheduling on One Processor

Continuously update to the best schedule for all arrived tasks. Compute the intensity for each incoming task and update

Theorem: Compared to the optimal off-line solution, the on-line schedule uses at most 27 times of the minimal energy consumption.

10.9 Dynamic Power Management (DPM)

- Shut down devices if they are not to be used
- Wake them up when service is required
- Can reduce leakage and dynamic pow
- States:
  - Run: operational
  - Idle: a SW routine may stop the CPU when not in use, while monitoring interrupts
  - Sleep: Shutdown of on-chip activity

Problem: overhead due to shutdown and wakeup delay

DVS Critical frequency (voltage): DVS Critical frequency (voltage): running at any frequency/voltage lower than this frequency is not worthwhile for execution.

10.10 Procrastination Schedule

Run the YDS algorithm but round all frequencies up to the critical frequency. If the frequency determined by the YDS algorithm is smaller than the critical frequency, there will be sleep time for the processor, since tasks are faster executed in that intervall than needed. To reduce the number of turn on/offs and maximize the sleep time, the tasks are bundled together and executed as late as possible.

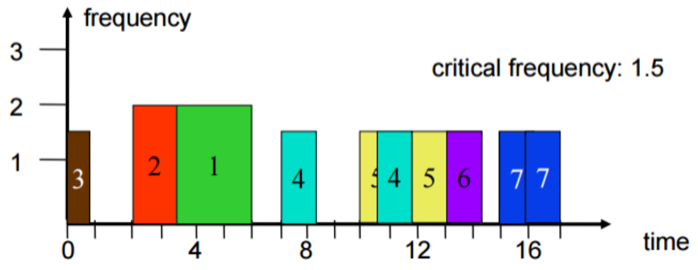


Figure 8: YDS with rounding up and procrastrination

11 Architecture Design

11.1 Task Graph/Dependence Graph/Data Flow Graph

Theorem (Task Graph or Dependence Graph (DG)): A dependence graph is a directed graph $G = (V, E)$ in which $E \subseteq V \times V$ is a partial order.

If $(v1, v2) \in E$, then v1 is called an immediate predecessor of v2 and v2 is called an immediate successor of v1.

Suppose E^* is the transitive closure of E. If $(v1, v2) \in E^*$, then v1 is called a predecessor of v2 and v2 is called a successor of v1.

- A dependence graph describes order relations for the execution of single operations or tasks. Nodes correspond to tasks or operations, edges correspond to relations (executed after).
- Usually, a dependence graph describes a partial order between operations and therefore, leaves freedom for scheduling (parallel or sequential). It represents parallelism in a program but no branches in control flow
- A dependence graph is acyclic
- A dependence graph cannot handle branches

11.2 Control-Data Flow Graph (CDFG)

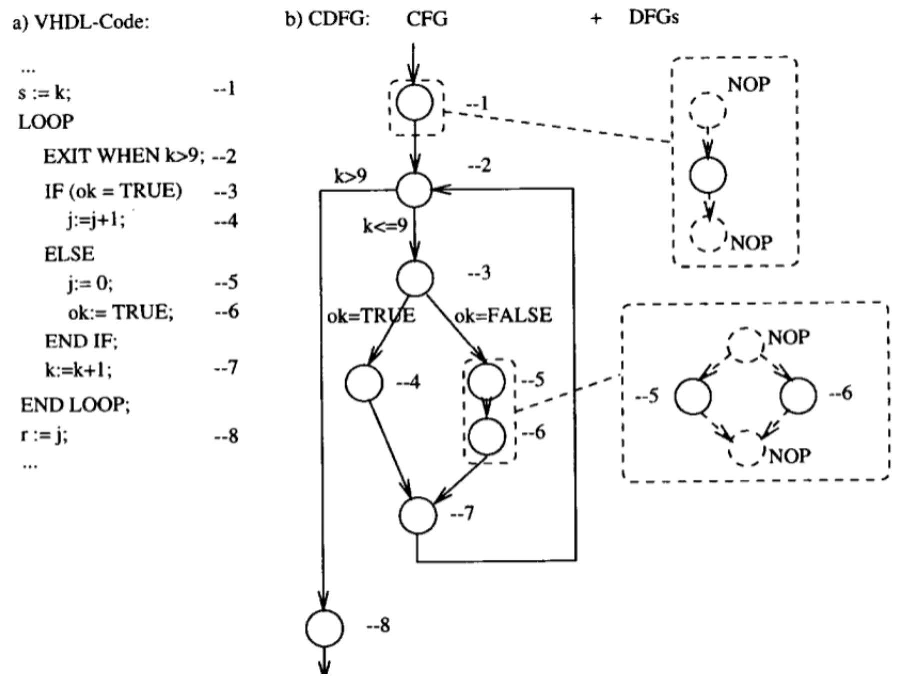
- Goal: Description of control structures (for example branches) and data dependencies
- Applications: Describing the semantics of programming languages, internal representation in compilers for hardware and software
- Representation: Combination of control flow (sequential state machine) using the CFG and dependence representation using DFG

11.2.1 Control Flow Graph (statement transitions)

- corresponds to a finite state machine, which represents the sequential control flow in a program
- Branch conditions are very often associated to the outgoing edges of a node.
- The operations to be executed within a state (node) are associated in form of a dependence graph

11.2.2 Data Flow Graph (statement dependence)

NOP (no operation) operations represent the start point and end point of the execution. This form of a graph is called a polar graph: it contains two distinguished nodes, one without incoming edges, the other one without outgoing edges



```
...
s := k;                 --1
LOOP
  EXIT WHEN k>9;        --2
  IF (ok = TRUE)        --3
    j:=j+1;             --4
  ELSE
    j:= 0;              --5
    ok:= TRUE;          --6
  END IF;
  k:=k+1;               --7
END LOOP;
r := j;                 --8
...
```

Figure 9: An example of a CDFG graph

11.3 Sequence Graph (SG)

A sequence graph is a hierarchy of directed graphs. A generic element of the graph is a dependence graph with the following properties

- It contains two kinds of nodes: (a) operations or tasks and (b) hierarchy nodes
- Each graph is acyclic and polar with two distinguished nodes: the start node and the end node. No operation is assigned to them (NOP)
- There are the following hierarchy nodes: (a) module call (CALL) (b) branch (BR) and (c) iteration (LOOP)

Note: A hierarchical sequence graph can be cyclic.

