Master informatique – première année

Option STR

Real-time systems

Isabelle PUAUT
puaut@irisa.fr
https://team.inria.fr/alf/members/isabelle-puaut/

Embedded real-time systems

Definitions

- Embedded system
  - Not inside a computer
  - Invisible to the end-user
- Typical characteristics
  - Joint use of hardware and software to implement a domain-specific function
    - Software used for more flexibility
    - Hardware (microprocessor/microcontroller, ASIC, DSP) used for more performance
  - Interactive systems

Definition

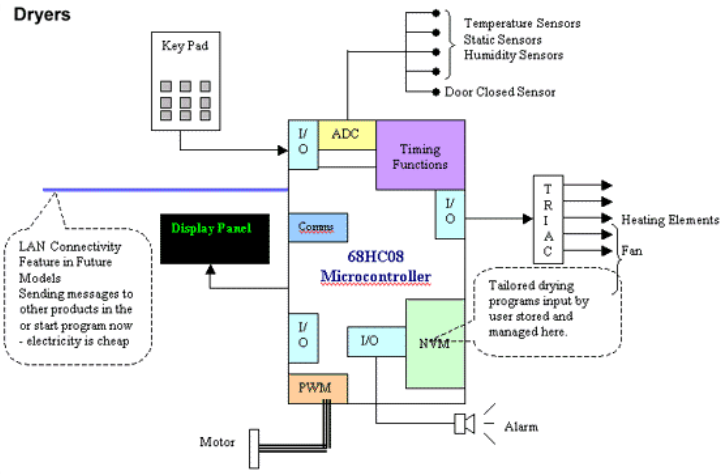
- Typical characteristics (cont.)
  - Domain-specific functions - not « general purpose »
  - Increasing hardware and software complexity
  - Increasing need for performance
  - Specific components might be part of the system
    - Domain-specific processors (DSP - Digital Signal Processors)
    - Hardware implemented components (decoders, encryption, etc.)

Application domains

- Household appliances
  - Coffee machine, washing machine, microwave oven
- Consumer electronics
  - Digital camera, video decoder, mobile phone, personal digital assistant, video game console
- Automotive
  - Anti-Blocking System (ABS), engine control
- Industrial control applications
- Avionics, spatial, energy production (nuclear)
- Computer peripherals : FAX, printer

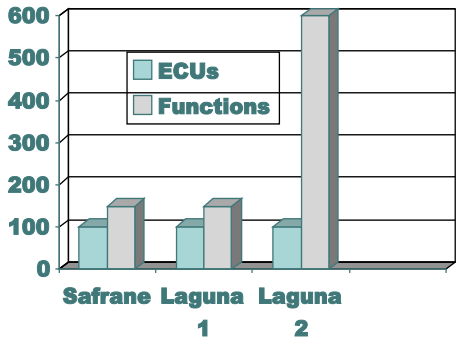
➔ Large variety of needs (performance, reliability, criticality)

Example: dryer



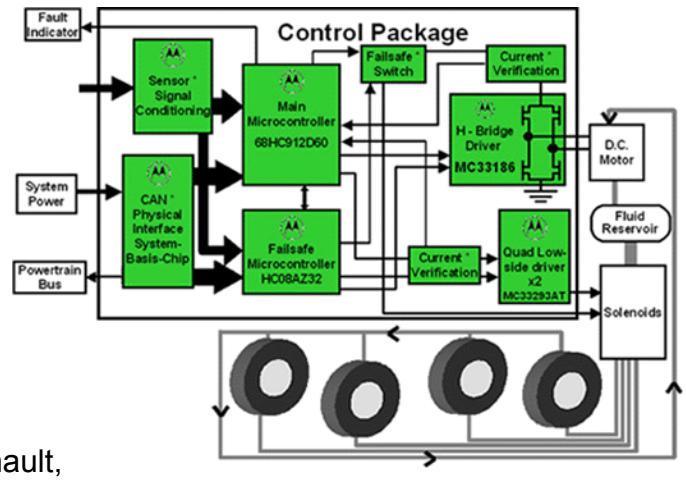
Example: automotive applications

- Number of software functions in cars



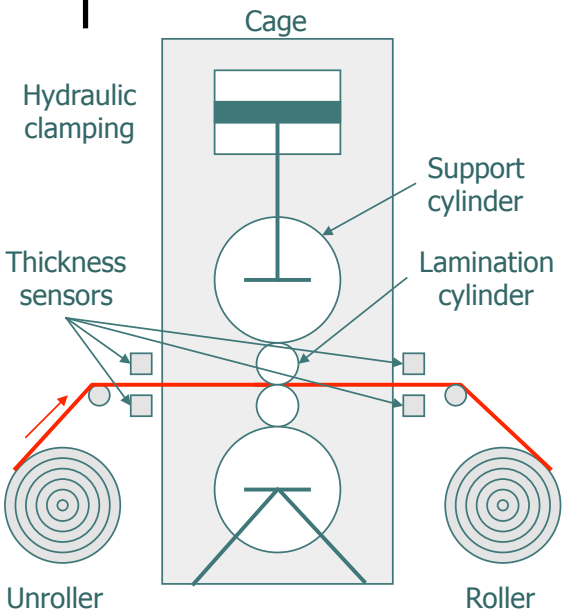
(Source: C. Balle, Renault, october 2001)

- Anti-Blocking system (ABS)



(source: www.motorola.com)

Example: aluminium lamination [Cottet, 2000]



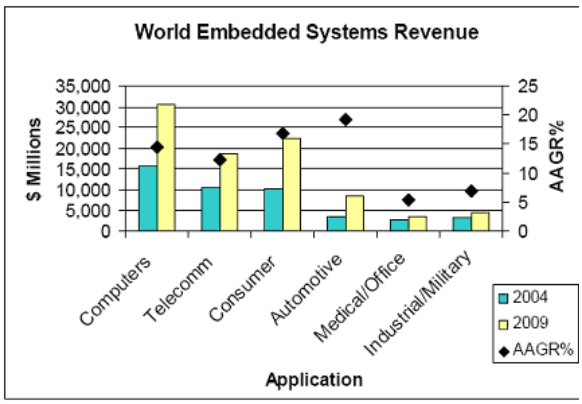
- Aluminium lamination
- Kerosene lubrication
- Output thickness : 0.25 to 0.45 mm
- Band speed: 108 km/h
- Computer controlled:
  - Output thickness
  - Band flatness
  - Motor speed,
  - …
- Specific constraints
  - Availability : 24h/24h, worst-case unavailability 16h/week during week-ends
  - Security: band break, kerosene blaze

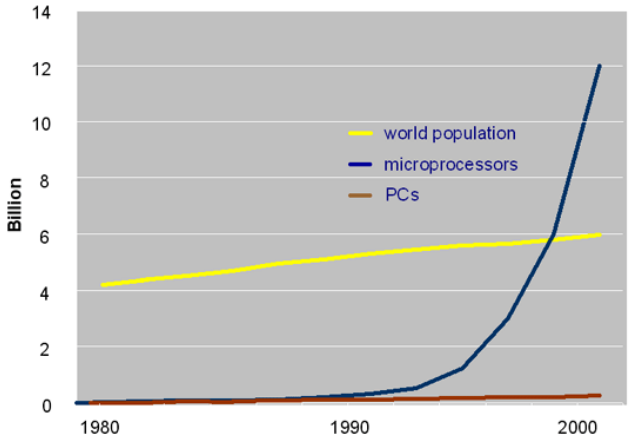
Example: aluminium lamination [Cottet, 2000]

- Real time monitoring of aluminium lamination
  - Subset of the control system
  - Role : monitor and store in a database the signals from the lamination control computers
- Signals (periodic)
  - Compensation of runouts (“faux ronds”): 984 bytes every 4 ms
  - Arrival of a new aluminium roll: 160 bytes every 3 mn
  - Regulation of clamping power: 544 bytes every 20 ms
  - Output flatness : 2052 bytes every 100 ms
- Timing constraints come from the external environment

Embedded real-time systems

The embedded systems market





(Source: the Hipeac roadmap on Embedded Systems, European Commission)

Embedded real-time systems

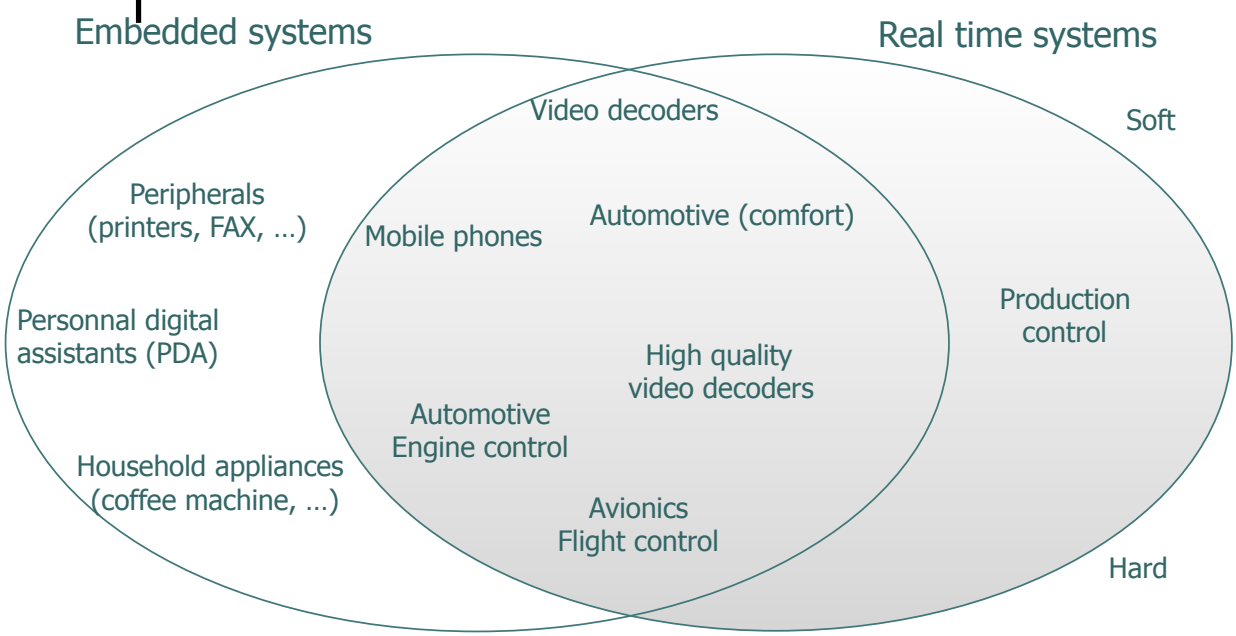
Design constraints (1/2)

- Real time
  - Real time (physical time) is a correctness criterion
  - Need to guarantee timing constraints (predictability)
- Safety
  - Definition: capability to be confident in the result of a service
  - Obstacles: failures, resulting from faults
  - Means: fault avoidance, fault tolerance, prevision or errors
  - Large spectrum of methods depending on the type of faults and the application criticality (hardware, software)
  - Safety critical systems : certification

Design constraints (2/2)

- Cost
  - Design cost (delay): time to market
  - Purchase cost: surface of silicium
    - Limited resources (memory)
    - Limited energy (cost of batteries)
  - Processors used in embedded systems simpler than general purpose processors (cost + predictability)
  - Ergonomy: weight constraints
- Trade-offs have to be found
  - Ex: energy vs performance trade-off
  - Multi-criteria exploration for hardware / software selection

Real time systems vs embedded systems



Course syllabus

- Introduction to real time systems
- Real time scheduling
  - Classification, scheduling policies
- Real time operating systems (RTOS)
  - Motivations, typical services
  - Scheduling
  - Synchronisation, communication, memory management
  - Non exhaustive list of RTOSs
- Schedulability analysis and worst-case execution time estimation
- Hybrid scheduling of hard and soft real time tasks
- Scheduling on multi-core systems

Outside the scope of the lecture

- Fault tolerance
- Parallel and networked systems
- Design, modelling
  - (UML-RT, HRT-HOOD)
- Programming languages, synchronous programming
- Formal methods in real-time systems (model checking), testing, simulation
- Signal processing, sampling, hardware-software co-design, dedicated hardware (DSP, ASICs)
  - C.f. ASE, U.E. M1

References

- A. Burns et A. Wellings. *Real-Time Systems and Programming Languages*, second edition, Addison Wesley
- G. C. Buttazzo. *Hard Real-Time Computing Systems, predictable scheduling and applications*, Kluwer academic publishers
- F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri. *Ordonnancement temps-réel, cours et exercices corrigés*, Hermes
- J. Liu. *Real-Time Systems*. Prentice Hall
- H. Kopetz. Real-Time Systems. Kluwer academic publishers

Introduction to real time systems

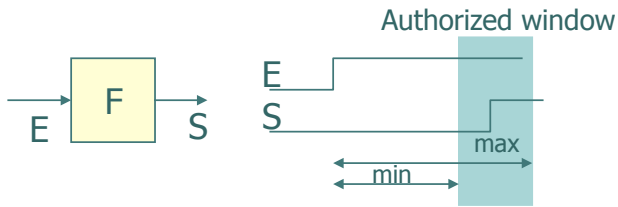
Real time: a definition

- Real time
  - Time: correctness depends not only on the logical result of the computation but also on the time at which the result is produced
  - Real: physical, external time

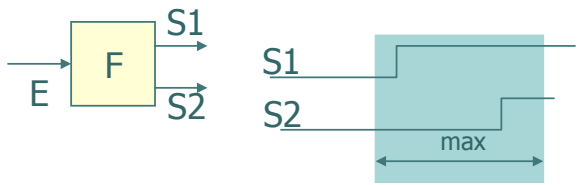
- Timing constraint: quantified limit between two events (min and/or max)
  - Quantified = expressed in terms of physical time
  - Ex: Deadline = maximum delay between task arrival and task termination

Examples of timing constraints

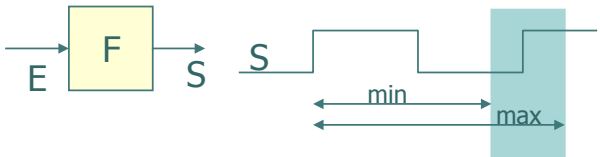
- Minimum/maximum delay after arrival



- Consistency between task termination points
  - Ex: sound/image synchronisation



- Regularity constraints
  - Ex: regularity of frame display in video applications

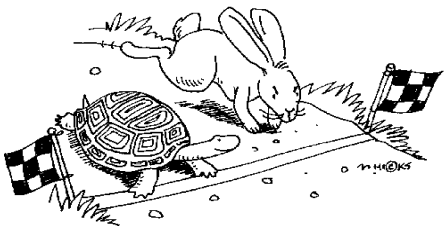


Classes of real time systems

- Hard real time: violating a timing constraint results in catastrophic consequences (loss of human lives, ecological of financial disaster): need of guarantees
  - Ex: control applications in transportation systems, nuclear power plants, spatial applications, etc.
  - Considered by default in this lecture
- Soft real time: meeting timing constraints is desirable, but missing a timing constraint does not jeopardize the system correctness
  - Ex : multimedia applications (video on demand, television)
- Interactive applications: reaction time invisible to the user
  - Not considered as real time here

Bad interpretations of real time

- « Real-time is not real-fast »



  - Hard timing constraints are not necessarily short (application dependent)
  - Computing power: may be necessary, but not always sufficient
  - Average case or peak performance is not the objective, worst-case performance is
    - Murphy's law : if something can go wrong, it will go wrong

Key points when designing real time systems

- Predictability
  - Knowledge of all computations is required (application, OS): worst-case load
  - Knowledge of worst-case execution time of any activity is required -> predictable hardware
- Scheduling (selection of execution order)
  - Known scheduling policies with associated theoretical results (optimality)
- Guaranteeing that deadlines are met
  - Testing is necessary, but not sufficient
  - Feasibility conditions (schedulability analysis)

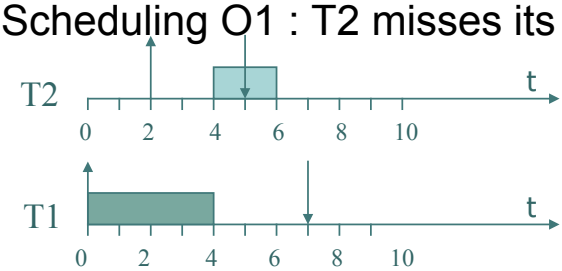
Real time scheduling

Introduction (1/2)

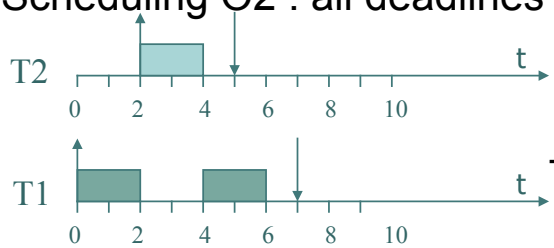
- What is scheduling?
  - Set of rules defining the execution order of tasks on the processor
- Why is scheduling important?
  - Task scheduling has an impact on timing constraints
  - Example :
    - Task T1 : arrival at time 0, WCET 4, absolute deadline 7
    - Task T2 : arrival at time 2, WCET 2, absolute deadline 5
    - Scheduling O1: FIFO (first-in first-out) non preemptive
    - Scheduling O2: preemptive, priority-based, higher priority to T2
    - Deadlines are met with O2, not with O1

Introduction (2/2)

- Scheduling O1 : T2 misses its deadline



- Scheduling O2 : all deadlines are met



T=2 : T1 is preempted by T2

Classification (1/5)
« Off-line » vs « on-line » scheduling

- Off-line (or « time-driven » scheduling)
  - Pre-computation of the execution sequence at design time (the schedule is stored in a table)
  - At run time, a dispatcher executes the generated schedule

- Evaluation
  - Pros
    - Implementation is simple
    - Deadline misses due to overloads are easy to detect
  - Cons : rigid
    - Need to know task arrival dates (includes periodic arrivals)

Classification (2/5)
« Off-line » vs « on-line » scheduling

- On-line scheduling
  - Scheduling decisions are taken at run time
  - Scheduling points: task arrival, task termination
  - Scheduling policy: defines which task is to be scheduled next
  - Mostly used scheduling policies: priority-based
- Evaluation
  - Pros: flexibility
  - Cons:
    - More costly implementation
    - Overloads harder to detect

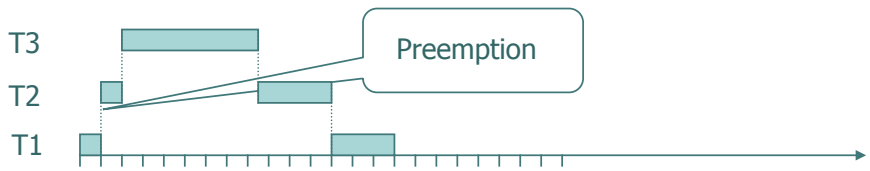
Classification (3/5)
Preemptive vs non preemptive scheduling

- Non preemptive
  - A task, once started, is executed until completion (except when the task self-suspends)
- Preemptive
  - The running task can be interrupted at any time to assign the processor to another active task

- Preemptive ⇒ need for a multitask real time operating system
- Remark : preemptive/non-preemptive criterion orthogonal to the off-line/on-line criterion

Classification (4/5)

- Priority-based scheduling
  - Preemptive. At any time, the running task is the active task with higher priority
- Fixed priorities / dynamic priorities
  - Fixed: selected at design time
  - Dynamic: the priority of a task may vary over time
- Example
  - prio(T3) > prio(T2) > prio(T1) (fixed priorities)
  - T1, T2 and T3 arrival dates: 1, 2, et 3



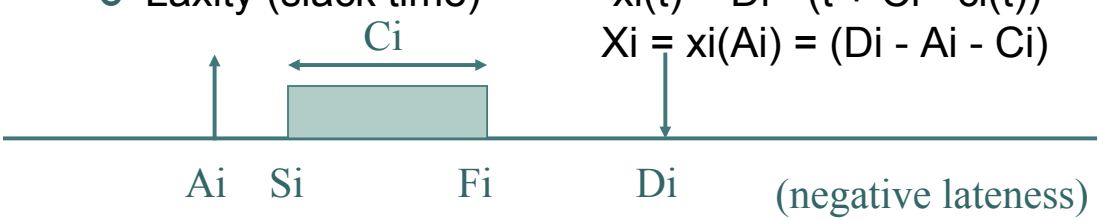
Classification (5/5)

- Vocabulary
  - Feasible schedule: schedule in which all task constraints are met
  - Schedulable task set: there exist at least one algorithm able to produce a feasible schedule
- Optimal vs heuristic
  - Optimal: may fail to meet a deadline only if no other algorithm of the same class can meet it
    - Superiority in given class of algorithms
    - Similarly: absolute optimality (all scheduling policies)
  - Heuristic: tends toward but does not guarantee to find a feasible schedule

Notations (1/2)

- Arrival — Ai
- Worst-case execution time in isolation — Ci
- Deadline (absolute or relative) — Di
- Start time — Si
- Finish time — Fi
- Lateness — Li = (Fi - Di)
- Laxity (slack time) — xi(t) = Di - (t + Ci - ci(t))

  Xi = xi(Ai) = (Di - Ai - Ci)



Notations (2/2)

- Task arrival
  - Periodic: infinite sequence of identical activations (jobs) regularly activated at a constant rate (Period, Pi)
    - Initial arrival date: offset (or phase) : Oi
    - Synchronous arrival: for all i,j Oi=Oj
    - Hyperperiod: scheduling cycle
      - synchronous task sets : interval [0,LCM(Pi)] for
      - else, interval [min(Oi), max(Oi,Oj+Dj) + 2 * LCM(Pi)]
  - Sporadic: infinite sequence of identical activations, minimum inter arrival time (Pseudo-period, Pi)
  - Apériodic: task arrivals are not regular
- Synchronisations (possible blockings)
  - Shared resources
  - Precedence

More terminology

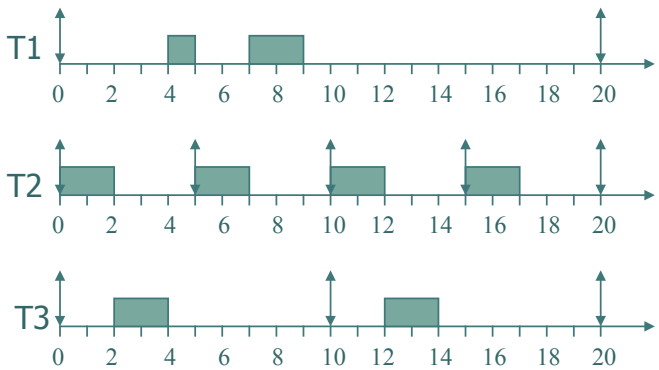
- Tasks vs jobs
  - Task : code to be executed
  - Job : a specific task execution (a periodic task results in an infinite number of jobs)
- Priority assignment
  - Fixed priorities
    - FTP: Fixed Task Priority
  - Dynamic priorities
    - FJP: Fixed Job Priority
    - DJP: Dynamic Job Priority

Preemptive fixed priority scheduling (FTP)
Rate Monotonic (1/2)

- For periodic task sets
- Definition [Liu & Layland, 1973]
  - Tasks with shorter periods will have higher priorities (in case of identical period, deterministic tie-breaking rule)
- Property
  - Optimal among fixed priority assignments for periodic independent tasks with Di=Pi

Preemptive fixed priority scheduling (FTP)
Rate Monotonic (2/2)

Task	Pi	Ci	Di
T1	20	3	20
T2	5	2	5
T3	10	2	10



Prio(T2) > Prio(T3) > Prio(T1)
Cyclic execution
(LCM of periods)

Preemptive fixed priority scheduling (FTP)
Feasibility of periodic task sets under RM

Task	Pi	Ci	Di
T1	20	3	20
T2	5	2	5
T3	10	2	10

- Per-task processor utilisation factor:
  - $U_i = C_i / P_i$
- Global processor utilisation factor (load):
  - $U = \Sigma C_i/P_i$
- Necessary feasibility condition: $U \leq 1$

Preemptive fixed priority scheduling (FTP)
Deadline Monotonic (1/2)

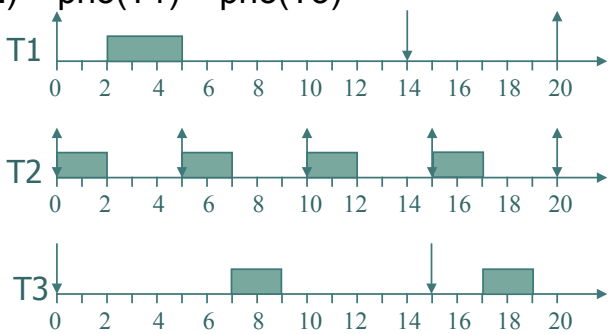
- For periodic task sets
- Definition [Leung & Whitehead, 1985]
  - Each task is assigned a priority inversely proportional to its relative deadline (deterministic tie-breaking rule in case of equality)
- Property
  - Optimal among fixed priority schedulers for periodic independent tasks with $Di \leq Pi$
- Remark: DM is a generalisation of RM

Preemptive fixed priority scheduling (FTP)
Deadline Monotonic (2/2)

- Example: DM scheduling for a synchronous periodic task set

Task	Pi	Ci	Di
T1	20	3	14
T2	5	2	5
T3	15	2	15

- prio(T2) > prio(T1) > prio(T3)



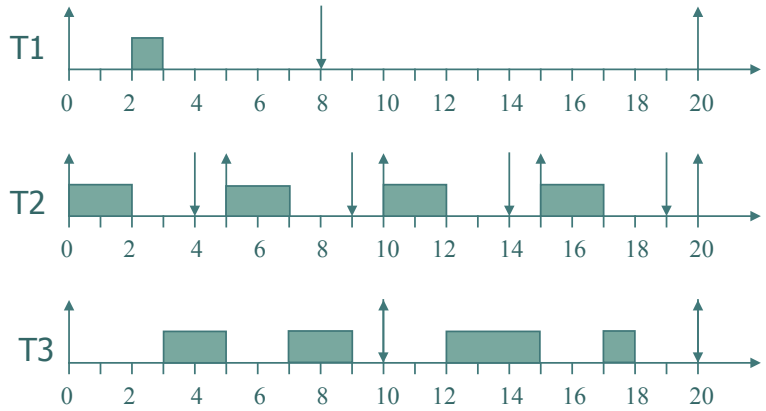
Preemptive dynamic priority scheduling (FJP)
Earliest Deadline First (1/2)

- Applicable for both periodic and non-periodic task sets
- Definition [Liu & Layland, 1973]
  - Tasks with earlier absolute deadlines will be executed at higher priorities (deterministic tie-breaking rule in case of equality)
  - Preemptive
- Property
  - Optimal among preemptive scheduling algorithms for periodic independent tasks with Di ≤ Pi

Preemptive dynamic priority scheduling (FJP)
Earliest Deadline First (2/2)

Task	Pi	Ci	Di
T1	20	1	8
T2	5	2	4
T3	10	4	10

(2 preemptions at times 5 and 15)



Preemptive dynamic priority scheduling (DJP)
Least Laxity First (LLF) (1/2)

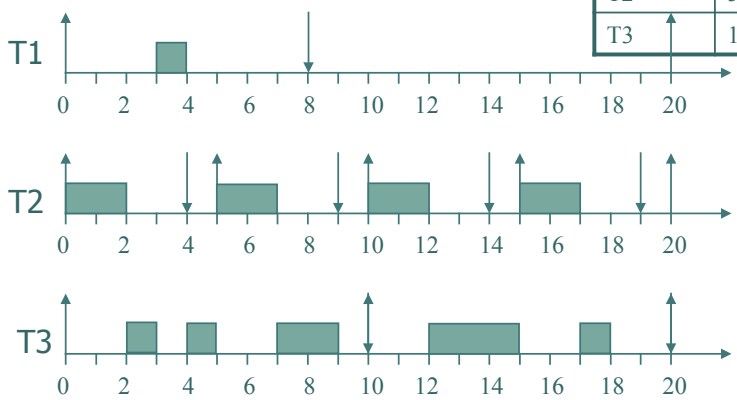
- Definition [Mok & Dertouzos, 1989]
  - Tasks with lower laxities xi(t) = Di - (t + Ci - ci(t)) will be executed at higher priorities (deterministic tie-breaking rule in case of equality)

- Optimal among preemptive scheduling algorithms for periodic independent tasks with Di ≤ Pi

Preemptive dynamic priority scheduling (DJP)
Least Laxity First (LLF) (2/2)

- Example (same task set as for EDF)

Task	Pi	Ci	Di
T1	20	1	8
T2	5	2	4
T3	10	4	10



(3 preemptions at times 3, 5 and 15)

(Here, in case of equality, selection of task with smaller tid, at t=3)

How to select a scheduling policy?

- According to feasibility
  - Some task sets meet their deadlines only if preemptive scheduling is used
- Feasibility conditions (schedulability analyses)
  - Exist for all introduced schedulers
  - Higher reachable load bound for EDF than for RM/DM

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(2^{\frac{1}{n}}-1) \qquad \sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$$

  - Bad behaviour of EDF in presence of overloads (domino effect)
- Implementation considerations
  - Off-line scheduling: simple (no need for a RTOS)
  - On-line scheduling: fixed priority schedulers easier to implement than dynamic ones

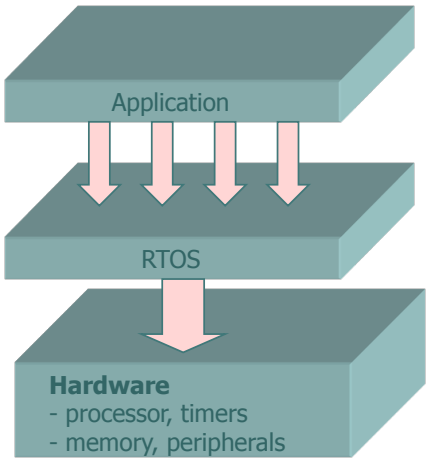
Real time operating systems (RTOS)

Why using a RTOS?

- Objective of a real time operating systems
  - Share the processor between concurrent computations (multi-tasking)
  - Abstract from the target hardware
- Why using a RTOS?
  - Intrinsic parallelism of applications
  - Some applications meet their deadlines only when using preemptive scheduling (see introductory example)
  - Portability considerations
  - Better exploitation of processor during I/O operations (re-scheduling during I/O operations)

Main functions of a RTOS

- Objective
  - Mask hardware peculiarities to applications through the provision of a standard interface



System calls

Ex : task management (creation, stop)
Ex : time management (program a timeout request)

(Hidden) hardware manipulations

Ex : save processor registers at preemption points
Ex : write into timer internal registers

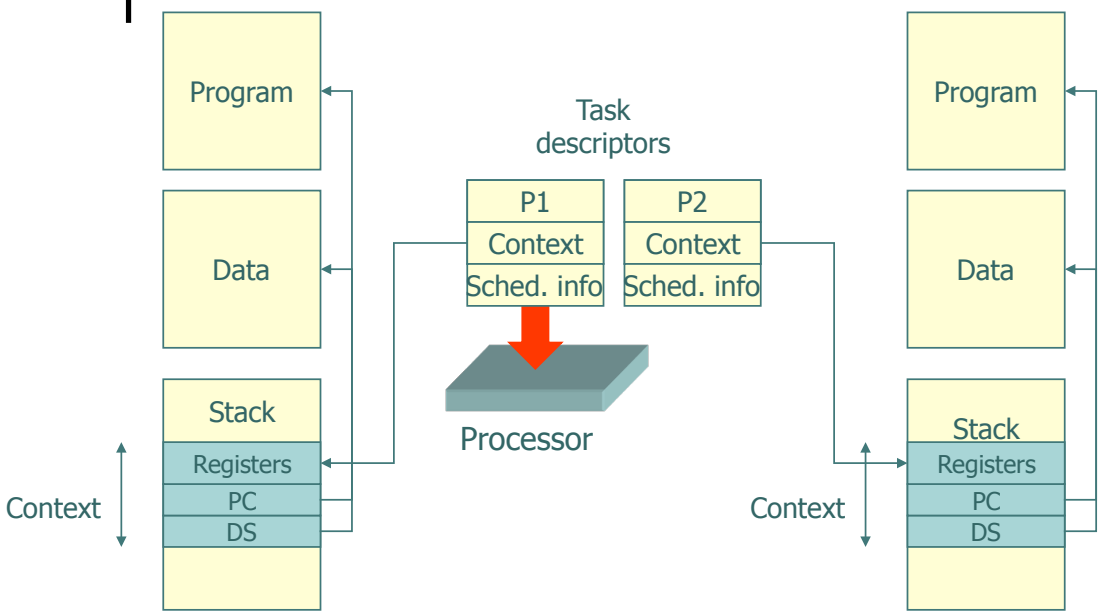
Types of services

- Task management
  - Creation, deletion, activation, stop, priority management
- Time management
  - Periodic task activation, bounded waiting
- Inter-task synchronisation (precedence, resource sharing)
  - Creation of synchronisation objects (ex: semaphores, signals), synchronisation primitives (signal/wait)
- Inter-process communication (IPC)
  - Creation of communication objects (ex: message queues), communication primitives (send/receive)
- Memory management
  - Memory allocation / deallocation (if any)
- Input/output (I/O) management

Task management
Terminology (1/2)

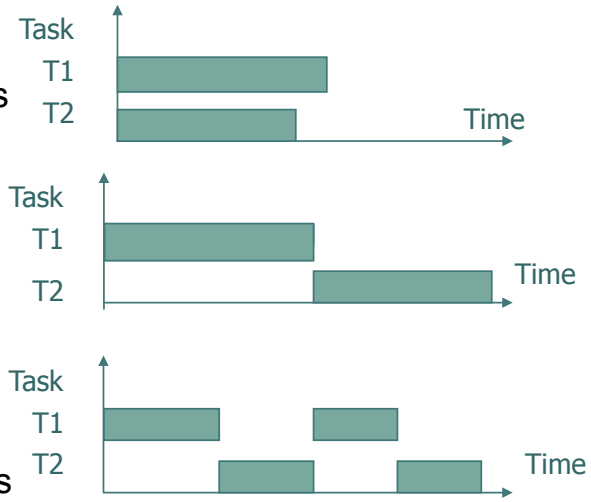
- Program
  - Static entity containing the executable code
- Process, task, thread
  - Dynamic entity:
    - Program (static component)
    - Data, stack, execution context (dynamic component)
  - Task descriptor: data structure grouping all information on the task (name, context, current status, etc.)
  - Thread vs task/process: threads do not have a private address space

Task management
Terminology (2/2)



Task management
Degrees of parallelism

- Real parallelism:
  - Multi processor systems only
- Pseudo-parallelism (uniprocessor)
  - Non preemptive
  - Preemptive
- Here, uniprocessor systems only

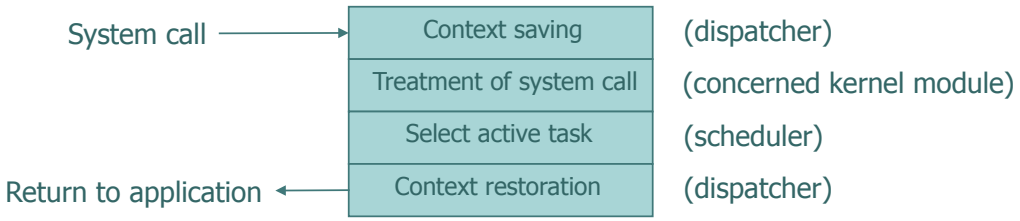


Task management
RTOS structure

- Scheduler:
  - Selection of the task to be scheduled according to a given scheduling policy
  - Data structure (list) to chain ready tasks
- Dispatcher (context switch):
  - Management of processor registers at context switch points
    - Saving of context (registers)
    - Restoration of context

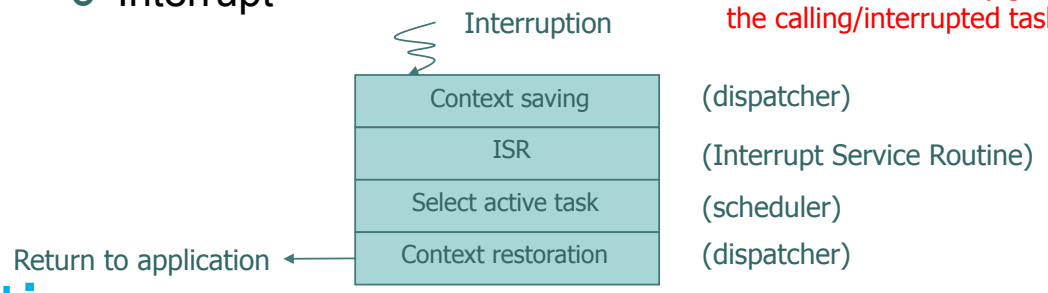
Task management
System calls and interrupts

- System call

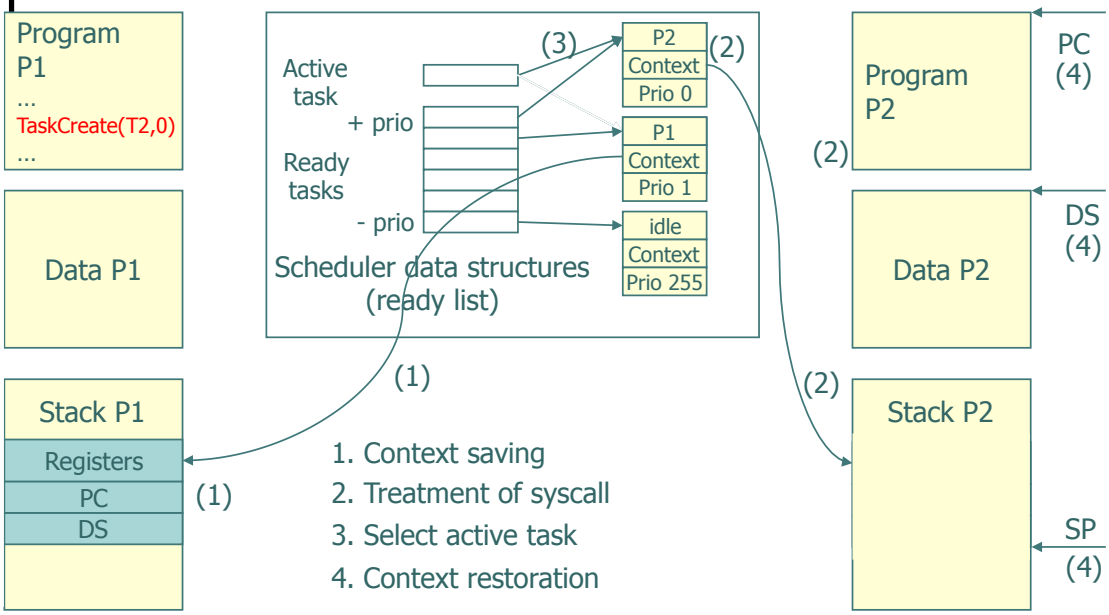


- Interrupt

Control does not necessarily go back to the calling/interrupted task

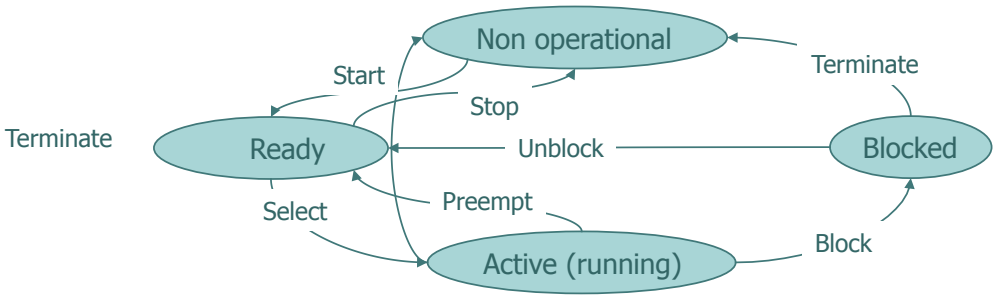


Task management
System call: example



Task management
Task states

- Processor sharing between tasks
  - Different task states: ready and active
- Synchronisation: task blocking (⇒ blocked state)
- State transition diagram



Terminate, Stop, Start: system calls (explicitly called by the tasks)
Block, Unblock: synchronisation system calls (explicitly called by the tasks)
Select, Preempt: scheduling decisions

Task management
Scheduling in RTOS: an overview

- Time triggered schedulers
  - Launch tasks according to a task dispatch table generated off-line by a scheduling tool
  - Proprietary RTOS mainly + OSEK Time specification from the OSEK/VDX consortium + Autosar/OS schedule tables
- Event triggered schedulers
  - Priority based preemptive scheduling
    - At any time, the processor is allocated to the higher priority task
    - Used by a large majority of commercial RTOSs
  - Time sharing scheduling
    - Processor allocation to a task for a fixed-length time slice
    - Real-time extensions of general-purpose operating systems or scheduling of tasks with identical priorities

Task management
Priority-based scheduling

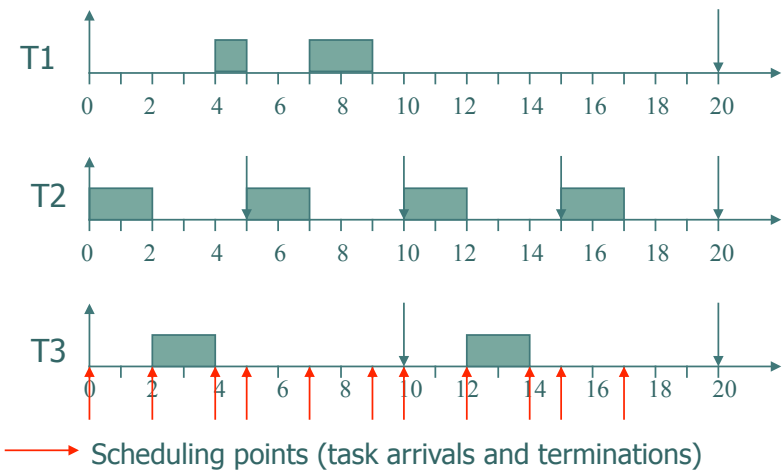
- At any time, the processor is allocated to the higher priority task, preemptive

(RM priority assignment : prio(T2) > prio(T3) > prio(T1)

C1 = 3
P1 = 20
D1 = 20

C2 = 2
P2 = 5
D2 = 5

C3 = 2
P3 = 10
D3 = 10

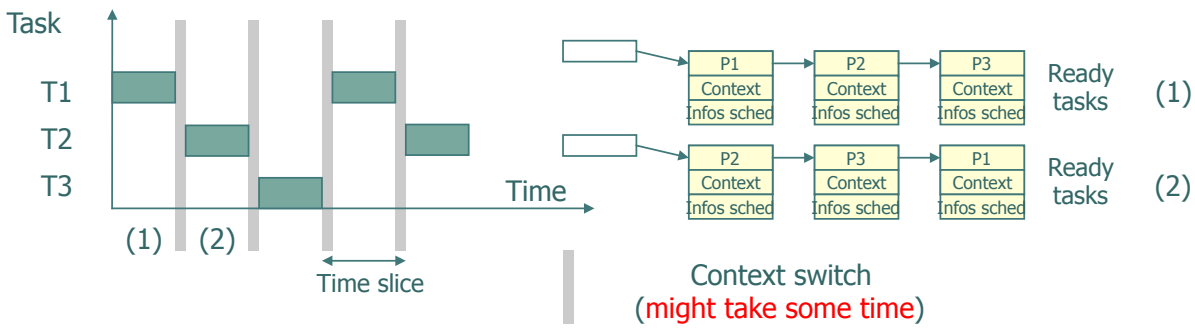


Task management
Priority-based scheduling

- Interest
  - Timing constraints impact scheduling decisions (task “importance” is coded into task static or dynamic priorities)
  - Numerous feasibility tests exist
- Limitations
  - Context switch cost
- Remarks
  - Coding timing constraints into priorities is the user responsibility
  - The concept of timing constraints is not explicit
  - No run-time verification of deadline violations