11.3.1 Example

```
x := a * b;
y := x * c;
z := a + b;
IF z > 0 THEN
        p := m + n;
        q := m * n;
END IF
```

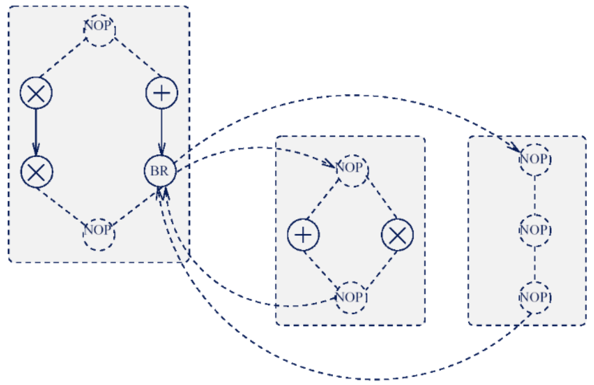


Figure 10: An example of a sequence graph. The edges to the if-else part are labelled with the condition or else

11.4 Resource Graph

Resource Graph: Resource Graph $G_R = (V_R, E_R), V_R = V_S \bigcup V_T$ where V_S denotes the operations of the algorithm and V_T the resource types of the architecture and G_R is a biparpite graph. An edge $(v_s, v_t) \in E_R$ represents the availability of a resource type v_t for an operation v_s.

Cost function: $c : V_T \to Z$

Execution times: $w : E_R \to Z \geq 0$ are assigned to each edge $(v_s, v_t) \in E_R$ and denote the execution time of operation $v_s \in V_S$ on resource type $v_t \in V_T$

11.5 Marked Graphs (MG)

Marked Graph: A marked graph $G = (V, A, del)$ consists of nodes (actors) $v \in V$, edges $a = (v_i, v_j) \in A, A \subseteq V \times V$, number of initial tokens on edges $del : A \to N$.

The marking (distribution of tokens) is often represented in a form of a vector. For an example, see figure 11.

- The token on the edges correspond to data that are stored in FIFO queues
- A node (actor) is called activated if on every input edge there is at least one token
- A node (actor) can fire if it is activated
- The firing of a node v_i (actor operates on the first tokens in the input queues) removes from each input edge a token and adds a token to each output edge. The output token correspond to the processed data
- Marked graphs are mainly used for modeling regular computations, for example signal flow graphs.

11.5.1 Implementation as a synchronous digital circuit

- Actors are implemented as combinatorial circuits
- Edges correspond to synchronously clocked shift registers

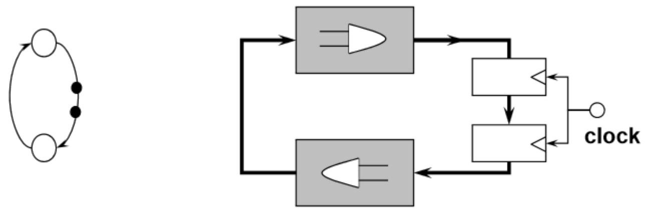


Figure 11: Implemenation of a marked graph as a synchronous digital circuit

11.5.2 Implementation as a self-timed asynchronous circuit

- Actors and FIFO registers are implemented as independent units
- The coordination and synchronization of firings is implemented using a handshake protocol
- Delay insensitive direct implementation of the semantics of marked graphs

11.5.3 Implementation in software (static scheduling)

Software implementation with static scheduling

- At first, a feasible sequence of actor firings is determined which ends in the starting state (initial distribution of tokens).
- This sequence is implemented directly in software

11.5.4 Implementation in software (dynamic scheduling)

- Scheduling is done using a (real-time) operating system.
- Actors correspond to threads (or tasks)
- After firing (finishing the execution of the corresponding thread) the thread is removed from the set of ready threads and put into wait state
- It is put into the ready state if all necessary input data are present
- This mode of execution directly corresponds to the semantics of marked graphs. It can be compared with the self-timed hardware implementation.

12 Architecture Synthesis

12.1 General

Goal: Determine a hardware architecture that efficiently executes a given algorithm.

- Allocation: the necessary hardware resources
- Scheduling: the timing of individual operations
- Binding: the relation between individual operations of the algorithm and hardware resources

12.2 Model

- V_S: the set of all operations.
- Sequence Graph $G_s = (V_S, E_S)$.
- V_T: the set of all resource types (eg. ALU, float-unit)
- $V_R = V_S \cup V_T$
- Resource graph $G_R = (V_R, E_R)$. Notice, the G_R is bipartite and denotes, which operation can be executed by which resource.
- $c : V_T \mapsto \mathbb{Z}$ (cost)
- $w : E_R \mapsto \mathbb{Z}_{\geq(0)}$(execution time)

12.3 Classification

- heuristics or exact methods
- unlimited resources: no constraints in terms of the available resources are defined.
- limited resources: constrains are given in terms of the number a types of available resources.
- iterative algorithms: an initial solution to the architecture synthesis is improved step by step.
- constructive algorithms: the synthesis problem is solved in one step.
- transformative algorithms: the initial problem formulation is converted into a (classical) optimization problem

12.4 Allocation, Binding, Scheduling, Latency, Mobility

Allocation: An allocation is a function $\alpha : V_T \to \mathbb{Z} \geq 0$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

Binding: A binding is defined as a function $\beta : V_S \to V_T$ and $\gamma : V_S \to Z \geq 0$. Here $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the rth instance of resource type $v_t \in V_T$. So the binding decides the type and instance for each operation.