Task management
Round-robin scheduling (1/2)

- Principle
  - Processor allocation to a task for a fixed-length time slice



Task management
Round-robin scheduling (2/2)

- Interest
  - Fair processor allocation among tasks
- Limitations
  - Timing constraints do not impact scheduling decisions
  - Few feasibility conditions
- Domains of use
  - Real-time extensions of general-purpose operating systems
  - Scheduling of tasks with identical priorities in RTOSs (ex: VxWorks Wind kernel)

Task management
Typical API (Application Programming Interface)

- Task creation (non operational → ready)
- Task termination (→ non operational)

- Task self suspension (ready → blocked)
- Task reactivation (blocked → ready)

- Task priority change
  - Allows the implementation of dynamic priority scheduling policies, not provided by default
  - Not available in all RTOSs

Task management
API of OSEK/VDX and VxWorks

Primitive	OSEK/VDX	VxWorks

Start	ActivateTask(tid);
	
	Schedule(); /* *Explicit re-scheduling request* */
		taskInit(adtcb,Nname,Prio,Opt,adstack,StSize,adstart, …);
		tid = taskSpawn(Name,Prio,Opt,,StSize,adstart);
		taskActivate(tid);

Termination	TerminateTask(); ChainTask(tid); /* *Combined termination and activation* */	taskDelete(Tid);
Priority change	None (priorities cannot be modified)	taskPrioritySet(Tid,priority); taskPriorityGet(Tid,adpriority);
Suspension / restart	None	taskSuspend(Tid); taskResume(Tid);

Task management
Selection of a scheduling policy

- Off-line
  - No need for a multi-task RTOS
  - No context switch costs
- On-line
  - Static priorities: priorities assigned at task creation time
    - Straightforward implementation
  - Dynamic priorities
    - More complex implementation
    - EDF example: need to sort tasks by increasing deadline at every task arrival (n log n, with n number of tasks in ready queue)

Time management
Typical services

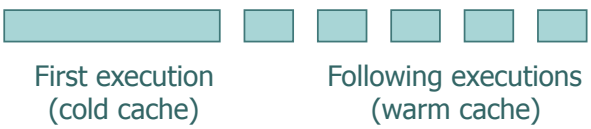
- Read/update primitives
  - Hardware based: real time clock (granularity is hardware-dependent)
  - Software based: periodic update of system clock in the clock interrupt handler (tick)
    - Clock granularity may be important (ms)
    - Some systems provide configurable granularities (with higher system overheads)
- Trigger actions at absolute dates
  - Possible actions: signal an event, execute a task
  - Possible recurrences: once, periodically

Time management
Typical services

- Trigger actions at relative dates
  - Timeouts when using blocking synchronisation primitives (semaphores, message queues)
  - Signal an event / start a task after a given delay
  - Pitfall: task start time ≥ arrival time. Use with care to launch tasks periodically

Time management
Typical services

- Execution monitoring
  - Example of VxWorks
    - timex(fn,args); measures the execution time of function fn (fn is executed once)
    - timexN(fn,args, N); measures the execution time of function fn (fn is executed N times)
  - Pitfalls:
    - Clock granularity
    - Hardware effects (caching)



Time management
The OSEK / VDX example

- Alarm = association between
  - A counter (ex: tick counter)
  - A task
  - An action on the task (activation / signal of an event).
  - Alarm attributes provided at design time (configuration file)
- Types of triggers
  - Cyclic or unique
  - On relative or absolute counter values
- Interface
  - SetRelAlarm(id_alarm,incr,cycle); */* cycle=0 : unique trigger */*
  - SetAbsAlarm(id_alarm,start,cycle);
  - CancelAlarm(id_alarm);
  - GetAlarm(id_alarm,&tick);

Time management
API in two RTOSs

Primitive	OSEK / VDX	VxWorks

Unique trigger	SetRelAlarm(aid,increment,0);
	SetAbsAlarm(aid,start,0)
		taskDelay(ticks);
		timer_create(CLOCK_REALTIME,event_j andler,&tid);
		timer_settime(tid,flags, /* ab or rel */,time, prevtime);

Periodic trigger	SetRelAlarm(aid,incr,cycle); SetAbsAlarm(aid,start,cycle);	
Cancel / delete	CancelAlarm(aid);	timer_delete(tid);

Timeouts	Aucun	wid = wdCreate();
		wdStart(wid,ticks,fnptr,param);
		wdCancel(wid);
		wdDelete();
		semTake(semid,timeout);
		msgQReceive(mid, buffer, maxNBytes, timeout);

Read	GetAlarm(aid,&tick);	timer_gettime (tid,&time);

Synchronisation and communication

Types of constraints

- Precedence relations (hereafter simply “synchronisation”)
  - Ordering constraints between tasks
- Resource sharing
  - To share variables or peripherals between tasks
- Communications
  - Precedence + resource sharing (producer/consumer scheme)
  - Combination of the two above relations

- Tools and system calls for three types of constraints

Synchronisation

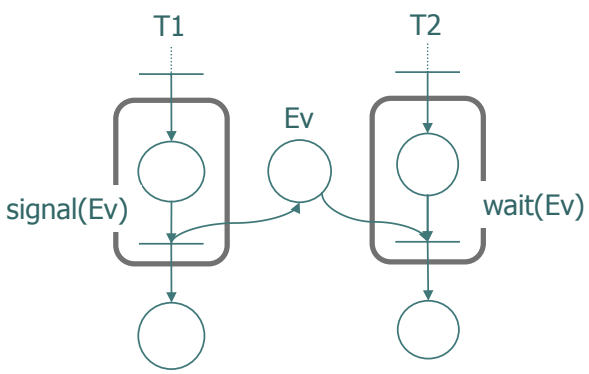
- Role
  - Block until an event is signalled
- Source of events
  - Hardware: sensors
  - Software: intra-application synchronisation
- Classification
  - Memorisation
    - Without memorisation (volatile event)
      - If a task is waiting for the event, unblock it, else the event is lost
    - With memorisation (persistent event)
      - No counting (events): event = boolean value (set / not set)
      - Counting (semaphores): counter of signals

Synchronisation

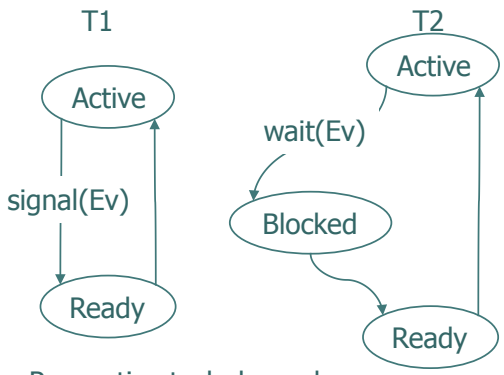
- Classification (continued)
  - Direct / indirect
    - Direct : signal to a precise task (signal(Ev,T);)
    - Indirect : global (signal(Ev);)
  - Synchronous / asynchronous
    - Synchronous : system call to wait for the event (wait(Ev);)
    - Asynchronous : asynchronous execution of a routine when the event is signalled (connect_handler(Ev,function);)
  - Public / private
    - Private : private set of events per task (fixed number, no dynamic creation of events)
    - Public : events are global objects (dynamic creation of events)

Synchronisation

Behavioral description (Petri nets)

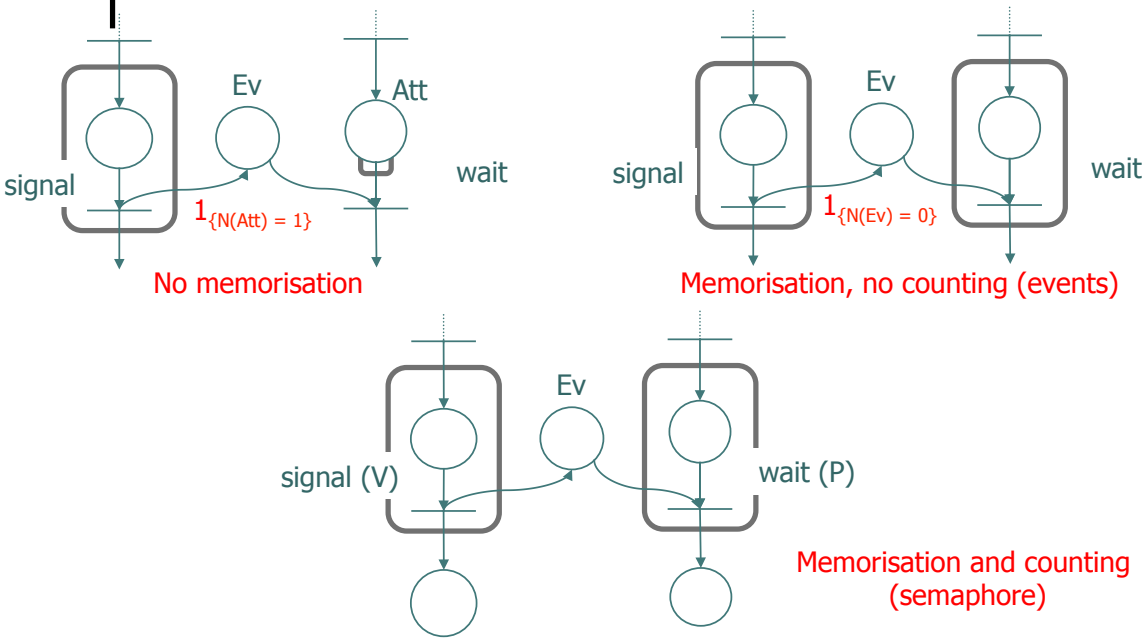


Impact of task states



Rq : active task depends on scheduling policy

Synchronisation

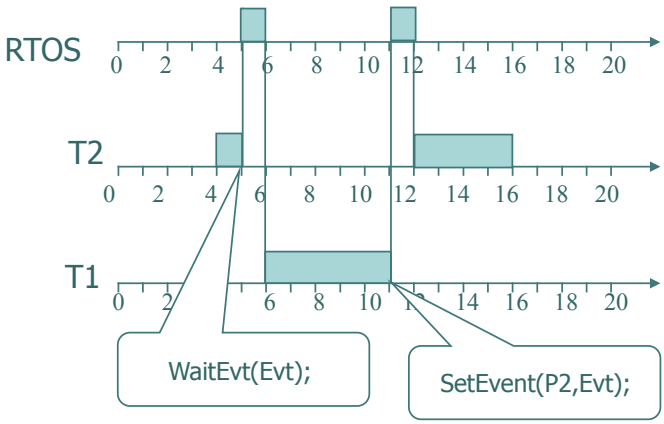


Event-based synchronisation
The OSEK/VDX API

- Types of events
  - Memorisation, no counting, explicit clearing of events, synchronous, direct signals
  - Private set of events per task (bitmap)
- Interface
  - SetEvent (tid,mask): signals all events of task *tid* identified by bitmap *mask* (1 = set, 0 = unchanged)
  - ClearEvent (mask): clear all events identified by bitmap *mask*
  - WaitEvent (mask): block until all events identified by bitmap *mask* are signalled (no timeout)
  - GetEvent(tid,&mask): returns in bitmap *mask* the current set of signalled events of task *tid* (regardless of the events task *tid* is waiting for)

Event-based synchronisation
The OSEK/VDX API

- Example (T2 has the higher priority)



Event-based synchronisation
Implementation

- Memorisation, no counter, implicit event clearance, public events

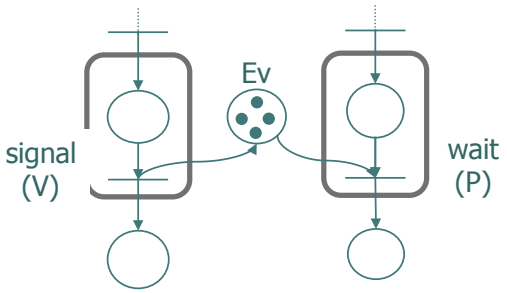
```
typedef struct {
      int state;          /* Ready, blocked, running */
      t_ctx context;      /* Task context */
} TCB;                    /* Task descriptor */
TCB T[MAX_TASK];          /* Task descriptors */
char Ev[MAX_EVT];         /* Event descriptors : Set/Clear */
```

```
Signal(int tid, int e) {
    Context saving;
    Ev[e] = Set;
    if (T[tid].state == Blocked) {
        T[tid].state = Ready;
        Ev[e] = Clear;
    }
    Call scheduler;
    Context restoration;
}
```

```
Wait(int e) {
    Context saving;
    if (Ev[e] == Set) {
        Ev[e] = Clear;
    }
    else T[Active].state = Blocked;
    Call scheduler;
    Context restoration;
}
```

Semaphore-based synchronisation
Basics

- Typical API
  - sid = create_sema(cpt); /* *cpt >= 0* */
  - P (sid); /* *Puis-je ?* */
  - V (sid); /* *Vas - y* */
- Behaviour



Initial value = initial number of tokens in place Ev (number of calls to P before blocking)

Semaphore-based synchronisation
Implementation

```
typedef struct {
        int cpt;        /* Counter
                        ≥ 0 = number of P before blocking (not yet consumed events)
                        < 0 = number of tasks in wait queue */
        queue *F;       /* Wait queue */
} t_sema;    /* Semaphore descriptor */
```

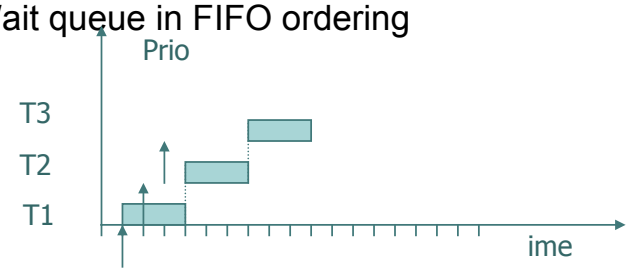
```
V (t_sema *s) {
    Context saving;
    (s->cpt) ++;
    if (s->cpt ≤ 0) {
        T = remove(s->F);
        T->state = Ready;
    }
    Call scheduler;
    Context restoration;
}
```

```
P (t_sema *s) {
    Context saving;
    (s->cpt) --;
    if (s->cpt < 0) {
        Active.state = Blocked;
        insert(Active,s->F);
    }
    Call scheduler;
    Context restoration;
}
```

Synchronisation
Impact of queue management policy (1/2)

Illustration on semaphore-based synchronisation

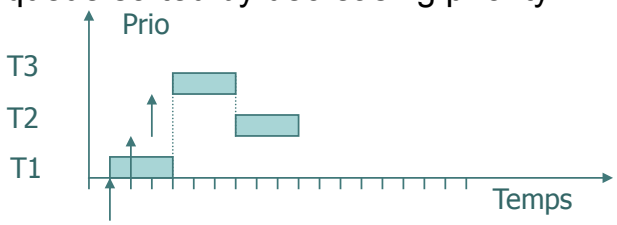
- Code
  s = create_sema(1);
  T1 : P(s); A1; V(s);
  T2 : P(s); A2; V(s);
  T3 : P(s); A3; V(s);
  Arrival date of T1 = 1, T2 = 2, T3 = 3.
  prio(T1) < prio(T2) < prio(T3); Task duration = 3 time units
- Wait queue in FIFO ordering



Synchronisation
Impact of queue management policy (2/2)

Illustration on semaphore-based synchronisation

- Wait queue sorted by decreasing priority



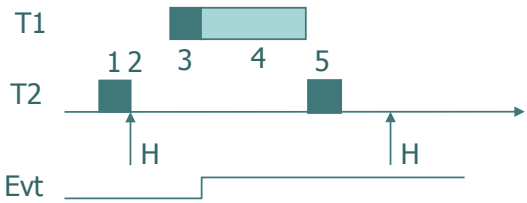
- FIFO wait queue: ensures fairness, avoids starvation
- Wait queue managed by priority: fairness is not ensured anymore, favours the higher priority tasks → used in most RTOS
  - VxWorks: configurable wait queue policy (FIFO or priority)
  - Applicable to all blocking synchronisation tools

Synchronisation
Need for atomicity (1/2)

```
T1 {
  cycle
    Wait(H);
    Signal(T2,Evt);
    Code_T1;
  fincycle;
}
```

```
T2 {
  cycle
    Wait(Evt);
    Code_T2;
  fincycle;
}
```

Initial state : Evt = Clear; T1 blocked (on H), T2 active




```
Wait(int e) {
    Context saving;
    if (Ev[e] == Set) Ev[e] = Clear;
    else T[Active].state= Blocked;
    Call scheduler + context restoration;
}
```

H

1. T2 tests Evt and finds Evt as Cleared
2. H arises → preemption of T2 by T1
3. T1 signals Evt. T2 is not awakened since T2 is not (yet) blocked
4. T1 executes its code
5. T2 resumes its execution and blocks because of the test done in step 1.
   → T2 does not detect Evt

⇒ ATOMIC execution of primitives

Synchronisation
Need for atomicity (2/2)

- Atomicity
  - Required for all synchronisation tools (events, semaphores, message queues)
- Enforcing atomicity
  - Masking interrupts
    - Easy to implement, fast (cli / sti on x86)
    - Applicable on uniprocessors only (shared bus, interrupts are local to a processor)
  - Dedicated instructions (Test-and-set, exchange)

Event-based synchronisation
API of OSEK/VDX and VxWorks

Primitive	OSEK/VDX	VxWorks (non exhaustive)
Creation	None (Fixed number of events per task)	oldhandler = signal (signo, ,handler);
Signal	SetEvent(tid,mask);	kill(tid,signo); raise(signo); /* Auto-signal */
Wait / Clear / Read	WaitEvent(mask); ClearEvent(mask); GetEvent(tid,&mask);	sigsuspend(mask); sigtimedwait (mask,info,timeout);
Deletion	None	

Semaphore-based synchronisation
API of OSEK/VDX and VxWorks

Primitive	OSEK / VDX	VxWorks
Creation	No semaphore	sid = semCCreate(Opt,Initialcount);
Deletion		semDelete(sid);
Wait		semTake(semid,timeout);
Signal		semGive(semid); semFlush(semid); /* *Flushes the wait queue* */
Read		semShow(semid,level);

Resource sharing

- Shared variable → one task must not read / write the variable while in use by another task
  - Concept of critical section
  - Need to enforce mutual exclusion

Resource sharing
Implementing a critical section (1/5)

- Interrupt masking
  - Easy to implement, fast (cli/sti)
  - Additional latency in interrupt handlers (at worst, interrupts might be lost)
  - Usable in uniprocessors only
  - → Usable for short, bounded critical sections
- Temporary increasing the priority of the task using the resource
  - Tasks not using the resources are delayed (and might be more “important”)

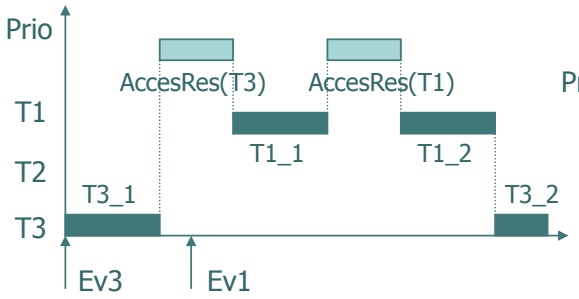
Resource sharing
Implementing a critical section (2/5)

T3_1;
AccesRes;
T3_2;

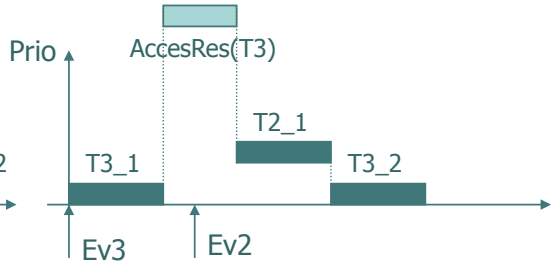
T1_1;
AccesRes;
T1_2;

T2_1;

Prio(T1) > Prio(T2) > Prio(T3)



Scenario 1: Ev1, then Ev3



Scenario 2: Ev1, puis Ev2
T2, which does not use Res, is blocked (higher prio!)