Schedule: A schedule is a function $\tau : V_S \to \mathbb{Z} \geq 0$ that determines the starting times of operations. A schedule is feasable if the conditions $\tau(v_j) - \tau(v_i) \geq w(v_i) \forall (v_i, v_j) \in E_S$ are satisfied. $w(v_i) = w(v_i, \beta(v_i))$ denotes the execution time of operation v_i.

Latency: The latency L of a schedule is the time difference between start node v_0 and end node $v_n : L = \tau(v_n) - \tau(v_0)$. Notice, this holds, because the sequence graph ends in a NOP, which has execution time 0.

Mobility: The mobility is the difference between the starting time of the operation in the ALAP to the starting time of the operation in the ASAP algorithm: $\mu = t^L - t^S$.

The mobility of operations on the critical path is zero (time of ALAP and ASAP is the same).

12.5 Multiobjective Optimization

Multiobjective Optimization: Optimization problems with several objectives are called multiobjective optimization problems.

Architecture Synthesis is an optimization problem with more than one objective

- Latency of the algorithm that is implemented
- Hardware cost (memory, communication, computing units, control)
- Power and energy consumption

12.6 Pareto

Pareto-Doinance: A solution $a \in X$ weakly Pareto-dominates a solution $b \in X$, denoted as $a \preceq b$, if it is as least as good in all objectives, i.e. $f_i(a) \leq f_i(b)$ for all $1 \leq i \leq n$. Solution a is better then b, denoted as $a \prec b$, iff $(a \preceq b) \wedge (b \npreceq a)$.

Pareto-optimal: A solution is named Pareto-optimal, if it is not Pareto-dominated by any other solution in X. Pareto-optimal solutions are incomparable.

Pareto-optimal front: The set of all Pareto-optimal solutions is denoted as the Pareto-optimal set and its image in objective space as the Pareto-optimal front

Note: Pareto-optimal solutions are incomparable!

12.7 Scheduling without Resource Constraints

Can be used...

- as a preparatory step for the general synthesis problem
- to determine bounds on feasible schedules in the general case
- if there is a dedicated resource for each operation

12.7.1 Problem Definition

Given is a sequence graph $G_S = (V_S, E_S)$ and a resource graph $G_R = (V_R, E_R)$. Then the latency minimization without resource constraints is defined as $L = min\{\tau(v_n) - \tau(v_0) : \tau(v_j) - \tau(v_i) \geq w(v_i) \forall (v_i, v_j) \in E_S\}$. Note: often $\tau(v_0) = const$ and therefore $\tau(v_0)$ can be removed from the formula. L denotes the latency.

12.7.2 As Soon as Possible (ASAP) - Algorithm

Run-time complexity: $O(|V_S| + |E_S|)$

```
ASAP($G_S(V_S, E_S)$, w) {

        $\tau(v_0) = 1$ //start-time
        REPEAT {
                Determine $v_i$ whose predecessor are planed;
                //Can only start when all predecessor
                //are finished
                $\tau(v_i)$ = max{$\tau(v_j)$ + $w(v_j) \forall (v_j, v_i) \in E_S$ }
        } UNTIL($v_n$ is planned);
        RETURN($\tau$);
        }
}
```

12.7.3 As Late As Possible (ALAP) - Algorithm

Run-time complexity: $O(|V_S| + |E_S|)$

```
ALAP($G_S(V_S, E_S)$, w, $L_{max}$) {
        $\tau(v_n) = L_{max} + 1$ //end-time
        REPEAT {
                Determine $v_i$ whose successor are planed;
                $\tau(v_i) = min\{\tau(v_j) \forall (v_i, v_j) \in E_S\} - w(v_i)$
        } UNTIL($v_0$ is planned);
        RETURN($\tau$);
        }
}
```

Note: L_{max} is chosen for example by the LIST algorithm (or ASAP).

12.8 Scheduling with Timing Constraints

12.8.1 Examples of Timing Constraints

- deadline: latest finishing times of operations. Example: $\tau(v_2) + w(v_2) \leq 5$
- release times: earliest starting times of operations. Example: $\tau(v_3) \geq 4$
- relative constraints: differences between starting times of a pair of operations. Example: $\tau(v_6) - \tau(v_7) \geq 4$
- Note: Deadlines and release times are defined relative to the start node v_0

12.8.2 Conversion Rules

- Minimum constraint: $\tau(v_j) \geq \tau(v_i) + l_{ij} \rightarrow \tau(v_j) - \tau(v_i) \geq l_{ij}$
- Maximum constraint: $\tau(v_j) \leq \tau(v_i) + l_{ij} \rightarrow \tau(v_i) - \tau(v_j) \geq -l_{ij}$
- Equality constraint: $\tau(v_j) = \tau(v_i) + l_{ij} \rightarrow \tau(v_j) - \tau(v_i) \leq l_{ij} \wedge \tau(v_j) - \tau(v_i) \geq l_{ij}$

12.8.3 Weighted Constraint Graph

Timing constraints can be represented in form of a weighted constraint graph

Weighted Constraint Graph: A weighted constraint graph $G_C = (V_C, E_C, d)$ related to a sequence graph $G_S = (V_S, E_S)$ constains nodes $V_C = V_S$ and a weighted edge for each timing constraint. An edge $(v_i, v_j) \in E_C$ with weight $d(v_i, v_j)$ denotes the constraint $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$

A consistent assignment of starting times $\tau(v_i)$ to all operations can be done by solving a single source longest path problem. For example by using the Bellmand-Ford algorithm (Complexity: $O(|V_C| * |E_C|)$)

12.8.4 Bellman-Ford

Iteratively set $\tau(v_j) = max\{\tau(v_j), \tau(v_i) + d(v_i, v_j) : (v_i, v_j) \in E_C\}$ for all $v_j \in V_C$ starting from $\tau(v_i) = -\inf$ for $v_i \in V_C \backslash \{v_0\}$ and $\tau(v_0) = 1$.