Resource sharing
Implementing a critical section (3/5)

- Use of busy waiting (never do that!)

```
void Request (char *V)
{
        while (*V) ; /* Wait for resource */
        *V =  1;
}
```

```
void Release (char *V)
{
        *V =  0; /* Signal */
}
```

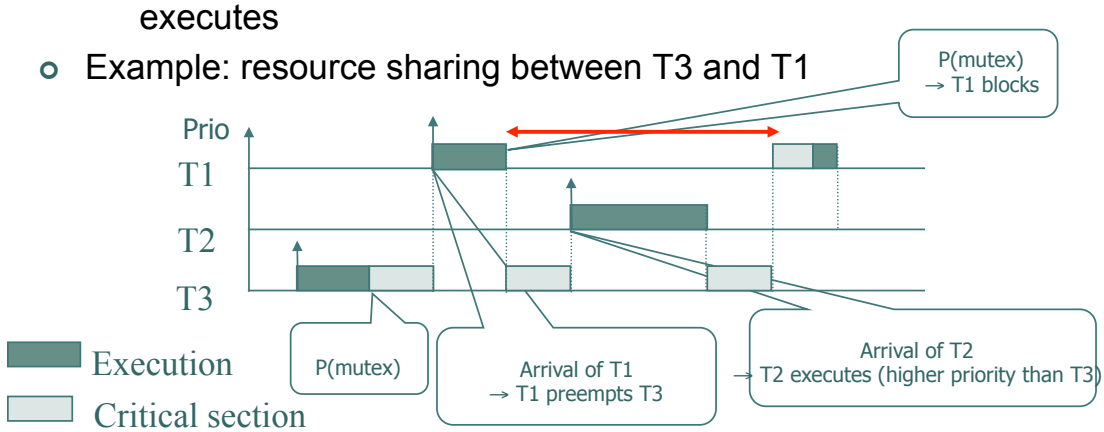
- Atomicity needs to be ensured (not shown here)
- Problems
  - Processor monopolisation
  - Starvation when a high priority task waits for a resource

Resource sharing
Implementing a critical section (4/5)

- Use of semaphores
  - Request(v) → P(s_v);
    Release(v) → V(s_v);
  - Counting semaphores (initial value of counter = 1)
  - Mutual exclusion semaphores (or binary semaphore): specialisation of semaphores:
    - No counter (boolean value instead)
    - Checks that Request (P) and Release (V) are called by the same task (ex: VxWorks)
    - Nested calls to P (without blocking) when done by the same task (ex: VxWorks)

Resource sharing
Priority inversion

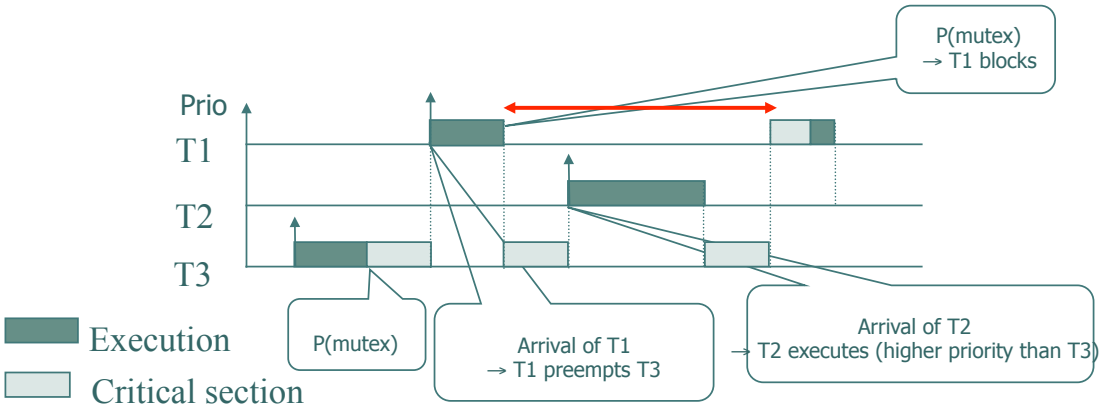
- Definition of priority inversion
  - A higher priority task is blocked while a lower priority task executes
- Example: resource sharing between T3 and T1



T1 blocked during the whole execution of T2. Blocking duration is unbounded (multiple arrival of intermediate-priority tasks).

Resource sharing
Priority inversion

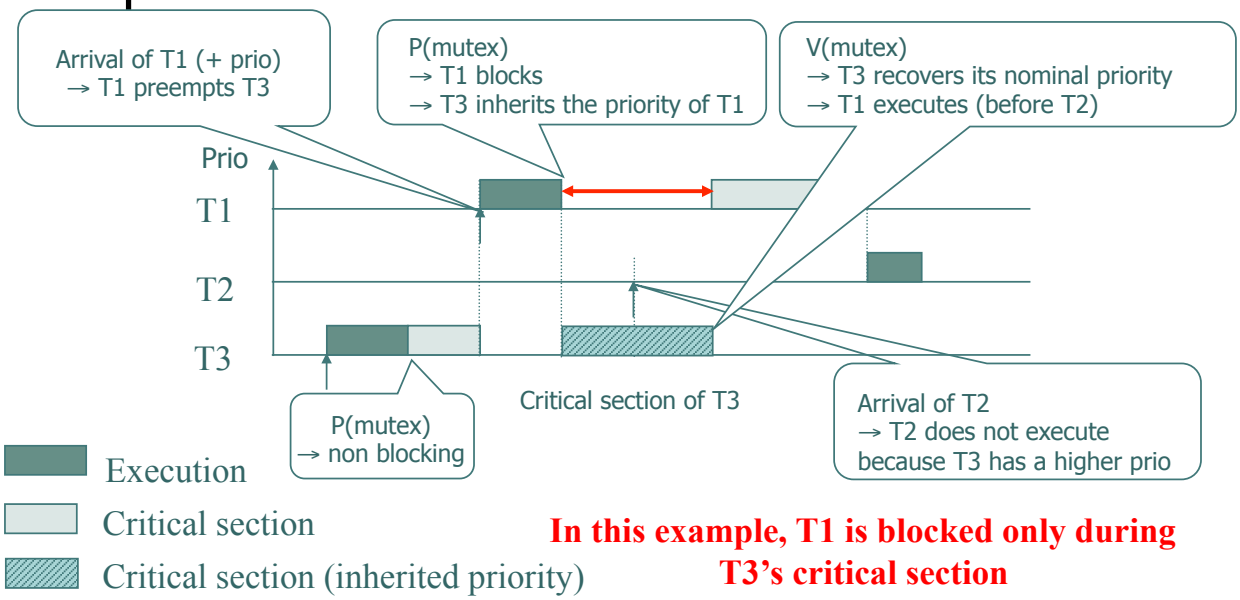
- Priority-sorted wait queues do not solve the problem



Resource sharing
Solutions to priority inversion : PIP (1/9)

- PIP stands for Priority Inheritance Protocol, Sha, Rajkumar, Lehoczky, 90
- Assumptions
  - Fixed priority scheduling
  - Nominal priority vs current priority
  - Use of mutual exclusion semaphores, "properly nested" critical sections
- Protocol definition
  - 1. Scheduling rule: use of nominal priority, except in case 3
  - 2. Resource allocation rule
    - Resource is free: no blocking
    - Resource not free: blocking
  - 3. Priority inheritance rule: when a task T is blocked on a mutex, the blocking task Tl inherits the current priority of task T. Tl uses the inherited priority until the mutex in freed; Tl then executes at the priority of the highest priority task still blocked on the mutex.

Resource sharing
Solutions to priority inversion : PIP (2/9)



In this example, T1 is blocked only during T3's critical section

Resource sharing
Solutions to priority inversion : PIP (3/9)

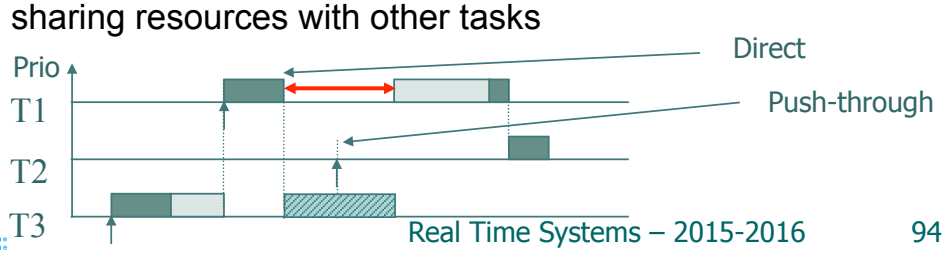
- Properties
  - Bounded blocking time (except in case of deadlock)
    - ➔ Allows integration in feasibility tests
  - Does not avoid deadlocks
  - Implementation of PIP in a RTOS does not require *a priori* knowledge of resource usage (but required to compute blocking times)
- Remark
  - Does not remove priority inversions, only bound their duration

Resource sharing
Solutions to priority inversion : PIP (4/9)

- Remark: transitive priority inheritance in case of nested critical sections
  - If T shares a resource R with a lower priority task T'
  - A critical section on R' is included in the critical section on R
  - If R' is used by a task T'' with priority lower than T'
  - T' depends on T'' and by transitivity, T depends on T''
- Blocking time
  - Definition: time interval during which a task T does not progress due to a lower priority task
  - Remark: a ready task can suffer from blocking due to the blocking time definition

Resource sharing
Solutions to priority inversion : PIP (5/9)

- Sources of blocking
  - Direct blocking: when a task T requests a resource used by a lower priority task
    - Suffered by a high priority task sharing a resource with a lower priority task
  - Indirect (push-through) blocking: when a task T is blocked by a lower priority task having inherited a priority higher than the one of task T
    - Suffered by intermediate-priority tasks do not necessarily sharing resources with other tasks



Resource sharing
PIP: computation of blocking times (6/9)

- Computation of blocking times
  - For non nested critical sections only
  - Complexity of $O(m\ n^2)$
  - Tighter method (lower blocking time bound) of exponential complexity [Rajkumar, 91]

Resource sharing
PIP: computation of blocking times (7/9)

- Computation of maximum blocking time suffered by task Ti

1. For every task Tj with priority lower than Ti
   - Identify the critical sections of Tj that can block Ti and take the longest one
   - Sum for all Tj the critical sections durations identified above
2. For every mutual exclusion semaphore sm
   - Identify the critical sections protected by sm that can block task Ti, and take the longest one
   - Sum for all sm the critical section durations identified above

- The maximum blocking time of Ti is the minimum of these two values

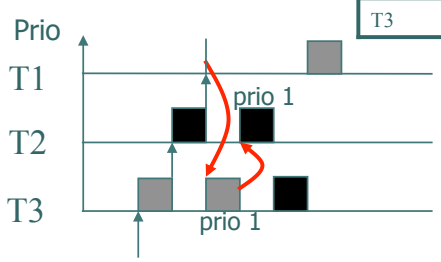
Resource sharing
PIP: computation of blocking times (8/9)

- Computation of maximum blocking time (task T1)
  - Item 1
    - Task T2
      - No direct blocking (no shared resource between T1 and T2)
      - No push-through blocking (neither T2 nor T3 can inherits priorities higher than T1's priority)
    - Task T3
      - Direct blocking (R shared between T1 and T3)
      - No push-through blocking
  - Item 2
    - One mutex used (to protect R). Can block (directly) T1 during the duration of the critical section of T3.
- Minimum = 1 critical section (the critical section of T3)

Resource sharing
PIP: last details (9/9)

- Remark
  - According to the definition of blocking times, the task with the lowest priority will always have a null blocking time
- Example

Task	Arrival Ai	WCET Ci	prio(Ti) (1 = +prio)	Shared resources
T1	2	1	1	[Grey;1]
T2	1	2	2	[Black;2]
T3	0	3	3	[Grey;2 [Black;1]]



Blocking time of task T3 = 0
Suspensions correspond to execution of higher priority tasks

Resource sharing
Solutions to priority inversion : PCP (1/6)

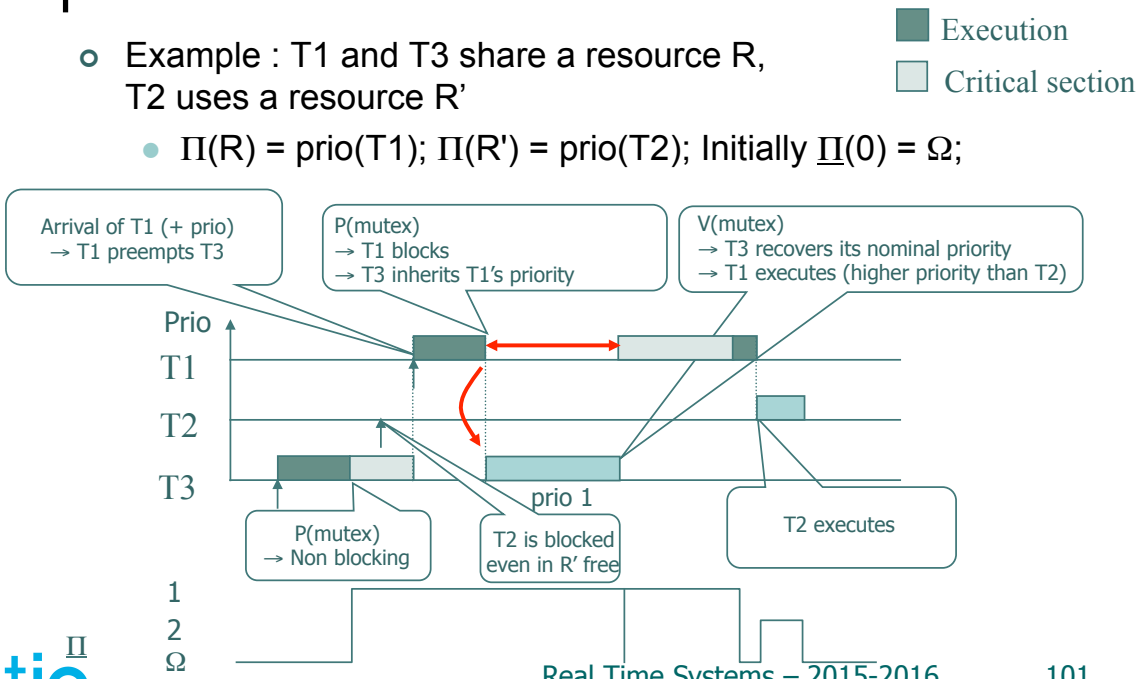
- PCP = Priority Ceiling Protocol
  - ex: OSEK/VDX
- Assumptions
  - Fixed priority scheduling
  - Off-line knowledge of resource usage (in contrast to PIP)
  - Nominal vs current priority
- Definitions
  - Priority ceiling of resource R, $\Pi(R)$ = maximum priority of the tasks known to use R
  - Current priority ceiling at time t, $\underline{\Pi}(t)$ = maximum ceiling of resources effectively used at time t

Resource sharing
Solutions to priority inversion : PCP (2/6)

- Protocol definition
  - 1. Scheduling rule: use of nominal priority, except in case 3
  - 2. Resource allocation rule (applied when T requests R)
    - Resource not free: blocking
    - Resource is free:
      - T has a current priority strictly higher then $\underline{\Pi}(t)$, no blocking
      - Else, R is allocated to T if T holds the resource of ceiling $\underline{\Pi}(t)$. Else T is blocked
  - 3. Priority inheritance rule: when a task T is blocked on a mutex, the blocking task Tl inherits the current priority of task T (Pt). Tl executes at priority Pt until Tl releases all resources with ceiling higher or equal to Pt. Tl then executes at the priority it had when asking for the resource.

Resource sharing
Solutions to priority inversion : PCP (3/6)

- Example : T1 and T3 share a resource R, T2 uses a resource R'
  - $\Pi(R) = prio(T1)$; $\Pi(R') = prio(T2)$; Initially $\underline{\Pi}(0) = \Omega$;



Resource sharing
Solutions to priority inversion : PCP (4/6)

- Properties
  - Bounded blocking time : bound = duration of only one critical section
  - Maximum blocking time duration for a task Ti (Bi)
    - Duration of the longest critical section among those belonging to tasks with lower priority than Ti and guarded by a semaphore with ceiling higher or equal to the priority of Ti, $\Pi(s) \geq prio(Ti)$
      - Identify critical section in lower priority tasks that can block Ti and with $\Pi(s) \geq prio(Ti)$
      - Blocking time = maximum duration of the critical sections identified above
  - Maximum locking time lower than the one of PIP
  - Deadlocks are prevented
- Requires the programmer to declare resource usage at design time
- Does not remove priority inversions, only bound their duration

Resource sharing
Solutions to priority inversion : PCP (5/6)

- Computation of maximum blocking time: example

	S1 (Π = T1)	S2 (Π = T1)	S3 (Π = T2)
T1	1	2	0
T2	0	9	3
T3	8	7	0
T4	6	5	4

  - T1 : critical sections that can block T1
    - No push-through blocking (T1 has the highest priority)
    - Direct blocking : T2(9), T3(8), T3(7), T4(6), T4(5)
    - So, B1 = max(8,6,9,7,5) = 9

Resource sharing
Solutions to priority inversion : PCP (6/6)

- Computation of maximum blocking time: example

	S1 (Π = T1)	S2 (Π = T1)	S3 (Π = T2)
T1	1	2	0
T2	0	9	3
T3	8	7	0
T4	6	5	4

  - T2 : critical sections that can block T2
    - Direct blocking : T3(7), T4(5), T4(4), push-through : T3(8), T4(6)
    - So, B2 = max (8,6,7,5,4) = 8
  - T3 : critical sections that can block T3
    - Direct blocking : T4(6), T4(6), push-through : T4(4)
    - So, B3 = max (6,5,4) = 6;
  - T4 : B4 = 0;

Resource sharing
Mutexes in OSEK/VDX and VxWorks

Primitive	OSEK / VDX	VxWorks
Init	None (static allocation)	sid = semBCreate(options,initialState); sid = semMCreate(options);
Deletion	None	

Acquire	GetResource(resid);	semTake(sid,timeout);
		taskLock(); /* Disable context switch */
		intLock();

Release	ReleaseResource(resid);	semGive(sid);
		taskUnlock(); /* Unable context switch */
		intUnlock();

PCP (ceiling computed at design time)

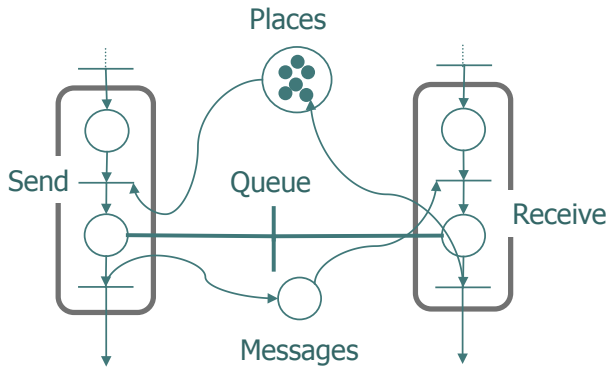
PIP, configurable management of wait queues (FIFO/prio)

Inter-process communications
(uni-processor)

- Classification
  - Synchronous / asynchronous
    - synchronous: explicit call to blocking message receive routine
    - asynchronous: message handler asynchronously called upon message arrival
  - With message queue / without message queue
    - with: not yet consumed messages are stored in a queue (mailbox)
      - Bounded / unbounded queue
      - Queue management policy : FIFO / priority based
    - without : overwriting of previous contents (« black board »)
  - With / without blocking upon full queue
    - Non blocking → possible message loss in case of bounded message queue

Inter-process communications
Mailbox, synchronous, bounded message queue, no loss

- Typical interface
  - Send(message,task);
  - Receive(message);
- Behaviour



Inter-process communications
Mailbox, synchronous, bounded message queue, no loss

- Implementation: synchronisation issues
  - Blocking when sending / receiving a message
    - Sending: full message queue
    - Receiving: empty message queue
  - Protection of message queue: mutual exclusion

Inter-process communications
Mailbox, synchronous, bounded message queue, no loss

- Implantation with semaphores
  - Data structures

```
typedef char t_msg[MAX_MSG]
typedef struct {
        t_sema s_messages init 0;        /* Counts the number of messages */
        t_sema s_places init N_PLACES;   /* Counts the number of free slots */
        t_msg buffer[N_PLACES];          /* message queue */
        t_sema s_buffer init 1;          /* Mutex on message queue */
        int i_msg init 0;                /* Next message to consume */
        int i_place init 0;              /* Next place to fill-in */
} t_port;
```


Inter-process communications
Mailbox, synchronous, bounded message queue, no loss

```
void send (t_msg msg, t_port *port)
{
  /* Wait for a free slot in msg queue */
  P(port->s_places);

  /* Copy message in msg queue */
  P(port->s_buffer);
  copy(&msg,port->buffer[port->i_places];
  port->i_places = (port->i_places + 1) modulo
          N_PLACES;
  V(port->s_buffer);

  /* Signal message arrival */
  V(port->s_messages);
}
```

```
void receive (t_msg *msg, t_port *port)
{
   /* Wait for a message */
   P(port->s_messages);

   /* Copy message from msg queue */
   P(port->s_buffer);
   copy(port->buffer[port->i_msg],msg);
   port->i_msg = (port->i_msg + 1) modulo
          N_PLACES;
   V(port->s_buffer);

   /* Signal a free slot in msg queue */
   V(port->s_places);
}
```

Inter-process communications
Mailbox API in two RTOSs

Primitive	OSEK / VDX	VxWorks
Init	InitMessage(mid,add);	mid = msgQCreate(maxMsgs, maxMsgLength, options);
Deletion	None	msgQDelete (mid);
Send	SendMessage(mid, add); SendDynamicMessage(mid,add,length); SendZeroMessage(mid,add);	msgQSend (mid, buffer, nBytes, timeout, priority);
Receive	ReceiveMessage(mid, add); ReceiveDynamicMessage(mid,add,length);	msgQReceive(mid, buffer, maxNBytes, timeout);
Read	GetStatus(mid);	nmess = msgQNumMsgs(mid);

Interrupt management

- Terminology
  - Interrupt request (IRQ)
  - Interrupt Service Routine (ISR): executed (asynchronously) upon IRQ arrivals
  - Some architectures provide priorities between IRQs
  - In general the scheduler is called before return from interrupt
- Remarks
  - An ISR in not a task (no descriptor)
    - Not allowed to call blocking primitives
  - Should be as short as possible (interrupts are masked in general in ISRs)

Interrupt management

- I/O management
  - Using interrupts
    - Pros: reactive
    - Cons: possibility of bursts of IRQ → difficult to check the system feasibility
  - Polling : periodic scanning of peripheral controller registers
    - Pros: the I/O management activity is predictable → easy to check the system feasibility
    - Cons: lack of reactivity

Interrupt management
Typical primitives

- Connect an ISR to an interrupt level
- Mask / unmask interrupts
  - Can be used to implement short duration critical sections
  - Some RTOS provide support for nested non interruptible regions

Interrupt management
Interruptibility of kernel

- Non preemptable kernel (interrupts masked during kernel execution)
  - Atomic system calls (by construction)
  - Predictability issues
    - Jitter when executing ISRs
    - Maximum jitter might be long (longest system call)
- Preemptable kernel
  - Predictability issues
    - No (or less) jitter
  - Atomicity of systems calls harder to enforce
  - Current state of practice
    - Masking of interrupts on short and bounded kernel sections
    - Provision of bounds to the kernel user

Interrupt management
API in two RTOSs

Primitive	OSEK / VDX	VxWorks
Masking	DisableAllInterrupts(); */* No nesting */* SuspendAllInterrupts(); */* Nesting */*	intLock();
Unmasking	EnableAllInterrupts(); */* No nesting */* ResumeAllInterrupts(); */* Nesting */*	intUnlock();
Connect handler	Not specified	intConnect(vector,routine,param);

Memory management

- Memory management issues
  - Static vs dynamic allocation
  - Protected vs non protected address spaces
  - What about demand paging and real time systems ?