Note: Bellman-Ford does not work with positive cycles, but this is no problem since a graph with positive cycles will not have a solution!

12.9 Scheduling with resource constraints

Given is a sequence graph $G_S = (V_S, E_S)$, a resource graph $G_R = (V_R, E_R)$ and an associated allocation α and binding β. Then the minimal latency is defined as:
$L = min\{\tau(v_n) : (\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i))\forall(v_i, v_j) \in E_S) \wedge$
$(|\{v_s : \beta(v_s) = v_t \wedge \tau(v_s) \leq t < \tau(v_s) + w(v_s, v_t)\}| \leq \alpha(v_t)$
$\forall v_t \in V_T, \forall 1 \leq t \leq L_{max})\}$ where L_{max} denotes an upper bound on the latency.

12.9.1 List Scheduling

- A heuristic algorithm: does not yield the minimal latency!
- To each operation there is a priority assigned which denoetes the urgendy of being scheduled. This priority is static, determined before List Scheduling
- The algorithm schedules one time step after the other
- U_k denotes the set of operations that are mapped onto resource v_k and whose predecessors finished
- T_k denotes the currently running operations mapped to resource v_k

```
LIST($G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta$, priorities)
{
        t = 1
        REPEAT {
                //go through all ressource types
                FORALL $v_k \in V_T$ {

                determine candidates to be scheduled $U_k$;
                determine running operations $T_k$;
                choose $S_k \subseteq U_k$ with maximal priority
                (choose set of operations)
                and $|S_k| + |T_k| \leq \alpha(v_k)$;
                //assign operations
                        $\tau(v_i) = t \forall v_i \in S_k$;

                }
                //increase time
                t = t +1;
        } UNTIL ($v_n$ planned)

        RETURN $\tau$;
}
```

12.9.2 Integer Linear Programming

Principle:

1. Synthesis Problem: transform into ILP
2. Integer Linear Program: optimization of ILP
3. Solution of ILP: back interpretation
4. Solution of Synthesis Problem

- Yields optimal solution to synthesis problem as it is based on an exact mathematical description of the problem
- Solves scheduling, binding and allocation simultaneously!
- Assumptions:
  - The binding is determined already, dh. every operation v_i has a unique execution time $w(v_i)$
  - We have determined the earliest and latest starting times of operations v_i as l_i and h_i respectively. For this we can use ASAP and ALAP

Theorem (Equation and Goal): minimize
$\tau(v_n) - \tau(v_0)$ with subject to

1. $x_{i,t} \in \{0, 1\} \forall v_i \in V_S \; \forall t : l_i \leq t \leq h_i$
   (declare variables to be binary)

2. $\sum_{t=l_i}^{h_i} x_{i,t} = 1 \qquad \forall v_i \in V_S$
   (make sure that exactly one variable $x_{i,j}$ for all t has the value 1, the others 0)

3. $\sum_{t=l_i}^{h_i} t * x_{i,t} = \tau(v_i) \qquad \forall v_i \in V_S$
   determine relation between x and starting times of operations τ

4. $\tau(v_j) - \tau(v_i) \geq w(v_i) \qquad \forall (v_i, v_j) \in E_S$
   (Guarantees that all precedence constraints are satisfied)

5. $\sum_{\forall i:(v_i,v_k)\in E_R} \sum_{p'=max\{0,t-h_i\}}^{min\{w(v_i)-1,t-l_i\}} x_{i,t-p'} \leq \alpha(v_k)$
   $\forall v_k \in V_T \forall t : 1 \leq t \leq max\{h_i : v_i \in V_S\}$
   (Make sure that the resource constraints are not violated. The number of active operations does not increase the number of available resource instances)

Note: Approach:

- Perform ASAP and ALAP scheduling to find l_i, h_i
- Define binary variables x_i
- Define the starting times of the operations $\tau(v_i)$
- Write down the set of constraints (precedence, resource)
- Specify the objective function

12.9.3 Iterative Algorithms

Iterative algorithms: Iterative algorithms consist of a set of indexed equations that are evaluated for all values of an index variable l: x_i denotes a set of indexed variables, F_i denotes arbitrary functions and d_{ji} are constant index displacements.

Example: Examples are signal flow graphs and marked graphs

Representation - One indexed equation:
$y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3] \qquad \forall l$
is equivalent to: $x_1[l] = au[l] \qquad \forall l; x_2[l] = x_1[l] + dy[l-3] \qquad \forall l; x_3[l] = x_2[l] + cy[l-2] \qquad \forall l; y[l] = x_3[l] + by[l-1] \qquad \forall l$
Representation - Extended sequence graph $G_S = (V_S, E_S, d)$: to each edge $(v_i, v_j) \in E_S$ there is assosicated the index displacement d_{ij}. An edge $(vi, v_j) \in E_S$ denotes that the variable corresponding to v_j depends on variable corresponding to v_i with displacement d_{ij}.
Representation - signal flow graph
Representation - Loop program

```
while(true)
{
        t1 = read(u);
        t5 = a*t1 + d*t2+c*t3+b*t4;
        t2  = t3;
        t3 = t4;
        t4 = t5;
        write(y, t5);
}
```

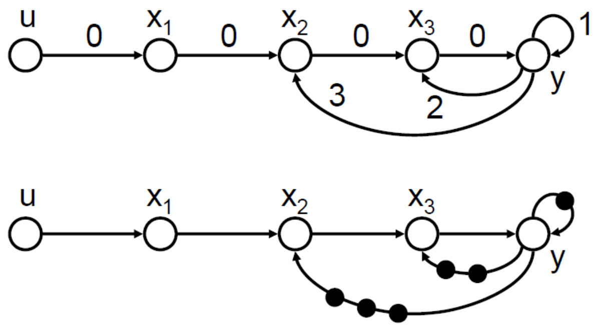


Figure 12: The iterative algorithm visualized as a marked graph.

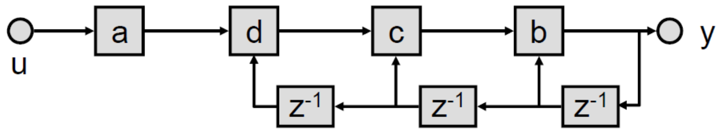


Figure 13: The iterative algorithm visualized as a signal flow graph.

Iteration: An iteration is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index l.

Iteration interval: The iteration interval P is the time distance between two successive iterations of an integer algorithm.

Throughput: $1/P$ denotes the throughput of the implementation.

Latency: The latency L is the maximal time distance between the starting and the finishing times of operations belonging to one iteration.

Functional pipelining: A pipelined implementation. Here exist time instances where the operations of different iterations l are executed simultaneously.

Loop folding: In case of loop folding, starting and finishing times of an operation are in different physical iterations.

Implementation principles

- Simple possibility: the edges with $d_{ij} > 0$are removed from the extended sequence graph. The resulting simple sequence graph is implemented using standard methods.
- Using functional pipelining: Successive iterations overlap and a higher throughput $1/P$ is obtained.

Solving the synthesis problem using integer linear programming

- Start with ILP formulation given for simple sequence graphs
- Use the extended sequence graph with displacements d_{ij}
- ASAP and ALAP scheduling for upper and lower bounds h_i and l_i use only edges with $d_{ij} = 0$
- Suppose that a suitable iteration interval P is choosen beforehand. If it is too small, no feasable solution to the ILP exists and P needs to be increased

Replacement of Equations We can use the same equations, only replace

- (4) replaced by $\tau(v_j) - \tau(v_i) \geq w(v_i) - d_{ij} * P \qquad \forall (v_i, v_j) \in E_S$
- (5) is replaced by $\sum_{\forall i:(v_i,v_k)\in E_R} \sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t-p'+p*P \leq h_i} x_{i,t-p'+p*P} \leq \alpha(v_k) \forall t : 1 \leq t \leq P, \forall v_k \in V_t$

12.9.4 Dynamic Voltage Scaling

We can transform the DVS problem into an integer linear program optimization: we can optimize the energy in case of dynamic voltage scaling.

Example of a set of tasks with dependency constraints:

- We suppose that a task $v_i \in V_S$ can use one of the execution times $w_k(v_i) \forall k \in K$ and corresponding energy $e_k(v_i)$. There are $|K|$ different voltage levels.
- We suppose that there are deadlines $d(v_i)$ for each operation v_i.
- We suppose there are no resource constraints, can execute everything in parallel

1. minimize: $\sum_{k\in K} \sum_{v_i \in V_S} y_{ik} * e_k(v_i)$
   (sum up all individual energies of operations)
   with subject to:
2. $y_{ik} \in \{0, 1\} \qquad \forall v_i \in V_S, k \in K$
   (make decision variables y_{ik} binary)

3. $\sum_{k\in K} y_{ik} = 1 \qquad \forall v_i \in V_S$
   (guarantee that exactly one implementation $k \in K$ is choosen for each operation v_i)
4. $\tau(v_j) - \tau(v_i) \geq \sum_{k\in K} y_{ik} * w_k(v_i) \qquad \forall (v_i, v_j) \in E_S$
   (implement the precedence constraints)
5. $\tau(v_i) + \sum_{k\in K} y_{ik} w_k(v_i) \leq d(v_i) \qquad \forall v_i \in V_S$
   (guarantee deadlines)

13 Labs

13.1 General

What is the BTnode?

- autonomous wireless communication and computing platform
- has bluetooth radio
- developed @eth
- Atmel ATmega128L microcontroller, 8MHz, 64+180 KB RAM

Features

- Non preemptive multi-threading
- Events
- Periodic and one-shot timers
- Dynamic heap memory allocation
- Interrupt driven streaming I/O

13.2 Terminal/minicom

We can connect to the BTnode by entering “minicom usb0“ into the linux terminal. The following commands are then available:

- bt - bluetooth radio
- led - toggle LED patterns
- bat - battery status
- nut - show OS information
- log - BTnut logging features

13.3 Running Time

In cycles, note: 8MHz cpu.

brne (taken)	2
brne (not taken)	1
ldi	1
rcall	3
ret	4
subi	1

13.4 Looping

```
//because only 8bit registers are available
//runs ca. 1.02 seconds
void waitabit() {
        int i, j;
        for(i = 0; i <= 1200; i++) {
                for(j = 0; j <= 1200; j++) {
                        asm volatile  ("nop" ::);
                }
        }
}
```