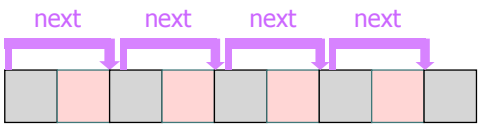
Memory management
Static vs dynamic

- Static memory management
  - No dynamic memory allocation primitive
  - Requires that the number and size of every object is known (or bounded) off-line (tasks, semaphores, message queues, tasks code and data regions, etc.)
- Evaluation
  - Pros:
    - No resource shortage at run-time (safety)
    - No memory fragmentation
    - No overhead due to dynamic memory allocation algorithm
  - Cons:
    - Rigid

Memory management
Static vs dynamic

- Dynamic memory management
  - Dynamic memory allocation primitives (malloc/free), dynamic creation of new objects (tasks, semaphores, etc.)
  - Example: “first-fit” memory allocation
    - List of free blocks stored inside free blocks
    - First-fit: allocation of the first big enough free block

```
for (b = h->first;  b != tail ; b = b->next) {
  if ( b->size >= size ) {
    break; /* block found */
  }
}
```

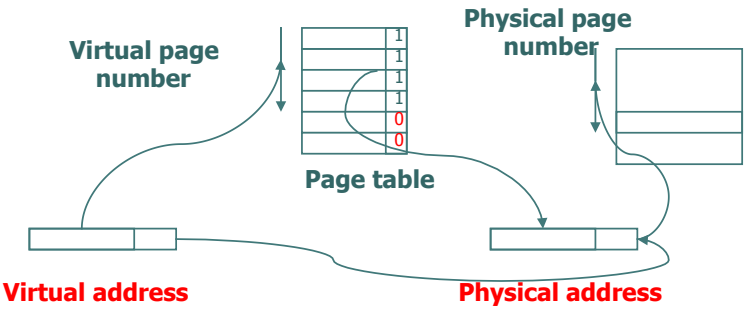


Memory management
Static vs dynamic

- Evaluation
  - Pros: flexible
  - Cons:
    - Memory fragmentation, possibly resulting in resource shortage at run-time
    - Predictability issues: allocation duration hard to bound
  - Considered as bad practice when programming real-time applications
    - Trade-off: use dynamic allocation at system start time

Memory management
Protected vs non-protected

- Protected memory management
  - Private address space per task
  - Hardware support required (MMU, segmentation)
  - Example: MMU and linear page tables



  - Note: RTOS support is required to share memory between tasks

Memory management
Protected vs non-protected

- Protected memory management
  - Evaluation
    - Pros:
      - Addressing errors are detected (safety)
    - Cons:
      - Predictability issue: address translation latency is hard to predict (use of a TLB, translation look-aside buffer)
      - Additional memory consumption (page tables)
      - Higher context switch latency

Memory management
Protected vs non-protected

- Non-protected memory management
  - No separate address spaces
  - Evaluation
    - Pros:
      - Predictability
      - Simpler hardware (cheaper)
      - Less memory consumption
    - Cons:
      - Possible memory corruptions in case of addressing errors
      - ex:

```
int tab[100];
for (i=0;i<100;i++) tab[2*i] = 0;
```


Memory management
What about demand paging?

- Definition
  - Pages are loaded on demand upon page faults
  - Pages are evicted from main memory when required
  - Interest: it's possible to execute applications not entirely fitting in main memory
- What about demand paging in real time systems?
  - Predictability issues: memory access time hard to bound (or bound is huge - page fault every memory access) → not suited to hard real time systems
- Trade-of (used in real time POSIX)
  - Use of page-locking primitives to selectively disable the paging activity

Memory management
API in two RTOSs

Primitive	OSEK / VDX	VxWorks (non exhaustive)

Allocation	None	ad = memalign (alignment, nbytes);
		ad = malloc(nbytes);
		ad = valloc(nbytes); /* *On page boundaries* */
		ad = calloc(nelem,elemsize);
		ad = realloc(adoldblock,newsize);

De-allocation	None	free(ad); cfree(ad);

A non exhaustive list of RTOSs

Outline

- Standards
  - Real time extensions of POSIX
  - OSEK/VDX: automotive industry
  - µitron: Japanese de-facto standard for embedded systems
- Commercial RTOS (non exhaustive list)
- Criteria to select a RTOS

Real-time extensions of POSIX

- POSIX
  - Standard of UNIX application programming interface
  - Standard IEEE 1003.1 (1987), normalising the API for managing:
    - Processes
    - Files
    - I/O
    - Process environment variables
- Different real time extensions of POSIX
  - 1003.4: real time extensions to POSIX 1003.1 (also named 1003.1b)
  - 1003.4a: multithread extensions of POSIX 1003.1

Real time extensions of POSIX : 1003.1b
Scheduling

- Scheduling
  - Priority based
  - Three scheduling policies, using separate ranges of priorities
    - SCHED_FIFO: priority based scheduling with FIFO scheduling when priorities are equal
    - SCHED_RR: priority based scheduling with round-robin scheduling when priorities are equal
    - SCHED_OTHER: other scheduling policy
  - Interface
    - sched_setparam(pid,&param); /* Param= priority for FIFO and RR */
    - sched_getparam(pid,&param);
    - sched_setscheduler(pid,policy,&param);
    - sched_getscheduler(pid);
    - sched_yield();
    - sched_get_priority_max/min(policy);

Real time extensions of POSIX : 1003.1b
Synchronisation

- Semaphores (counting semaphores)
  - sem_init(&sem,,shared,value);
  - sem_destroy(&sem);
  - sem_wait(&sem); sem_trywait(&sem);
  - sem_post(&sem);
  - sem_getvalue(&sem,&val);
  - Priority based wait queue management for SCHED_FIFO and SCHED_RR schedulers
- Signals

Real time extensions of POSIX : 1003.1b

Inter-process communication

- Message queues
  - Bounded message queue
  - flag to control if emissions and receptions are blocking or not
  - Priority based message queues, FIFO in case of identical priorities
  - mq_send(mq_id,ptr,size,prio);
  - mq_receive(mq_id,ptr,len,&prio);
  - mq_notify(mq_id,notification); /* To be notified of the message arrival through a signal */

Real time extensions of POSIX : 1003.1b

Memory management (1/2)

- Memory locking
  - mlockall(flags); /* Lock all pages */
    - Flag MCL_CURRENT: locking applies to all pages currently in memory
    - MCL_FUTURE: locking applies to all pages that will be loaded on demand in memory in the future (both flags must be set to lock the entire address space)
  - munlockall();
  - mlock(addr,size); /* Lock a set of consecutive pages */
  - munlock(addr,size);

Real time extensions of POSIX : 1003.1b

Memory management (2/2)

- Mapped files: give access to a file through a region in the task address space (does not lock the pages)
  - mmap(addr,len,prot,flags,filedesc,offset);
  - munmap(addr,len);
  - mprotect(addr,len,prot); /* Change access rights */
  - mmsync(addr,len,flags); /* Flush modifications to disk */
- Shared memory objects
  - shm_open(name,flag,mode); /* Returns a file descriptor, usable as a file afterwards */
  - shm_unlink(name);

Real time extensions of POSIX : 1003.1b

Time management

- Time management
  - clock_settime(clock_id,&time); clock_gettime(clock_id,&time);
  - clock_getres(clock_id,&res);
- Timers (sends a signal)
  - timer_create(clock_id,&sigevent,&tidem_id); timer_delete(tid);
  - timer_settime(timer_id,flag (absolute/not),&value,&ovalue); (value specifies a periodicity)
  - timer_gettime(timer_id,&value);
  - timer_getoverrun(timer_id); /* Indicates a double signal */
  - nanosleep(&delay,&reminder);
- Notes
  - Standardises the API, not the implementation

Commercial RTOSs
VxWorks

- Sold by WindRiver Systems
- Modular system built around the *wind* kernel
- Support for multi processors (VxMP), memory protection (VxMI), networking (VindNet) through dedicated libraries
- Rich set of target processors
- *wind* API + complete POSIX 1003.1b API (very rich API)
- Associated tools
  - Tornado: host development environment
  - Compiler and debugger
  - Simulator (VxSim)
  - Elaborated monitoring tools (WindView, StethoScope)
- Other kernel from the same company: OSEKworks

Commercial RTOSs
VxWorks

- Task management
  - Dynamic task creation
  - Preemptive priority based scheduling (256 priorities)
  - Round robin scheduling in case of identical priorities
  - Primitive to dynamically change task priorities
  - Primitive to temporarily inhibit scheduling
- Synchronisation
  - Mutual exclusion semaphores (mutex), binary semaphores, counter semaphores
  - Mutex: binary semaphore with 3 properties:
    - Priority inheritance (PIP)
    - Management of nested critical sections
    - Management of task abortion while inside a critical section
  - Binary semaphores : mutex without the 3 above-mentioned properties

Commercial RTOSs
VxWorks

- Communication
  - Mailboxes with variable-size messages
  - Pipes
  - Sockets (TCP/IP) and Remote Procedure Calls (RPC) VxWorks
- Time management
  - Timeouts with function called upon timeout expiration (base mechanism to launch tasks periodically)
  - Timeouts for blocking system calls (semaphores, messages)
- Memory management
  - Dynamic, optionnal support for protected address spaces (library)
- Support for multi processor architectures (library)

Commercial RTOSs
QNX Neutrino

- Sold by QNX
- Modular system built around the Neutrino kernel
- Micro-kernel architecture
  - Micro-kernel: basic services (scheduling, IPC, synchronisation). Communications with the RTOS modules through message passing
  - Modules for the additional services
- Fault tolerance services : memory protection + « high availability manager »: heartbeats, automatic checkpointing / restart, post-mortem analysis
- POSIX compliant interface
- Services
  - Java, file system (QNX, Linux, DOS, NFS, …), network (TCP/IP stacks, « tiny » and complete), device drivers (USB, audio, PCI, IDE, SCSI, …)

Commercial RTOSs
Other systems (1/2)

- Lynx-OS of LynuxWorks
  - Fully compliant with real time extensions of POSIX (1003.1, .1b,1c, multithreading extensions)
  - Linux binary compatibility
- Nucléus (Accelerated Technologies - Embedded systems division of Mentor Graphics)
  - Open-source, no royalties
  - Kernels : Nucléus PLUS, Nucléus OSEK, Nucléus μiPLUS
- eCOS : Embedded Configurable Operating System (Red Hat, Inc.)
  - Open-source kernel

Commercial RTOSs
Other systems (2/2)

- RTEMS
  - Multi-processor open-source jernel
  - Support and services provided by OAR (On-line Application Research)
  - POSIX, μitron, and RTEMS APIs
- Jaluna (Jaluna SA, formerly ChorusOS from Sun micro-systems)
  - Open-source since october 2002
  - Micro-kernel based : modular scheduler, virtual memory management, communication and synchronisation, portable device driver development environment
  - RT-POSIX API
  - Support for high availability

Criteria to select a RTOS (1/3)

- Is a RTOS really needed?
  - Pros: productivity gain (portability, scheduling)
  - Cons: needs more memory, time overhead
- Selection criteria
  - Supported target processors
  - Supported programming languages (most common : C,C ++,Ada)
  - Memory footprint (ROM, RAM)
    - Modular kernels → variable kernel size
    - Problem: identify the kernel size of a given set of required functionalities

Criteria to select a RTOS (2/3)

- Kernel services
  - Scheduling: must be suited to real-time applications
  - Synchronisation management: wait queue management (FIFO/priority), mechanisms to control priority inversion
  - Memory management: memory protection, static or dynamic management
  - Time management: ability to launch periodic tasks
  - Kernel preemptability
  - System modularity
- Additional services
  - Protocol stacks, real-time databases, web server, etc
  - Port time if such services are not available

Criteria to select a RTOS (3/3)

- Performance
  - Meaning of performance metrics (worst-case, average-case?)
  - ISR jitter
- Device drivers
- Development environment
- Technical support from the company, additional services (port to new processors, development of device drivers)
- Availability of source code
- Licence conditions, royalties
- Reputation and durability of the company
- Certification

Schedulability analysis

Temporal validation of real time systems

- Testing
  - Input data + execution (target architecture, simulator)
  - Necessary but not sufficient (coverage of worst case scenarios)
- Schedulability analysis (models)
  - Hard real-time: need for guarantees in all execution scenarios, including the worst-case scenario
  - Task models
  - Schedulability analysis methods (70s ⇨ today)

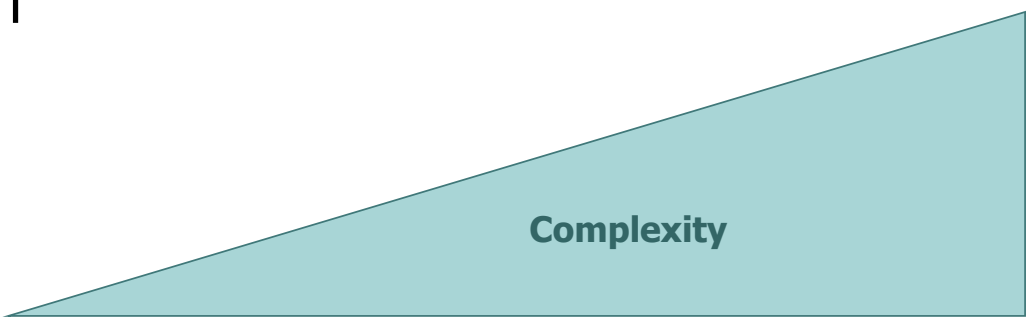
Schedulability analysis
Introduction (1/2)

- Definition
  - Algorithms or mathematical formulas allowing to prove that timing constraints will be met
- Feasibility
  - A schedule is feasible if all tasks meet their timing constraints
- Classification
  - Off-line validation
    - Target = hard real-time systems
  - On-line validation
    - Acceptance tests executed at the arrival of new tasks
    - Some tasks may be rejected → target = soft real-time systems

Schedulability analysis
Introduction (2/2)

- Input: system model
  - Task model
    - Arrival: periodic, sporadic, aperiodic
    - Inter-task synchronization: precedence constraints, resource sharing
    - Worst-case execution time (WCET)
  - Architecture
  - Known off-line for hard real-time systems
- Output
  - Schedulability verdict
  - Additional information (e.g. dispatch table)

Schedulability analysis
Complexity (1/2)



Monoprocessor periodic, synchronous, independent tasks

Multiprocessor

Distributed systems, aperiodic or dependent taks

In general, schedulability problems are NP-hard

Schedulability analysis
Complexity (2/2)

Monoprocessor			Multiprocessor		
Indep.	NoPr |Lmax Pr | Lmax NoPr,Ri | Lmax	Poly Poly NP-hard	**NoPr, 2 proc**	NoPr,prec,Ci=1 | Lmax NoPr,prec, ress NoPr, Ci <> 1 | Lmax NoPr, Ci=1, 1 ress	Poly NP-hard NP-hard NP-hard
Prec.	NoPr,prec | Lmax NoPr, prec, Ri | Lmax Pr, prec, Ri | Lmax	Poly NP-hard Poly	**NoPr, N Proc**	NoPr, prec, Ci=1	NP-hard
Res.	Per + sémas Pr,Ri,Ci=1 | Lmax NoPr, prec, Ri, Ci=1 | Lmax	NP-hard Poly Poly	**Pr, N proc**	Pr | mbmiss	NP-hard

(type | metric)

Schedulability analysis
Outline (1/2)

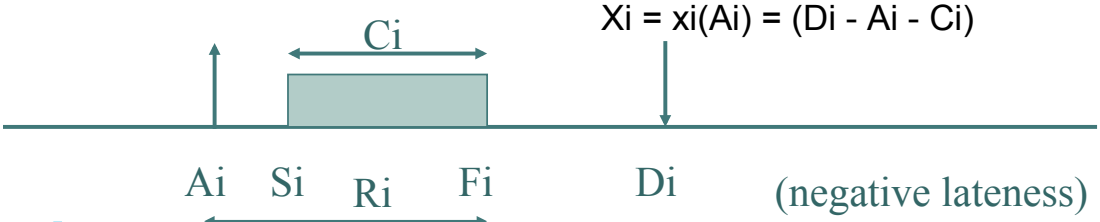
- Feasibility tests for periodic tasks and preemptive priority based scheduling
  - Assumptions and notations
  - Simulation
  - Necessary feasibility test (NC)
  - Fixed priority scheduling
    - Sufficient condition (SC) for Rate Monotonic and Deadline Monotonic
    - Exact condition (NSC) for fixed priority scheduling: Response Time Analysis
  - Dynamic priority scheduling
    - Exact condition for EDF when Di = Pi (NSC)
    - Exact condition for EDF when Di ≤ Pi: Processor Demand Analysis (NSC)

Schedulability analysis
Outline (2/2)

- Resource sharing
  - Fixed priority (RM,DM,RTA) and dynamic priority (EDF,Processor Demand) scheduling
- Accounting for RTOS overheads
- Feasibility tests for aperiodic tasks with known arrivals and non preemptive scheduling
- Worst-Case Execution Times (WCET) estimation

Feasibility analysis for periodic task systems
Notations (reminder - 1/2)

- Arrival — Ai
- Worst-case execution time in isolation — Ci
- Deadline (absolute or relative) — Di
- Start time — Si
- Finish time — Fi
- Response time — Ri = Fi - Ai
- Lateness — Li = (Fi - Di)
- Laxity (slack time) — xi(t) = Di - (t + Ci - ci(t))
  Xi = xi(Ai) = (Di - Ai - Ci)



Feasibility analysis for periodic task systems
Notations (reminder - 2/2)

- Task arrival
  - Periodic: infinite sequence of identical activations (jobs) regularly activated at a constant rate (Period, Pi)
    - Initial arrival date: offset (or phase) : Oi
    - Synchronous arrival: for all i,j Oi=Oj
    - Hyperperiod: scheduling cycle
      - synchronous task sets : interval [0,LCM(Pi)] for
      - else, interval [min(Oi), max(Oi,Oj+Dj) + 2 * LCM(Pi)]
- Critical instant : instant when a task arrival will cause its worst response time

Feasibility analysis for periodic task systems
Assumptions

- Explicit hypothesis
  - Periodic tasks of period Pi
  - Same WCET Ci for all task instances
  - Di = Pi
  - Independent tasks (no synchronisation/communication)
  - Synchronous arrivals (Oi=0)
- Implicit hypothesis
  - No self-suspensions
  - No release jitter, a task can execute immediately after arrival Ai
  - Null RTOS overhead (system call duration, context switch time)
- Unless explicitly mentioned, considered all along this chapter

Feasibility analysis for periodic task systems
Simulation-based validation

- Task periodicity → cyclic schedule
- Scheduling cycle (hyperperiod)
  - synchronous task sets : interval [0,LCM(Pi)] for
  - else, interval [min(Oi), max(Oi,Oj+Dj) + 2 * LCM(Pi)]
- Principle: simulate the task set on the hyperperiod using Ci as execution time
- Evaluation
  - Independent of scheduling strategy
  - Hyperperiod can be long
  - Suited to off-line verification
  - Safe for the considered task hypotheses, not in general (scheduling anomalies)

Feasibility analysis for periodic task systems
Necessary condition

- Processor load

$$U = \sum_{i=1}^{n} \frac{Ci}{Pi}$$

- Necessary feasibility condition : U ≤ 1
  - This condition is not sufficient (not for all scheduling policies)

Feasibility analysis for periodic task systems
Sufficient condition for Rate Monotonic (1/2)

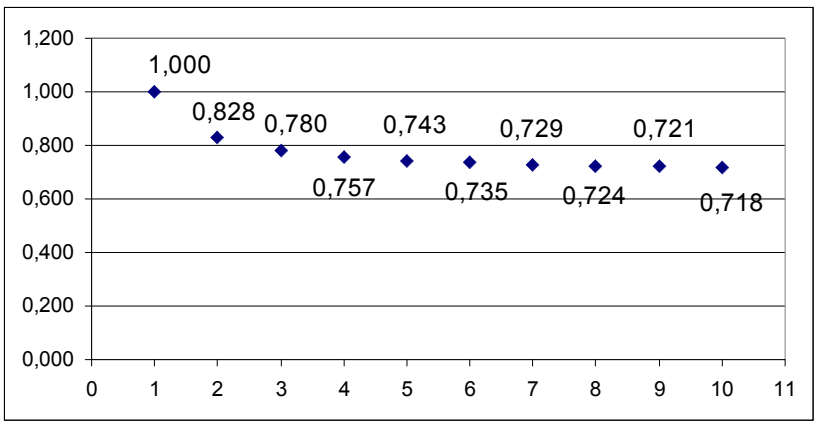
- Definition: Tasks with shorter periods will have higher priorities
- Property: Optimal among fixed priority assignments for periodic independent tasks with Di=Pi
- Feasibility condition

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(2^{\frac{1}{n}} - 1)$$

  - Low complexity O(n)
  - Sufficient condition only (some task sets with load in interval $[U_{lub},1]$ meet their deadline)
  - Also applies to asynchronous task sets