13.5 LED on

```
#include <avr/io.h>
void init(void) {
        MCUCR |= 1<<SRE; // enable external memory bus
        DDRB |= 1<<DDB5; // set latch pin as output
}
int main(void)
{
        volatile char* pointer;
        char dummy;
        init();
        // compute the pseudo-address that
        // contains the values for the LEDs
        //write 0x0 to turn off
        pointer  = (char*) (((short)0x2) << 8); //blue
        // force the compiler to write this
        // pseudo-address to the address-bus
        dummy   = * pointer;
        // now enable the latch
        PORTB |= 1<<PB5;
        // wait a moment - volatile keeps the
        // compiler from removing this line
        asm volatile  ("nop" ::);
        // disable the latch, i.e. hold the value
        PORTB &= ~(1<<PB5);
        return 0;
}
```

13.6 Battery

13.6.1 Theorem

We need a conversion from the input to get the voltage: $ADC = \lceil 1023 * V_in/V_ref \rceil$ with $V_ref = 3.3V$ Note, that this BAT-SENSE signal delivers the half voltage. For example, for 1V, then $\lceil 1023 * 0.5/3.3 \rceil = 155$.

13.6.2 Code

To read out the battery value, we have to code the following:

```
ADCSRA |= 1<<ADPS0; //for the
ADCSRA |= 1<<ADPS1; // slowest
ADCSRA |= 1<<ADPS2; //speed = hightes accuracy
ADCSRA |= 1<<ADEN; // ADC enable to enable the ADC
ADCSRA |= 1<<ADSC; // ADC start conversion bit
```

Do now a polling on the ADSC to check if conversion has finished. If so, read out the 10 bits (ADCL the lowest 8 bit, then ADCH the highest two bits).

Note: See Lab 2, Task 1.3 for the solution Code!

13.7 Interrupts

The interrupts are based on the ATmega128's Timer/Counter3. It has a Timer/-Counter interrupt mask register (ETIMSK). Because CPU as a frequency of 7.3 MHz, to generate a 2second timer, we compute: $2^{16} = 65536 clocks = 9ms$. So to get 2s, we compute $2s/9ms = 222.2$. So we take 256 as a prescalar and check, if the value reaches then $7.3Mhz/256 * 2s = 57031 = 0xdec7$.

```
CR3AH = 0xde; //first part of the threshold (because only 8 bit)
CRAL = 0xc7; //second part of the threshold
ETIMSK |= 1 << OCIE3A; // interrupt is based on the threshold
```

The routine to register a interrupt has the following declaration: `NutRegisterIrgHandler(irq, handler, arg)`. Where

- IRQ: Interrupt number to be associated with this handler. Zb `sig_OVERFLOW3` to register an overflow of counter3, `sig_OUTPUT_COMPARE3A` to trigger an interrupt if the counter reaches the threshold saved.
- Handler: This routine is called by the OS, when a specified interrupt occurs
- Arg: the argument passed

Note: See Lab 2, Task 2.2, 2.3 for the code

13.8 OS

13.8.1 General

- Called Nut/OS
- Three layers:
  - the rudimentary C-libraries as implemented by avr-gccs $avr - libc$
  - the higher-level OS routines built on top of $avr - libc$ by Nut/OS
  - The BTnode-specific device drivers
- Compiled by avr-gcc
- Compiled in the console by `make btnode3 upload`
- Main should never exit, otherwise the behaviour is undefined

13.8.2 LED

Light all four LEDS after each other.

```
#include <hardware/btn-hardware.h>      /*for hardware init*/
#include <led/btn-led.h>                /*for LEDs*/
#include <sys/timer.h>                  /*for NutSleep*/
```

```
int main(void)
{
        u_char led = 0;
        btn_hardware_init();                    /*init hardware*/
        btn_led_init(0);                        /*init LEDs*/
        for(;;)
        {
                led = 0;
                while (led < 4)
                {
                        /*turn LED on*/
                        btn_led_set(led);
                        /*delay to make LED switch visible*/
                        NutSleep(500);
                         /*turn LED off*/
                        btn_led_clear(led);
                        led++;
                }
        }
}
```

13.8.3 Timer

```
#include <hardware/btn-hardware.h>
#include <sys/timer.h>
// use multiple timer if you have multiple methods to call
HANDLE hTimer;
//arbitrary name for callback method
static void _tm_callback(HANDLE h, void* a)
{
        // do something when timer expires
}
int main (void)
{
        btn_hardware_init();
        //install and start timer
        //ARGS: a) time in ms (every 3 seconds)
        // b) Argument to pass
        //c) TM_ONESHOT (fire only once ) or 0 (periodic)

        hTimer = NutTimerStart(3000, _tm_callback, NULL, 0);
        for (;;) { NutSleep(1000); }
}
```

13.8.4 Terminal

On the host, run `minicom usb0` or `minicom usb1`. Check, if configured to `57.6k, 8N1, no flow control`.

Note: printf does not support %f !

```
#include <stdio.h>
#include <dev/usartavr.h>
//link the printf output
void init_stdout(void)
{
        u_long baud = 57600;
        NutRegisterDevice(&APP_UART, 0, 0);
        //r+ for read+write
        freopen(APP_UART.dev_name, "r+", stdout);
        _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
}

int main(void)
{
        btn_hardware_init();
        init_stdout();

        int variable = 13;
        printf("Hello world, ");
        printf("my lucky number is %d\n",variable);
        for (;;);
}
```

13.8.5 Threads

- Uses cooperative multithreading
- Thread functions should never return. They are exited by themselves by explicitly calling `NutThreadExit();`
- A thread can be in one of the following states: `RUNNING` (only one thread), `READY`, `SLEEPING`.
- `NutThreadYield();` will check for threads that are more important than myself. If so, then another thread is run, otherwise it will keep running. It passes the current thread into the READY state.
- Idle-Thread is a thread with lowest priority. Runs if no other thread can be run

- `NutSleep(INTEGER_TIME_IN_MS);` will set the thread into the `SLEEPING` state and run another thread that is waiting (or the idle-thread)
- There are priorities. Can be set with `NutThreadSetPriority(INTEGER);`. 0 is hightes priority, 254 lowest priority! Default is 64
- Waiting for an event puts the thread into the `SLEEPING` state.
- Multiple threads with the same priority are executed in FIFO order
- A thread can get his name by using `runningThread->td_name`

Because of the higher priority, the thread my_thread will not give up the cpu.

```
#include <hardware/btn-hardware.h>
#include <sys/thread.h>
#include <led/btn-led.h>

// without any NutThreadYield the red LED is on
THREAD(my_thread, arg)
{
        //higher priority: red is on
        NutThreadSetPriority(20);

        // lower priority: blue is on
        //NutThreadSetPriority(80);
        for(;;)
        {
                btn_led_set(LED0);
                btn_led_clear(LED1);
                NutThreadYield();
        }
}
int main(void)
{
        btn_hardware_init();
        btn_led_init(0);
        NutThreadCreate("My thread", my_thread, 0, 192);
        for(;;)
        {
                btn_led_clear(LED0);
                btn_led_set(LED1);
                NutThreadYield();
        }
}
```

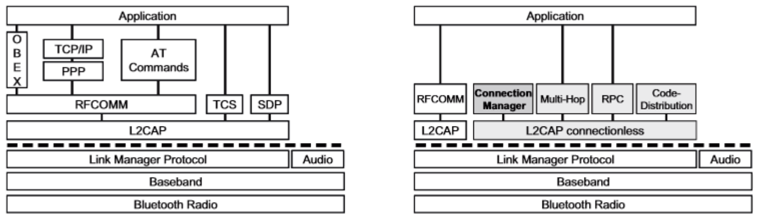
13.8.6 Events

- Datatype `HANDLE var_name`
- Post an event by `NutEventPost(&event)`
- Wait for an event by `NutEventWait(&event, timeout)`. For infinite timeout, use `NUT_WAIT_INFINITE`

Note: See Lab 3, Task 6 for code.

14 Bluetooth

14.1 Overview



(a) Standard Bluetooth protocol stack. (b) BTnut Bluetooth protocol stack.

Figure 14: Comparison between the the standard and the BTnut Bluetooth protocol stack

- Link Manager Protocol (LMP): controls the radio link between two devices.
- Host Controller Interface (HCI): defines a common standardized interface between the Bluetooth host (e.g. the Atmega128 microcontroller) and the Bluetooth controller (e.g. the baseband controller and the Bluetooth radio)
- Logical Link Control and Adaptation Protocol (L2CAP): provides an abstract interface for data communication
- Connection Manager: forms and maintains a connected topology, manages discovery and connection of devices.
- Multi-Hop: performs multi-hop routing and forwarding, provides an API for sending and receiving packets.

14.2 Bluetooth Header

- Byte [0, 4]: HCI-header

- Byte [5, 8]: L2CAP header
- Byte [5, 6]: message length. To set the length, use `packet[HCI_HEADER_LEN] = (char) (msg_len & 0xFF)` and `packet[HCI_HEADER_LEN + 1] = (char) ((msg_len >> 8) & 0xFF)`. To read the length, use `int msg_len = packet[5] | (packet[6] << 8);`
- Byte [7, 8]: channel. To set the channel, use `packet[HCI_HEADER_LEN + 2] = (char) (CHANNEL & 0xFF);` and `packet[HCI_HEADER_LEN + 3] = (char) ((CHANNEL >> 8) & 0xFF);`
- Byte [9, ...]: The message

15 Threads

15.1 Creating a thread

```
THREAD(my_thread, arg)
{
        for(;;)
        { //do something }
}

int main(void)
{
        //192 = stack size
        if( 0 == NutThreadCreate("My thread", my_thread, 0, 192))
        { //creation failed }

        for(;;)
        { //do something }
}
```

15.2 Terminating a thread

```
THREAD(my_thread, arg)
{
        for(;;)
        { //do something

                //can only kill itself
                if( CONDITION )
                        NutThreadExit();
        }
}
```

15.3 Sleep

```
THREAD(my_thread, arg)
{
        for(;;)
        {
                //do something
                NutSleep(1000);
        }
}
```

15.4 Co-Operative Scheduling

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

HANDLE my_event;

THREAD(thread_A, arg)
{
        for(;;)
        {
                //do something
                NutEventWait(&my_event, NUT_WAIT_INFINITE);
                //do something
        }
}