Feasibility analysis for periodic task systems
Sufficient condition for Rate Monotonic (2/2)

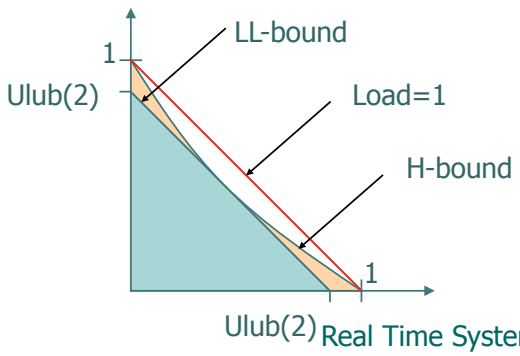
- Variation of Ulub with n (limit = ln(2) =0,69314718)



- Particular case: harmonic periods (for all i,j st Pi<Pj, Pj = k Pi with k integer) : Ulub = 1

Feasibility analysis for periodic task systems
Tighter bound for RM: hyperbolic bound [2003]

- Feasibility condition $$\prod_{i=1}^{n}\left(\frac{Ci}{Pi}+1\right)\leq 2$$
  - Low complexity O(n)
  - Sufficient condition
  - Also applies to asynchronous task sets
- Illustration (2-tasks system)



Feasibility analysis for periodic task systems
Sufficient condition for Deadline Monotonic

- Assumption Di = Pi relaxed (here, Di ≤ Pi)
- Definition: Tasks with shorter relative deadlines will have higher priorities
- Property: Optimal among fixed priority assignments for periodic independent tasks with Di ≤ Pi
- Feasibility condition
$$\sum_{i=1}^{n}\frac{Ci}{Di}\leq n(2^{\frac{1}{n}}-1)$$
  - Low complexity O(n)
  - Sufficient condition only (some task sets with Σ (Ci/Di) > U_{lub},meet their deadline)
  - Also applies to asynchronous task sets

Feasibility analysis for periodic task systems
Response Time Analysis (1/4)

- Response time analysis [Joseph & Pandya, 1986, Audsley & al, 1993]
  - Applicable to any preemptive fixed priority scheduling algorithm, regardless of priority assignment (RM, DM, other)
  - Exact test (necessary and sufficient condition) for synchronous task sets
  - Sufficient condition for asynchronous task sets
- Principle
  - For all every task Ti, iterative calculation of the task response time Ri t its critical instant
  - Verification for every task Ti that Ri ≤ Di

Feasibility analysis for periodic task systems
Response Time Analysis (2/4)

- Introduction to response time computation Ri

  Ri = Ci (worst case execution time of task Ti)
  + Ii (interferences due to preemptions by higher priority tasks)

  Maximum number of preemptions suffered by an instance of Ti : $\sum_{j \in hp(i)} \left\lceil \frac{Ri}{Pj} \right\rceil$

  Corresponding interference (Cj for every preemption by task Tj) $Ii = \sum_{j \in hp(i)} \left\lceil \frac{Ri}{Pj} \right\rceil Cj$

  Response time Ri $Ri = Ci + \sum_{j \in hp(i)} \left\lceil \frac{Ri}{Pj} \right\rceil Cj$
- No simple solution exists (Ri appears on both sides)
  - Solution Ri = smallest value such that the equation is verified
  - No need to check the equation ∀ Ri ≤ Di (ony when the number of preemptions increases)

Feasibility analysis for periodic task systems
Response Time Analysis (3/4)

- Iterative computation of response time Ri

$$w_i^0 = C_i$$
$$w_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^k}{P_j} \right\rceil C_j$$

- When the series converges, the convergence value is the task response time Ri
- The feasibility of task Ti is guaranteed if and only if Ri ≤ Di

Feasibility analysis for periodic task systems
Response Time Analysis (4/4)

- Example

	Ci	Pi	prio
T4	3	10	
T3	1	5	
T2	2	20	
T1	2	20	

$$w_1^0 = C_1 = 2$$
$$w_1^1 = C_1 + \left\lceil \frac{C_1}{P_4} \right\rceil C_4 + \left\lceil \frac{C_1}{P_3} \right\rceil C_3 + \left\lceil \frac{C_1}{P_2} \right\rceil C_2 = 2 + \left\lceil \frac{2}{10} \right\rceil 3 + \left\lceil \frac{2}{5} \right\rceil 1 + \left\lceil \frac{2}{20} \right\rceil 2 = 2+3+1+2 = 8$$
$$w_1^2 = C_1 + \left\lceil \frac{8}{P_4} \right\rceil C_4 + \left\lceil \frac{8}{P_3} \right\rceil C_3 + \left\lceil \frac{8}{P_2} \right\rceil C_2 = 2 + \left\lceil \frac{8}{10} \right\rceil 3 + \left\lceil \frac{8}{5} \right\rceil 1 + \left\lceil \frac{8}{20} \right\rceil 2 = 2+3+2+2 = 9$$
$$w_1^3 = C_1 + \left\lceil \frac{9}{P_4} \right\rceil C_4 + \left\lceil \frac{9}{P_3} \right\rceil C_3 + \left\lceil \frac{9}{P_2} \right\rceil C_2 = 2 + \left\lceil \frac{9}{10} \right\rceil 3 + \left\lceil \frac{9}{5} \right\rceil 1 + \left\lceil \frac{9}{20} \right\rceil 2 = 2+3+2+2 = 9$$

Feasibility analysis for periodic task systems
Exact condition for EDF (Di=Pi)

- Definition [Liu & Layland, 1973]
  - Tasks with earlier absolute deadlines will be executed at higher priorities (deterministic tie-braking rule in case of equality)
  - Preemptive
- Property
  - Optimal among preemptive scheduling algorithms for periodic independent tasks with Di ≤ Pi
- Feasibility condition (valid for Di=Pi)

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$$

  - Low complexity O(n)
  - Necessary and sufficient condition
  - Also valid on asynchronous task sets

Feasibility analysis for periodic task systems
Conditions for EDF (Di≤Pi)

- Feasibility condition (Di ≤ Pi)
  - Sufficient condition $\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$
  - Necessary condition $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$

Feasibility analysis for periodic task systems
Condition for EDF: processor demand (1/6)

- Processor demand approach [Baruah et al, 90]
- Principle
  - Compute the cumulated processor demand on time interval [0,L]
  - Check for all L that the processor demand on interval [0,L] is ≤ L
- Outline
  - Case Di = Pi
    - Concepts: « processor demand », « busy period »
    - Feasibility test
  - Generalisation to case Di ≤ Pi

Feasibility analysis for periodic task systems
Processor demand - Di=Pi (2/6)

- Processor demand
  - Definition: cumulated processor demand on interval [0,L] = total amount of processor time that have to be executed with deadlines less than of equal to L
  - Processor demand in interval [0,L]:

$$C_p(0,L)=\sum_{i=1}^{n}\left\lfloor\frac{L}{P_i}\right\rfloor C_i$$

  - ⌊L/Pi⌋ is the number of arrivals of task Ti in interval [0,L] that must terminate before L (explaining the ⌊⌋, the arrivals in [0,L] those deadline are strictly after L are not counted)
  - Ci is the worst-case execution time of task Ti
- The system is schedulable under EDF if and only if for all L positive [Jeffay & Stone]

$$C_p(0,L)=\sum_{i=1}^{n}\left\lfloor\frac{L}{P_i}\right\rfloor C_i \leq L \qquad (e1)$$

Feasibility analysis for periodic task systems
Processor demand - Di=Pi (3/6)

- Remarks
  - Cyclic scheduling: no need to test (e1) for L > LCM(Pi)
  - Cp(0,L) is a step function: sufficient to test (e1) at task arrival/ deadline dates
- Concept of busy period
  - Smallest interval [0,L] in which the total processing time W(L) requested in [0,L] is completely executed, with W(L):

$$(W(L)=\sum_{i=1}^{n}\left\lceil\frac{L}{P_i}\right\rceil C_i)=L$$

  - ⌈L/Pi⌉ is the number of arrivals of task Ti in interval [0,L]
  - W(L) = processor load demanded in interval [0,L]
  - Busy period Bp= min{ L | W(L) = L}
  - Busy period = period during which the processor is always busy (equal to infinity if the system is overloaded)
  - Result: no need to test equation (e1) for L > Bp

Feasibility analysis for periodic task systems
Processor demand - Di=Pi (4/6)

- Algorithm for computing the busy period
  - Busy period Bp= min{ L | W(L) = L}
  - Algorithm (iterative)

$$W(L)=\sum_{i=1}^{n}\left\lceil\frac{L}{P_i}\right\rceil C_i$$

```
L = Σ Ci; L' = W(L);
H = LCM(P1, …, Pn)
while (L' != L && L' <=H) {
   L = L';
   L' = W(L);
}
if (L' ≤ H) Bp = L; else Bp = ∞;
```

- Note: when the system is not overloaded, end of BP is
  - Either beginning of an idle time
  - Or arrival of a periodic instance

Feasibility analysis for periodic task systems
Processor demand - Di=Pi (5/6)

- Feasibility test : necessary and sufficient condition
  - For all L corresponding to a task arrival in interval min(Bp,hyperperiod)

$$C_p(0,L)=\sum_{i=1}^{n}\left\lfloor\frac{L}{P_i}\right\rfloor C_i \leq L$$

- NB: the test is not interesting for Di=Pi: a less complex exact test already exists (load less than or equal to 1)

Feasibility analysis for periodic task systems
Processor demand - general case Di≤Pi (6/6)

- Processor demand when Di≤Pi

$$C_p(0,L)=\sum_{i=1}^{n}\left(\left\lfloor\frac{L-D_i}{P_i}\right\rfloor+1\right)C_i$$

- Feasibility test (necessary and sufficient)
  - For all L corresponding to a task deadline in interval min(Bp,hyperperiod)

$$\sum_{i=1}^{n}\left(\left\lfloor\frac{L-D_i}{P_i}\right\rfloor+1\right)C_i \leq L$$

  - Busy period duration is computed as when Di=Pi

Feasibility analysis for periodic task systems
Summary

	Di=Pi	Di ≤ Pi
Fixed prio	**Rate Monotonic** Load-based condition (SC) $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(2^{\frac{1}{n}}-1)$	**Deadline Monotonic** $\sum_{i=1}^{n} \frac{Ci}{Di} \leq n(2^{\frac{1}{n}}-1)$ SC Response time analysis (NSC) - **for all fixed prio. assignment** $R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \leq D_i$
Dyn. prio	**EDF** Load-based condition (NSC) $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$	**EDF** Processor demand approach (NSC) $C_p(0,L) = \sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1 \right) C_i$

Accounting for resource sharing
Introduction

- Assumptions
  - Addition of resource sharing to the initial assumptions
- Non preemptive scheduling
  - No problem (tasks are not interrupted during critical sections)
- Preemptive priority based scheduling
  - Identification for every task of the worst-case duration the task is blocked by lower priority tasks (blocking factor) : Bi
    - Use priority inheritance (PIP,PCP,SRP) required
  - Modification of feasibility tests to account for blocking factor Bi

Accounting for resource sharing
Fixed-priority scheduling: RM

- Original feasibility test (SC) (Di=Pi) $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(2^{\frac{1}{n}}-1)$
- Modified feasibility test (SC) $\forall i \in [1,n], \left(\sum_{k=1}^{i} \frac{C_k}{P_k}\right) + \frac{B_i}{P_i} \leq i(2^{\frac{1}{i}} - 1)$
  - Term 1: effect of preemptions by higher priority tasks
  - Term 2: blocking by lower priority tasks
- Simpler, but more pessimistic condition
  - We observe that $\frac{B_i}{P_i} \leq \max_k(\frac{B_k}{P_k})$ and $n(2^{\frac{1}{n}} - 1) \leq i(2^{\frac{1}{i}} - 1)$
  - Resulting condition $\sum_{i=1}^{n} \frac{C_i}{P_i} + \max(\frac{B_1}{P_1}, \cdots, \frac{B_n}{P_n}) \leq n(2^{\frac{1}{n}}-1)$

Accounting for resource sharing
Fixed-priority scheduling: DM

- Original condition (SC) (Di ≤ Pi)

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq n(2^{\frac{1}{n}}-1)$$

- Modified condition (SC)

$$\forall i \in [1,n], \left(\sum_{k=1}^{i} \frac{C_k}{D_k}\right) + \frac{B_i}{D_i} \leq i(2^{\frac{1}{i}} - 1)$$

- Simpler, but less precise condition

$$\sum_{i=1}^{n} \frac{C_i}{D_i} + \max(\frac{B_1}{D_1}, \cdots, \frac{B_n}{D_n}) \leq n(2^{\frac{1}{n}}-1)$$

Accounting for resource sharing
Fixed-priority scheduling: extension of RTA

- Original condition (NSC)

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \leq D_i$$

- Modified condition

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \leq D_i$$

  - The modified condition is sufficient only (Bi overestimates the task blocking time)

Accounting for resource sharing
Dynamic priority scheduling: EDF

- Original condition (NSC) (Di=Pi)

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$$

- Modified condition (Di=Pi) - sufficient condition

$$\forall i \in [1,n] \left(\sum_{k=1}^{i} \frac{C_k}{P_k}\right) + \frac{B_i}{P_i} \leq 1$$

Accounting for resource sharing
Dynamic priority scheduling: EDF

- Original condition (SC) (Di ≤ Pi)

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$$

- Modified condition (SC) (Di ≤ Pi)

$$\forall i \in [1,n] \left(\sum_{k=1}^{i} \frac{C_k}{D_k}\right) + \frac{B_i}{D_i} \leq 1$$

Accounting for resource sharing
Dynamic priority scheduling: extension of processor demand

- Original condition (NSC)

$$\forall L, C_p(0,L) = \sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1 \right) C_i \leq L$$

- Modified condition

$$\forall i, \forall L, C_p(0,L) = \sum_{k=1}^{i} \left(\left\lfloor \frac{L-D_k}{P_k} \right\rfloor + 1 \right) C_k + \left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1 \right) B_i \leq L$$

  - Sufficient condition only

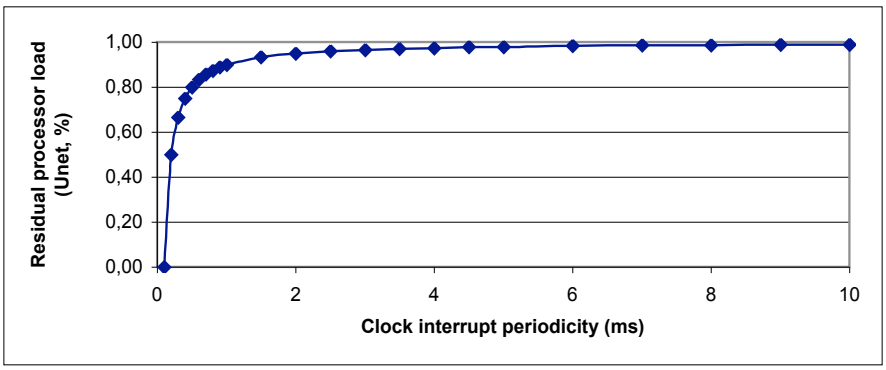
Accounting for system overheads
Introduction

- Origin of overheads
  - Execution of interrupt handlers (clock, sensors, device controllers)
  - Systems calls
  - Context switches
- Ignored so far

- System calls
  - Synchronous: can be integrated in tasks WCETs (Ci)
  - How to evaluate the WCET of system calls?
    - Measurements: be careful of hardware effects (e.g. caches)
    - Static analysis: see chapter on WCET estimation

Accounting for system overheads
Periodic interrupts

- Example: clock interrupt
  - Periodic: can be modelled as an additional periodic task with high priority
  - Decreases the residual available processor capacity (Unet). Example of an ISR duration of 100µs



Accounting for system overheads
Context switches

- Estimation of the number of context switches suffered by a task (Ni)
  - Depends on the scheduling algorithm
- Estimation of context switch time δ (saving + restoration of context)
  - Direct cost: processor register manipulations
  - Indirect cost : due to hardware effects (cache/TLB/pipeline)
- Integration in feasibility tests
- Example: RM scheduling, periodic, synchronous independent tasks
  - Upper bound of number of context switches (much tighter bound can be found)

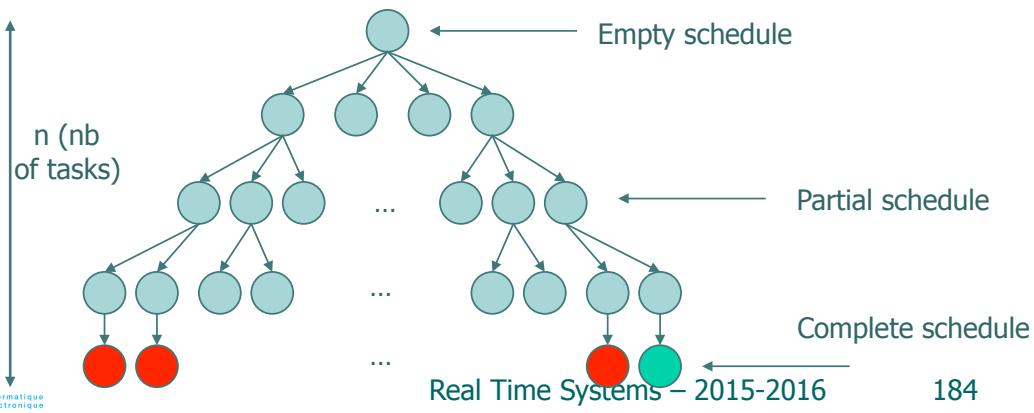
$$N_i = \sum_{k \in hp(i)} \left\lceil \frac{P_i}{P_k} \right\rceil$$

  - Modified feasibility condition (SC)

$$\sum_{i=1}^{n} \frac{C_i + N_i \delta}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

Feasibility for aperiodic tasks and non preemptive scheduling and known arrival dates

- Problem complexity
  - The problem of finding a feasible non preemptive schedule for aperiodic tasks with arbitrary (but known) arrival times is NP-hard
- The problem can be modelled as a tree search problem



Feasibility for aperiodic tasks and non preemptive scheduling

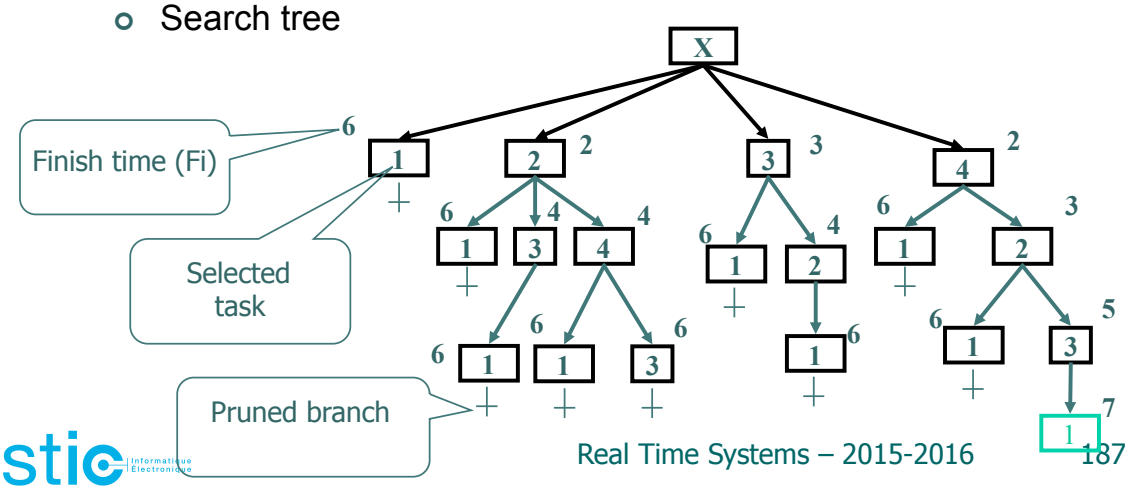
- Solutions
  - Exhaustive of the search tree O(n n!): too complex in general
  - Search space reduction
    - Pruning
      - Example: Bratley's algorithm
    - Use of heuristics to follow promising paths
- Problem complexity → oriented towards off-line verification

Feasibility for aperiodic task systems and non preemptive scheduling
Bratley's algorithm (1971) (1/3)

- Task model
  - Aperiodic tasks with known arrival times and deadlines
- Output
  - Feasible schedule (if any) without preemption
- Principle
  - Each node is associated a partial schedule
  - Depth first exploration of the search tree
  - At every node, attempt to schedule all not yet scheduled task
  - Pruning in two cases
    - A feasible schedule has been found
    - The addition of any task to the current path causes a deadline miss

Feasibility for aperiodic task systems and non preemptive scheduling
Bratley's algorithm (1971) (2/3)

- Example
  - A1 = 4 A2 = 1 A3 = 1 A4 = 0
  - C1 = 2 C2 = 1 C3 = 2 C4 = 2
  - D1 = 7 D2 = 5 D3 = 6 D4 = 4
- Search tree



Feasibility for aperiodic task systems and non preemptive scheduling
Bratley's algorithm (1971) (3/3)

- Properties
  - Pruning technique very effective for reducing the search space
  - Worst-case complexity is still O(n n!)
  - Can only be used in off-line mode

Feasibility for aperiodic task systems, preemptive scheduling, precedence constraints
Chetto-Silly, 1990

- Polynomial time complexity
- Principles
  - Input : task set with arrival times, deadlines, precedence constraints
  - Modification of task timing parameters to make tasks independent
    - Arrival times
      - For all nodes in precedence graph $A_i^*=A_i$
      - While non stable, for all direct precedences $J_k \rightarrow J_i$, $A_i^* = \max(A_i^*, A_k^*+C_k)$
    - Deadline
      - For all nodes in precedence graph $D_i^*=D_i$
      - While non stable, for all direct precedence $J_i \rightarrow J_k$, $D_i^* = \min(D_k^* - C_k, D_i^*)$
    - Schedule tasks using preemptive EDF : if feasible, OK