THREAD(thread_B, arg)
{
        for(;;)
        {
                //do something
                NutEventPost(&my_event);
                //so something
        }
}
```

16 Appendix

16.1 Derivative

$f(x)$	$F(x)$
x^n	$\frac{1}{n+1}x^{n+1}$
$(ax+b)^n$	$\frac{1}{a\cdot(n+1)}(ax+b)^{n+1}$
$\sqrt{x}$	$\frac{2}{3}x^{\frac{3}{2}}$
$\sqrt[n]{x}$	$\frac{n}{n+1}x^{\frac{1}{n}+1}$
$\frac{1}{ax+b}$	$\frac{1}{a}\ln\lvert ax+b\rvert$
$\frac{ax+b}{cx+d}$	$\frac{ax}{c}-\frac{ad-bc}{c^2}\ln\lvert cx+d\rvert$
$\frac{1}{x^2-a^2}$	$\frac{1}{2a}\ln\left\lvert\frac{x-a}{x+a}\right\rvert$
$\sqrt{a^2+x^2}$	$\frac{x}{2}f(x)+\frac{a^2}{2}\ln(x+f(x))$
$\sqrt{a^2-x^2}$	$\frac{x}{2}\sqrt{a^2-x^2}-\frac{a^2}{2}\arcsin\frac{x}{\lvert a\rvert}$
$\sqrt{x^2-a^2}$	$\frac{x}{2}f(x)-\frac{a^2}{2}\ln(x+f(x))$
$\frac{1}{\sqrt{x^2+a^2}}$	$\ln(x+\sqrt{x^2+a^2})$
$\frac{1}{\sqrt{x^2-a^2}}$	$\ln(x+\sqrt{x^2-a^2})$
$\frac{1}{\sqrt{a^2-x^2}}$	$\arcsin(\frac{x}{\lvert a\rvert})$
$\frac{1}{\sqrt{1-x^2}}$	$\arcsin(x)$
$\frac{-1}{\sqrt{1-x^2}}$	$\arccos(x)$
$\frac{1}{x^2+a^2}$	$\frac{1}{a}\arctan(\frac{x}{a})$
$\frac{-1}{1+x^2}$	$\operatorname{arccot}(x)$
$\frac{1}{\sqrt{x^2+1}}$	$\operatorname{arsinh}(x)$
$\frac{-1}{\sqrt{x^2-1}}$	$\operatorname{arcosh}(x)$
$\frac{1}{1-x^2}$	$\operatorname{artanh}(x)$
$\sin(ax+b)$	$-\frac{1}{a}\cos(ax+b)$
$\cos(ax+b)$	$\frac{1}{a}\sin(ax+b)$
$\tan(x)$	$-\ln\lvert\cos(x)\rvert$
$\cot(x)$	$\ln\lvert\sin(x)\rvert$

$f(x)$	$F(x)$
$\frac{1}{\sin(x)}$	$\ln\left\lvert\tan(\frac{x}{2})\right\rvert$
$\frac{1}{\cos(x)}$	$\ln\left\lvert\tan(\frac{x}{2}+\frac{\pi}{4})\right\rvert$
$\sin^2(x)$	$\frac{1}{2}(x-\sin(x)\cos(x))$
$\cos^2(x)$	$\frac{1}{2}(x+\sin(x)\cos(x))$
$\tan^2(x)$	$\tan(x)-x$
$\cot^2(x)$	$-\cot(x)-x$
$\arcsin(x)$	$x\arcsin(x)+\sqrt{1-x^2}$
$\arccos(x)$	$x\arccos(x)-\sqrt{1-x^2}$
$\arctan(x)$	$x\arctan(x)-\frac{1}{2}\ln(1+x^2)$
$\sin^n(x)$	$s_n=-\frac{1}{n}\sin^{n-1}x\cos x+\frac{n-1}{n}s_{n-2}$
	$s_0=x \quad s_1=-\cos(x)$
$\cos^n(x)$	$c_n=\frac{1}{n}\sin x\cos^{n-1}x+\frac{n-1}{n}c_n$
	$c_0=x \quad c_1=\sin(x)$
$\sinh(x)$	$\cosh(x)$ und umgekehrt
$\tanh(x)$	$\ln(\cosh(x))$
$\frac{f'(x)}{f(x)}$	$\ln\lvert f(x)\rvert$
$\ln\lvert x\rvert$	$x\cdot(\ln\lvert x\rvert-1)$
$\frac{1}{x}(\ln x)^n$	$\frac{1}{n+1}(\ln x)^{n+1} \quad n\neq-1$
$\frac{1}{x}\ln x^n$	$\frac{1}{2n}(\ln x^n)^2 \quad n\neq0$
$\frac{1}{x\ln x}$	$\ln\lvert\ln x\rvert \quad x>0,x\neq1$
a^{bx}	$\frac{1}{b\ln a}a^{bx}$
$x\cdot e^{cx}$	$\frac{cx-1}{c^2}\cdot e^{cx}$
$x^n\ln x$	$\frac{x^{n+1}}{n+1}\left(\ln x-\frac{1}{n+1}\right) \quad n\neq-1$
$e^{ax}p(x)$	$e^{ac}(a^{-1}p(x)-a^{-2}p'(x)+$
	$\ldots+(-1)^n a^{-n-1}p^{(n)}(x))$
	p: Polynom n-ten Grades
$e^{cx}\sin(ax+b)$	$\frac{e^{cx}(c\sin(ax+b)-a\cos(ax+b))}{a^2+c^2}$
$e^{cx}\cos(ax+b)$	$\frac{e^{cx}(c\cos(ax+b)+a\sin(ax+b))}{a^2+c^2}$

$f(x)$	$f'(x)$
x^n	nx^{n-1}
$\sqrt{x}$	$\frac{1}{2\sqrt{x}}$
$\sqrt[n]{x}$	$\frac{1}{n}x^{\frac{1}{n}-1}$
$\frac{1}{f(x)}$	$\frac{-f'(x)}{(f(x))^2}$
$\tan(x)$	$\tan^2(x)+1=\frac{1}{\cos^2(x)}$
$\cot(x)$	$-(1+\cot^2(x))=\frac{-1}{\sin^2(x)}$

$f(x)$	$f'(x)$
$\ln\|x\|$	$\frac{1}{x}$
$\log_a \|x\|$	$\frac{1}{x\ln a}=\log_a(e)\frac{1}{x}$
a^{cx}	$a^{cx}\cdot c\ln a$
x^x	$x^x\cdot(1+\ln x)\quad {\scriptstyle x>0}$
$(x^x)^x$	$(x^x)^x(x+2x\ln(x))\quad {\scriptstyle x>0}$
$x^{(x^x)}$	$x^{(x^x)}(x^{x-1}+\ln x\cdot x^x(1+\ln x)))$