Schedulability analysis
Summary

- Plenty of feasibility tests for different assumptions and scheduling policies
  - Still a very active research area
  - This lecture has presented the most simple results
- Beware of the application conditions of feasibility tests
  - Task model
  - Scheduling policies
  - A feasibility test is only a formula

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}}-1)$$

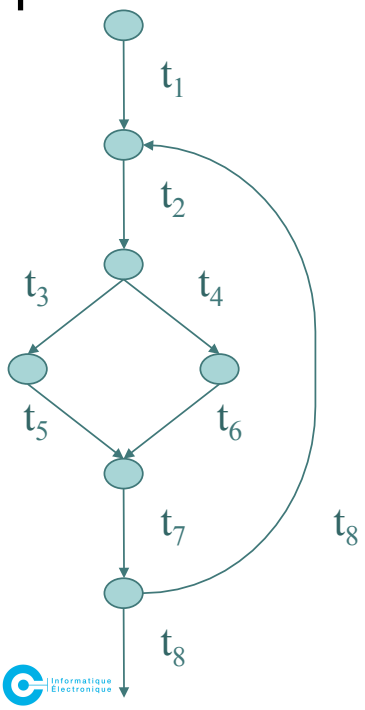
- Testing is not sufficient (identification of worst-case execution scenarios)
- Schedulability analysis tools:
  - Commercial: RapidRMA, Sympta/S, software by RTaW
  - Academic: Mast, Cheddar

Worst-case execution time (WCET) estimation

Outline

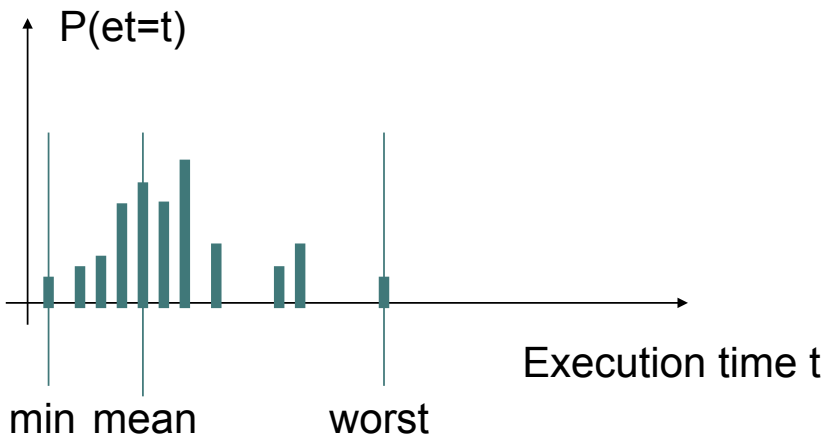
- Worst-case execution times: definitions
- Static timing estimation methods
  - Flow analysis
  - Computation
  - Hardware-level analysis
- Measurement-based and hybrid methods
- Predictable programming and compilers

Execution time
Influencing elements



- Sequencing of actions (execution paths)
  - Input data dependent
  - A priori unknown input data
- Duration of every action on a given processor
  - Hardware dependent
  - Depends on past execution history

Execution time
Distribution



Execution times

Application needs

- Desktop applications:
  - As-fast, and as stable as possible
  - Typical input data (average-case)
  - Performance evaluation: execution/simulation with typical input data
- Real-time applications:
  - Before a deadline (real-time is not real-fast)
  - For all input data (including the worst-case)

Validation of real-time systems

What is needed to demonstrate that deadlines are met?

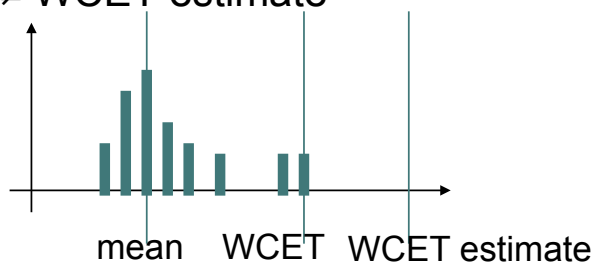
- System-level validation
  - Worst-case knowledge of system load (task arrivals, interrupt arrivals)
  - Ex: synchronous arrival of periodic tasks
- Task-level validation
  - Worst-case execution time of individual tasks
  - WCET (Worst-Case Execution Time)
    - Upper bound for executing an isolated piece of code
    - Code considered in isolation
    - WCET ≠ response time

Different uses of WCET

- Temporal validation
  - Schedulability analysis
  - Schedulability guarantees (worst-case)
- System dimensioning
  - Hardware selection
- Optimization of application code
  - Early in application design lifecycle

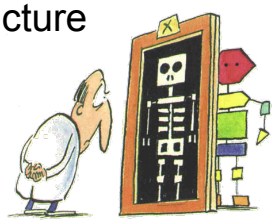
WCET: Definition

- Challenges in WCET estimation
  - Safety (WCET > any possible execution time) : confidence in schedulability analysis methods
  - Tightness
    - Overestimation ⇒ schedulability test may fail, or too much resources might be used
- WCET ≠ WCET estimate

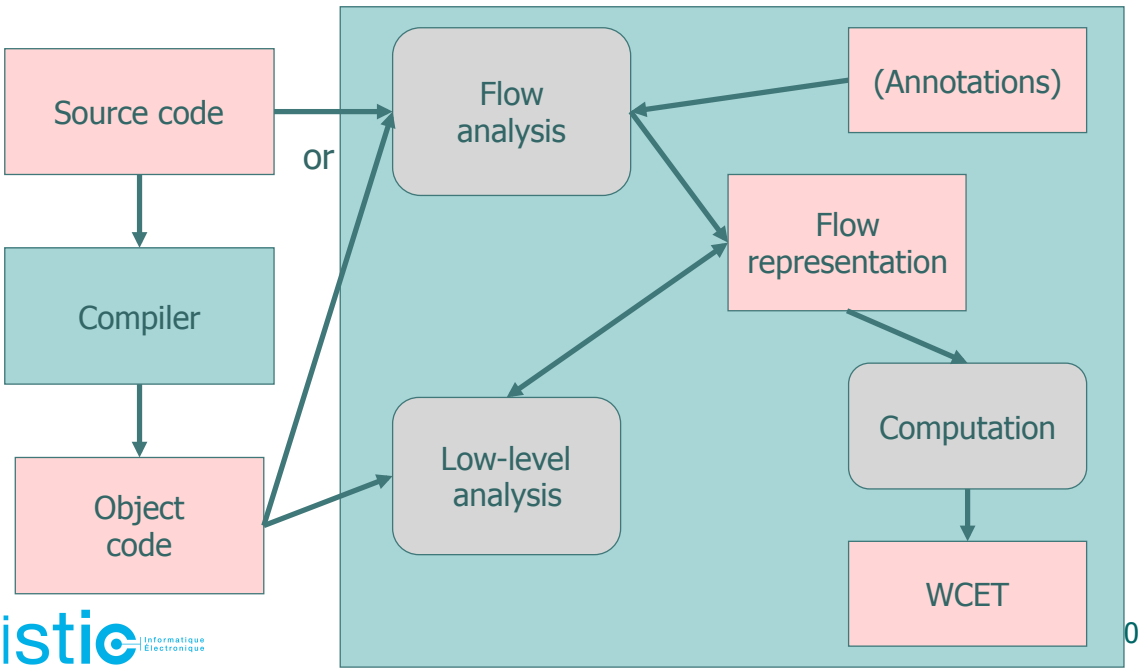


Static WCET estimation methods

- Principle
  - Analysis of program structure (no execution)
  - Computation of WCET from program structure
- Components
  - Flow analysis
    - Determines possible flows in program
  - Low-level (hardware-level) analysis
    - Determines the execution time of a sequence of instructions (basic block) on a given hardware
  - Computation
    - Computation from results of other components
    - All paths need to be considered : safety



Overview of components

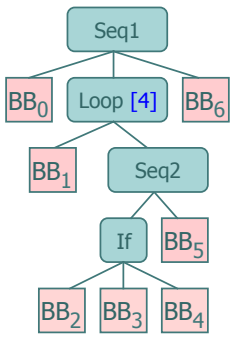


WCET computation

Assumptions

- Simple architecture
  - Execution time of an instruction only depends on instruction type and operand
  - No overlap between instructions, no memory hierarchy

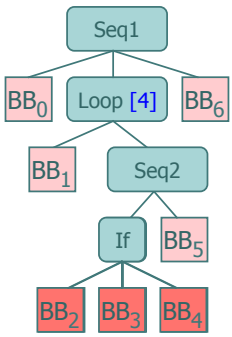
Tree-based computation



- Data structures
  - Syntax tree (control structures)
  - Basic block
- Principle
  - Determination of execution time of basic block (low-level analysis)
  - Computation based on a bottom-up traversal of the syntax tree (timing schema)

Tree-based computation

WCET(SEQ)	S1;…;Sn	WCET(S1) + … + WCET(Sn)
WCET(IF)	if(test) then else	WCET(test) + max(WCET(then) , WCET(else))
WCET(LOOP)	for(;tst;inc) {body}	maxiter * (WCET(tst)+WCET(body+inc)) + WCET(tst)



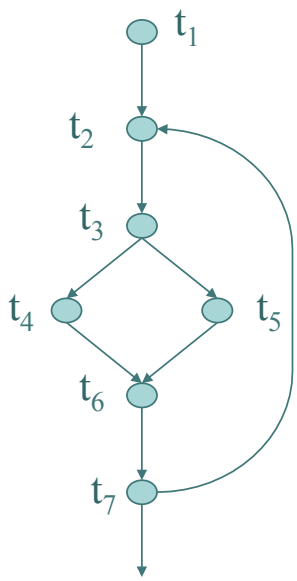
Timing schema

WCET(Seq1) = WCET_BB0 + **WCET(Loop)** +WCET_BB_6
WCET(Loop) = 4 * (WCET_BB1 + **WCET(Seq2)**) + WCET_BB1
WCET(Seq2) = **WCET(If)** + WCET_BB5
WCET(If) = WCET_BB2 + **max**(WCET_BB3 , WCET_BB4)

Tree-based computation

- Pros
  - Low computational effort
  - Good scalability with respect to program size
  - Good user feedback (source-code level)
- Cons
  - Not compatible with aggressive compiler optimizations
  - Expression of complex flow facts difficult (inter-control-structure flow facts)

IPET (Implicit Path Enumeration Technique)



- Integer linear programming
  - Objective function: max: $f_1t_1+f_2t_2+\ldots+f_nt_n$
  - Structural constraints

    $\forall$ *basic block* n_i: $f_i = \sum_{p \in In(ni)} f_{p\text{-}ni} = \sum_{s \in Out(ni)} f_{ni\text{-}s}$

    $f_1 = 1$
  - Extra flow information

    $\forall$ basic block n_i *in loop,* $f_i \leq k$ (loop bound)

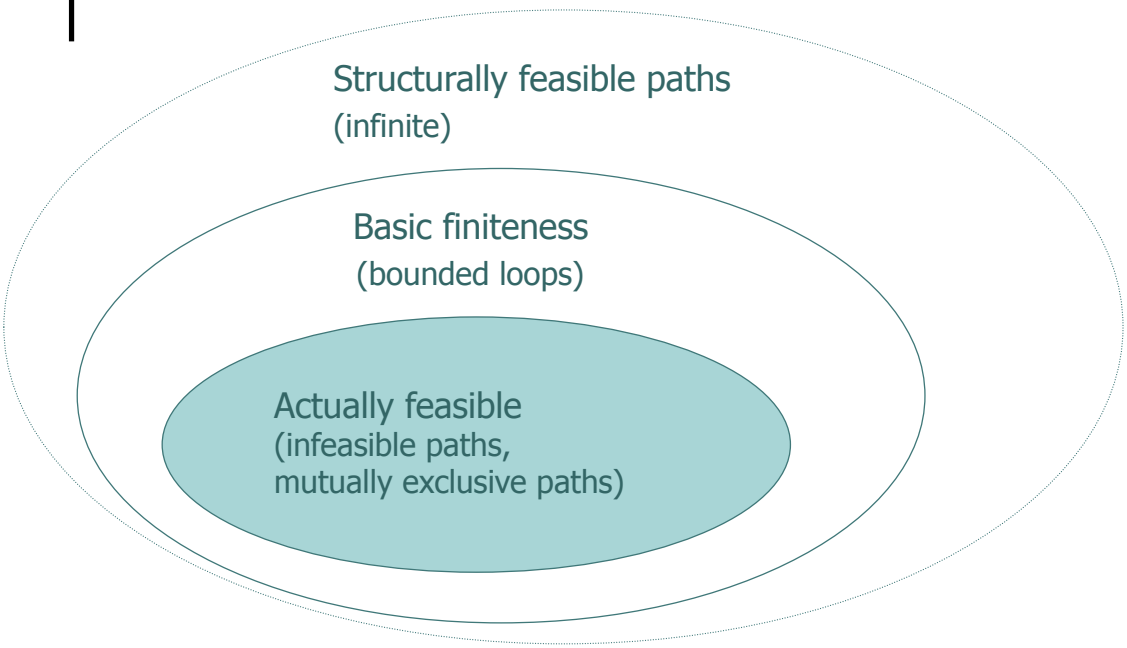
    $f_i + f_j \leq 1$ (mutually exclusive paths - not in loop)

    $f_i \leq 2\, f_j$ (relations between execution freqs.)

IPET

- Pros
  - Supports all unstructured flows (gotos, etc.)
  - Supports all compiler optimizations, including the most aggressive ones
- Cons
  - More time-consuming than tree-based methods
  - Low-level user feedbacks (works at binary level)
  - Annotations are hard to provide (need to know compiler optimizations)
- Mostly used in commercial/academic tools

Flow analysis



Flow analysis

- Infeasible paths

```
int baz (int x) {
   if (x<5)          // A
      x = x+1;       // B
   else x=x*2;       // C
   if (x>10)         // D
      x = sqrt(x);   // E
   return x;         // F
}
```

- Path ABDEF is infeasible
- Identification of infeasible paths: improves tightness

Flow analysis

- Maximum number of iterations of loops

```
for i := 1 to N do          ← Loop bound: N
   for j := 1 to i do       ← Loop bound: N
   begin
      if c1 then A.long
            else B.short
      if c2 then C.short
            else D.long
   end
```

$\frac{(N+1)N}{2}$ executions

- Tight estimation of loop bounds: improves tightness

Flow analysis

- Determination of flow facts
  - Automatic (static analysis): not decidable in general (equivalent to halting problem)
  - Manual: annotations
    - Loop bounds: constants, or symbolic annotations
      - Minimum required path information
    - Annotations for infeasible / mutually exclusive paths, relations between execution counts

Flow analysis

Flow analysis using abstract interpretation [Gus06]

- Some numbers

Pgm	Time	WCETorig	Time	WCETff	#FF
Crc	4.9	834159	6.65	833730	56
Inssort	0.16	31163	0.17	18167	7
Ns	6.09	130733	6.81	130733	8
Nsichneu	36.88	119707	435.70	41303	65280

Low-level analysis

Introduction

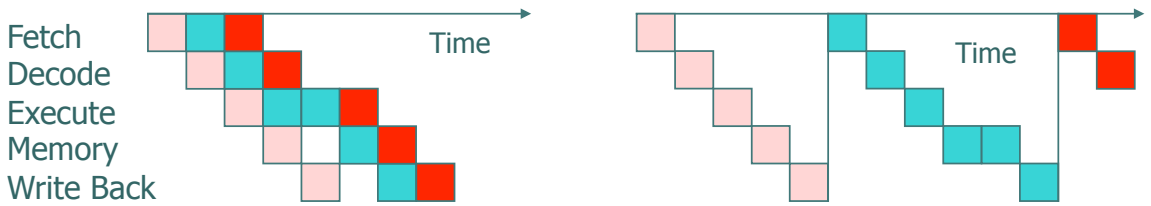
- Simple architecture
  - Execution time of an instruction only depends on instruction type and operand
  - No overlap between instructions, no memory hierarchy
- Complex architecture
  - Local timing effects
    - Overlap between instructions (pipelines)
  - Global timing effects
    - Caches (data, instructions), branch predictors
    - Requires a knowledge of the entire code

Low-level analysis
Introduction

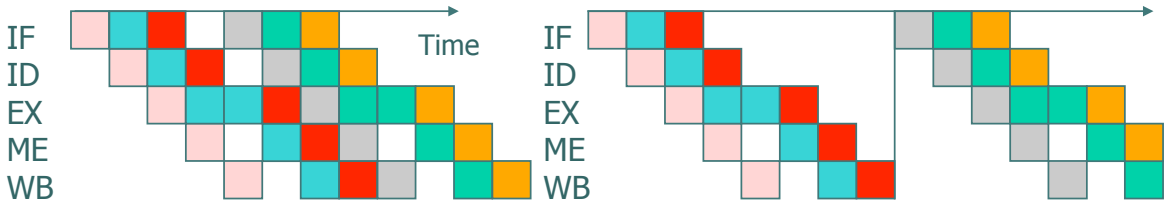
- Outline
  - Pipelines, caches, branch predictors,… considered in isolation
  - Interferences between analyses
- Restrictions
  - Simple in-order pipelines first

Low-level analysis
Pipelining

- Principle : parallelism between instructions
  - Intra basic-block

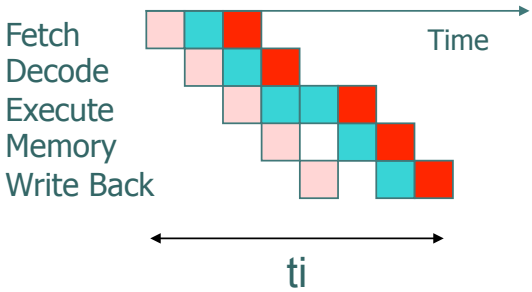


  - Inter basic-block



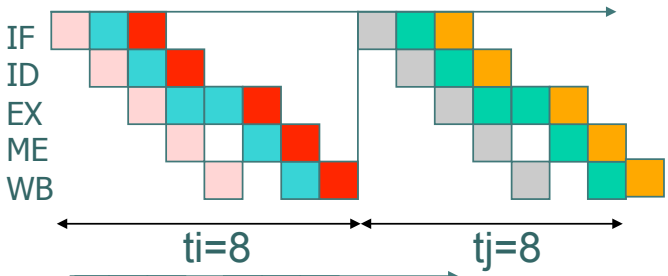
Low-level analysis
Pipelining

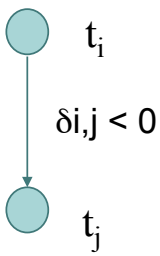
- Intra basic block: "Simulation" of the flow on instructions into the pipeline stages
  - Takes in consideration data hazards (RAR, RAW, WAW)
  - Bypass mechanism (if any) has to be modeled
  - Obtained by WCET analysis tool or external tool (simulator, processor)

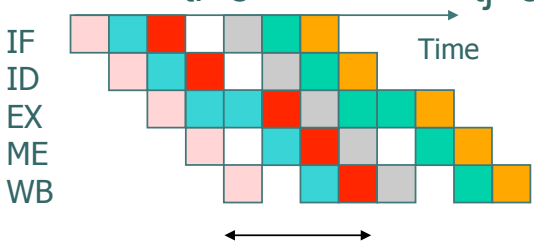


Low-level analysis
Pipelining

- Inter basic-block : modification of computation step
  - Example: IPET, extra constraints in ILP problem (negative costs on edges)





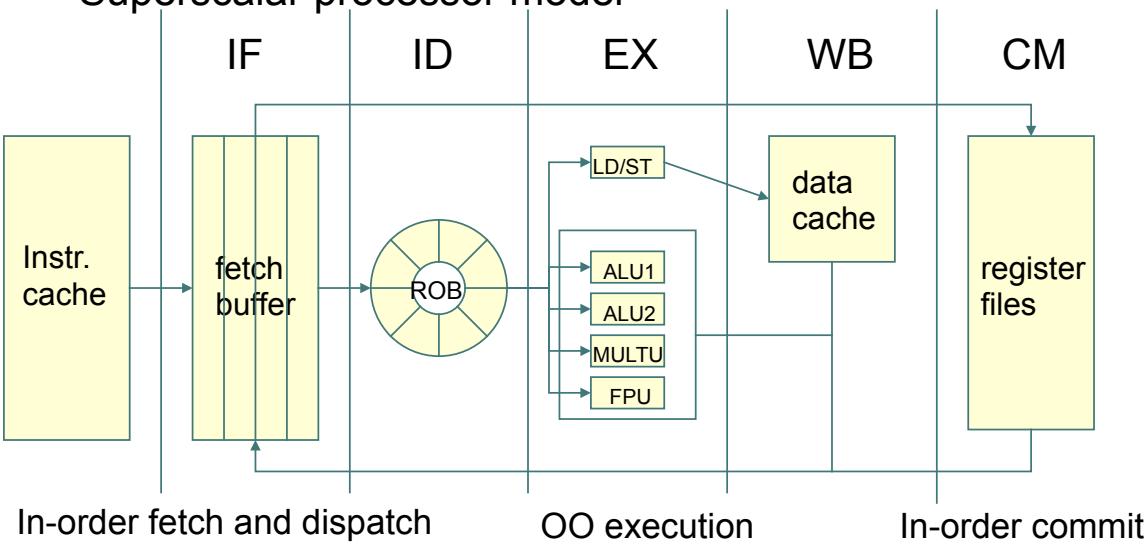


max: $\Sigma f_i t_i + \Sigma a_{i,j} \delta_{i,j}$

Low-level analysis

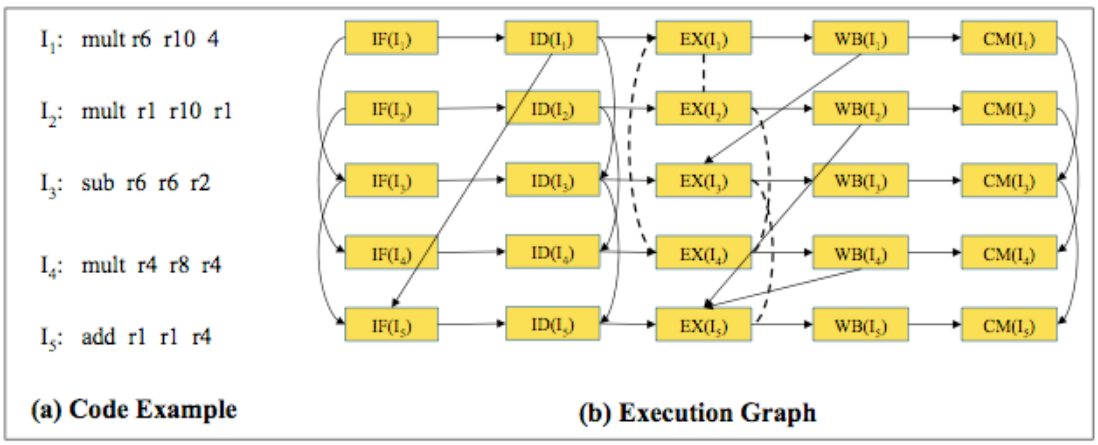
Out-of-order execution [Li&Mitra, 06]

Superscalar processor model



Low-level analysis

Out-of-order execution: pipeline modelling



- Plain arrows: Pipeline stages of same instructions, buffer size, data dependencies among instructions, in-order fetch and ID
- Dashed arrows: contentions (shared resources)

Low-level analysis
Out-of-order execution: timing analysis

- Assumptions (simplified version)
  - Fixed-latency functional units
- Principle
  - Applied at every basic block
  - Scheduling problem: find the longest path in an acyclic graph with precedence and exclusion constraints
  - Algorithm (simplified)
    - Traverse the graph in topological order (finish time of all predecessors known)
    - Conservative estimation of contention time

Low-level analysis
Instruction caches

- Cache
  - Takes benefit of temporal and spatial locality of references
  - Speculative: future behaviour depends on past behaviour
  - Good average-case performance, but predictability issues
- How to obtain safe and tight estimates?
  - Simple solution (all miss): overly pessimistic
  - Objective: predict if an instruction will (certainly) cause a hit or might (conservatively) cause miss.

Low-level analysis
Instruction caches

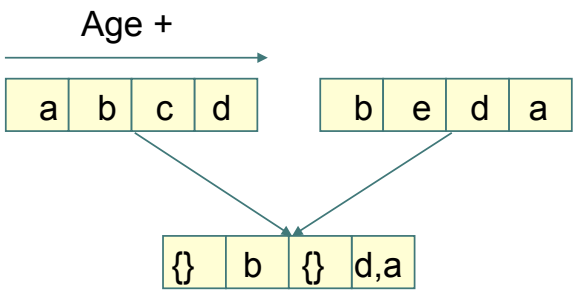
- Based on abstract interpretation
- Computation of Abstract Cache States (ACS)
  - Contains all possible cache contents considering "all" possible execution paths
  - Fixpoint computation
- Instruction categorization from ACS
  - *Always hit (AH)* - guaranteed to be a hit
  - *Always miss (AM)* - guaranteed to be a miss
  - *First miss (FM)* - guaranteed to be a hit after first referenced
  - *Not-classified (NC)* - don't know

Low-level analysis
Instruction caches

- Analyses
  - Must: ACS contain all program lines guaranteed to be in the cache at a given point
  - May: ACS contain all program lines that may be in the cache at a given point
  - Persistence: ACS contains all program lines not evicted after loaded
- Modification of ACS
  - Update: at every reference
  - Join: at every path merging point

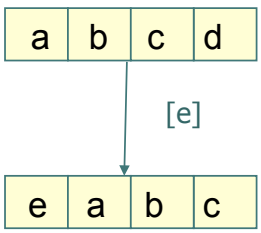
Low-level analysis

Instruction caches - Must analysis



Join

Intersection
+ max age

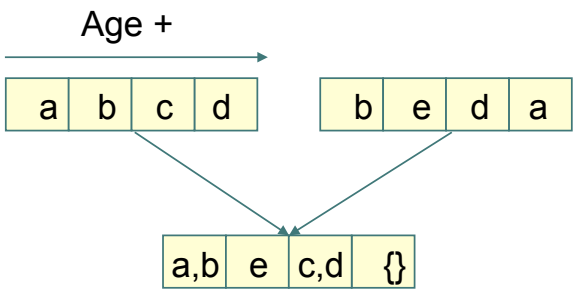


Update

Apply replacement policy

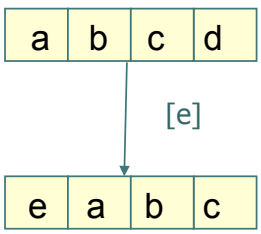
Low-level analysis

Instruction caches - May analysis



Join

Union
+ min age



Update

Apply replacement policy

Low-level analysis
Instruction caches

- From ACS to classification
  - If in ACS_must: Always Hit
  - Else, if not in ACS_May: Always Miss
  - Else, if ACS_Persistence: First Miss
  - Else Not Classified
- From classification to WCET (IPET)
  - For every BB: t_first, t_next
  - Objective function: max Σ (f_first * it_firsti + f_next * it_nexti)
  - New constraints:
    - fi = f_firsti + f_nexti
    - Constraints based on cache classification

Low-level analysis
Data caches

- Extra issue: determination of addresses of data
- Means: abstract interpretation / DF analysis
  - Value analysis
  - Pointer analysis
- Results:
  - Superset of referenced addresses per instruction
- Example:
  - for (int i=0;i<N;i++) tab[i+j/z] = x;
  - Any address inside tab may be referenced per loop iteration (but only one)
  - Hard-to-predict reference (input-dependent)

Low-level analysis
Data caches

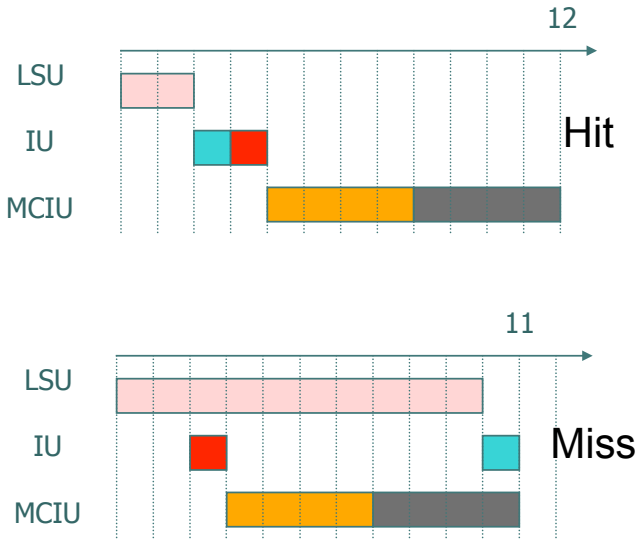
- Solutions (with some limitations)
  - Compiler-controlled methods: don't cache input-dependent data structures
  - Assume every potentially referenced address is actually referenced (2*N misses in example)
  - Cache Miss Equations (CME): for affine data-independent loop indices

Low-level analysis
Branch prediction

- Branch predictors
  - Local predictors [Colin, 00]
  - Global predictors [Mitra], [Rochange]
  - Most complex predictors out of reach !

The bad news …
Timing anomalies (out-of-order execution)

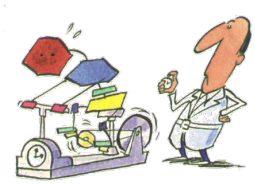
	Disp. Cycle	Instruction
A	1	LD r4,0(r3)
B	2	ADD r5, r4, r4
C	3	ADD r11, r10, r10
D	4	MUL r11, r11, r11
E	5	MUL r13, r12, r12



Exploration is required

WCET estimation methods
Dynamic methods

- Principle
  - Input data
  - Execution (hardware, simulator)
  - Timing measurement
- Generation of input data
  - User-defined: reserved to experts
  - Exhaustive
    - Risk of combinatorial explosion

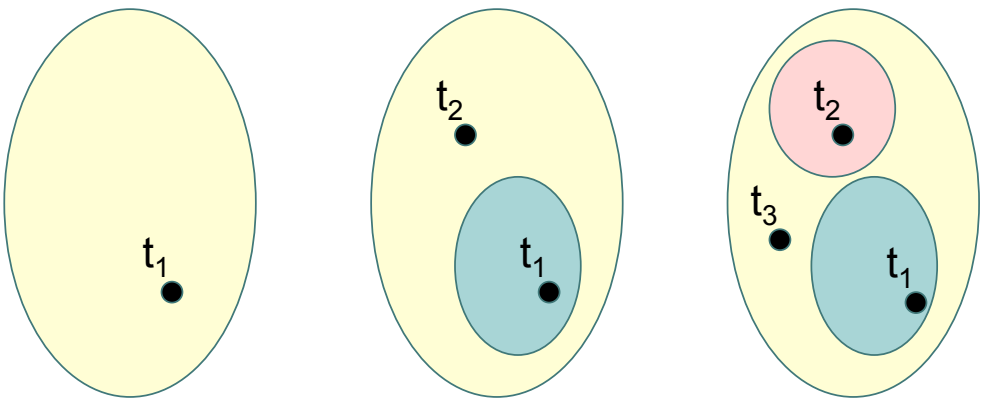


Safe dynamic method

- Principles
  - Structural testing
  - “All-paths” coverage criterion
  - Avoid enumerating all input-data combination
- Assumptions
  - A1. One feasible path in source code gives rise to one path in binary code
  - A2. Execution time of a feasible path is the same for all input values which cause the execution of the path
  - A3. Existence of a worst-case machine state

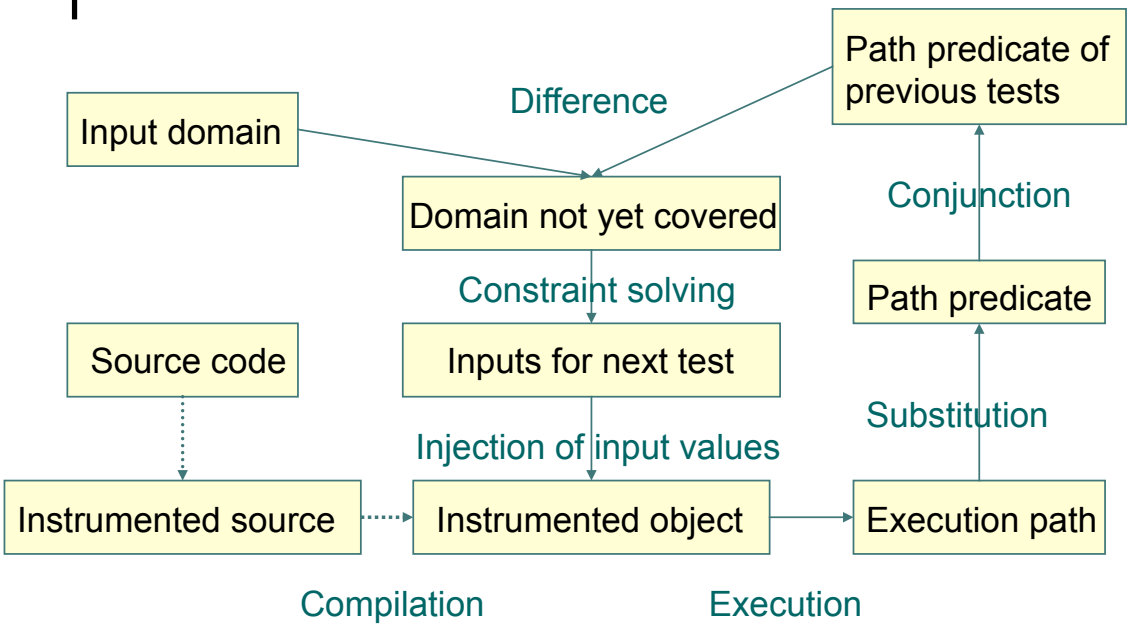
Safe dynamic method

- Iterative coverage of input domain



WCET estimation methods

Safe dynamic method



WCET estimation methods

Safe dynamic method

- Advantages
  - No hardware model
  - All paths are covered
- Drawbacks
  - Assumption A2 restricts the architecture to be used
  - Still an explosion of the number of paths to be explored for some programs
- Ongoing work: measuring fewer execution paths

WCET estimation methods

Another safe hybrid method: symbolic simulation

- Based on a cycle-accurate simulator
- Values and registers tagged with a “type” (known/unknown)
- Extension of the semantics of instructions
  - Memory transfer from an unknown register content: target address tagged as “unknown”
  - Conditional branch on unknown condition: exploration of both branches
  - For every instruction …
- State merging to avoid exploring all paths
  - State = (memory/registers)
  - Merging = pessimistic union of the two states
  - Merging points: different granularities
- Benefits: flow-analysis “for-free”
- Drawbacks: hardware models, terribly slow

WCET estimation methods

Less safe dynamic methods

- Use of genetic algorithms
- Data structures
  - Population, individuals, genes (sets of input data)
- Operators
  - Cross-over = mix two genes (sets of input data)
  - Mutation = change one gene (input data)
  - Selection = individuals with largest execution time (obtained through measurements)
- Benefits
  - Measurement-based: no need for hardware models
  - No safety guarantee
  - Used to validate static analysis methods, or when safety is not mandatory

WCET estimation methods

Hybrid method

- High-level analysis and WCET combination
  - Static analysis
- Execution times of basic blocks
  - Measurements
  - Distributions of execution time
- WCET computation (tree-based)
  - Operators defined on distributions instead of single values
    - Independent basic blocks: convolution
    - Non-independent basic blocks: biased convolution
  - Result: probabilistic WCET
- Evaluation
  - Benefits: no need for hardware model, path analysis is not probabilistic
  - Safety is not guaranteed regarding hardware effects

Open issues (low-level analysis)

- Complexity of static WCET estimation tools
- Timing anomalies, integration of sub-analyses
- Analysis tools may be released a long time after the hardware is available
- Trends:
  - Probabilistic low-level analysis
  - Software control of hardware (cache locking, partitioning, compiler-directed branch prediction)
  - Multi-core architectures

Static WCET estimation methods
A method for every usage

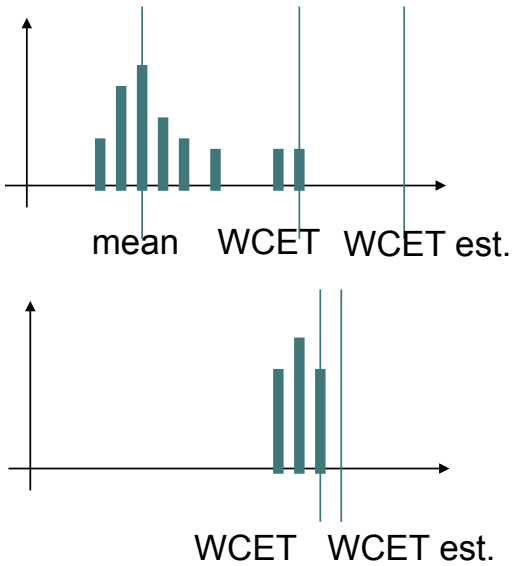
- Static WCET estimation
  - Safety ☺
  - Pessimism ☹
  - Need for a hardware model ☹
  - Trade-off between estimation time and tightness (tree-based / IPET) ☺
- Measurement-based methods
  - Safety ? Probabilistic methods
  - Pessimism ☺
  - No need for hardware models ☺

WCET estimation tools

- Academic
  - Cinderella [Princeton]
  - Heptane [IRISA]
  - Otawa [IRIT Toulouse]
  - SWEET [Sweden]
- Industrial
  - Bound-T [SSF]
  - AIT [AbsInt, Sarbrüken]
  - Rapitime [Rapita systems, York]

Predictable programming goals

- Traditional programming
  - Good average-case performance
- Hard real-time programming
  - WCET must be computable
  - WCET estimate must be as low as possible
  - Low variability of possible execution times



Traditional programming

- Goal : good average-case performance
- Strategy
  - Test input data to favor frequent cases
- Negative effect of WCET and WCET estimate
  - Costs for identifying frequent situations (inputs)
  - Branching costs
  - Complexity is unevenly distributed over input data sets
  - Savings of frequent situations paid back in rare situations

Single-path paradigm

- Produce code free from input-data dependent control decisions
- Use predication (data-dependencies) instead of control dependencies
- Hardware support: predicated instructions

Single-path paradigm (trad. prog.)

```
void bubble (int a[]) {
   for (i=N-1;i>0;i--) {
         for (j=1,j<=i;j++) {
                  if (a[j-1]>a[j]) { /* Swap */
                           t = a[j]; a[j]=a[j-1]; a[j-1]=t;
                  }
         }
   }
}
```

Single-path paradigm (single-path. prog.)

```
void bubble (int a[]) {
    for (i=N-1;i>0;i--) {
        for (j=1,j<=i;j++) {
            s = a[j-1];
            t = a[j];
            s<=t: a[j-1] = s;
            s<=t: a[j] = t;
            s>t: a[j]=t;
            s>t: a[j]=s;
        }
    }
}
```


Single-path paradigm

	Nb Paths	Min E.T.	WCET
Traditional	3628800	675	810
Single-path	1	972	972

- Benefit
  - Flow estimation simplified
- Drawback
  - Programmer or compiler-directed: modification of programming habits / tool chain
  - WCET higher than with traditional programming

Hybrid scheduling of hard real-time and soft real-time tasks

Hybrid scheduling
Introduction (1/2)

- Previous chapters
  - Periodic arrivals or aperiodic arrivals with known arrival times
  - Hard real time tasks
- Typical characteristics of real-time applications: hybrid
  - Control tasks: hard real time, mostly periodic
  - Other tasks : event-triggered, unknown arrival times
  - Second class of tasks not supported by feasibility conditions presented so far. No off-line guarantees can be provided for this class of tasks
- Firm real-time tasks: tasks requiring on-line guarantees on individual instances

Hybrid scheduling
Introduction (2/2)

- Chapter contents: scheduling of hybrid task sets
  - Hard real time periodic tasks
  - Firm or soft real time aperiodic tasks
- Assumptions and notations
  - Periodic tasks T_i, synchronous, $D_i = P_i$, WCET C_i
  - Fixed priority scheduling for periodic tasks (RM)
  - Arrival times of aperiodic tasks J_a a priori unknown
- Example: software for mobile phones
  - Network management tasks: periodic, hard real time
  - Application tasks: soft real time

Hybrid scheduling
Outline

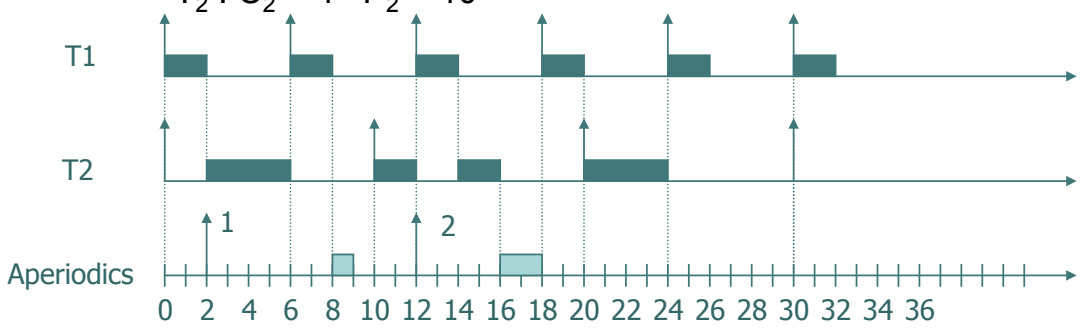
- Background scheduling
- Server based approaches
  - Polling server
  - Deferrable server
  - Priority exchange server

Background scheduling (1/3)

- Principle
  - Execution of aperiodic tasks in the background (i.e. at a lower priority than periodic tasks)
- Example

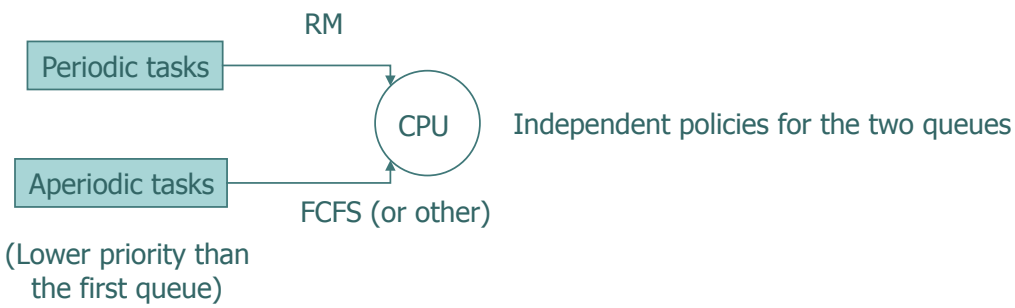
  $T_1 : C_1 = 2 \quad P_1 = 6$

  $T_2 : C_2 = 4 \quad P_2 = 10$



Background scheduling (2/3)

- Remark: by construction, no impact on periodic tasks
- Typical implementation



Background scheduling (3/3)

- Pros
  - Straightforward implementation
  - No impact on periodic tasks (feasibility tests need not be modified)
- Cons
  - Aperiodic tasks only use the spare time left by periodic tasks
    - If the processor load is high, the response time of aperiodic tasks can be important
  - Usable only at moderate processor load, for non urgent aperiodic tasks

Server based hybrid scheduling for fixed priority scheduling

Polling server (1/8)

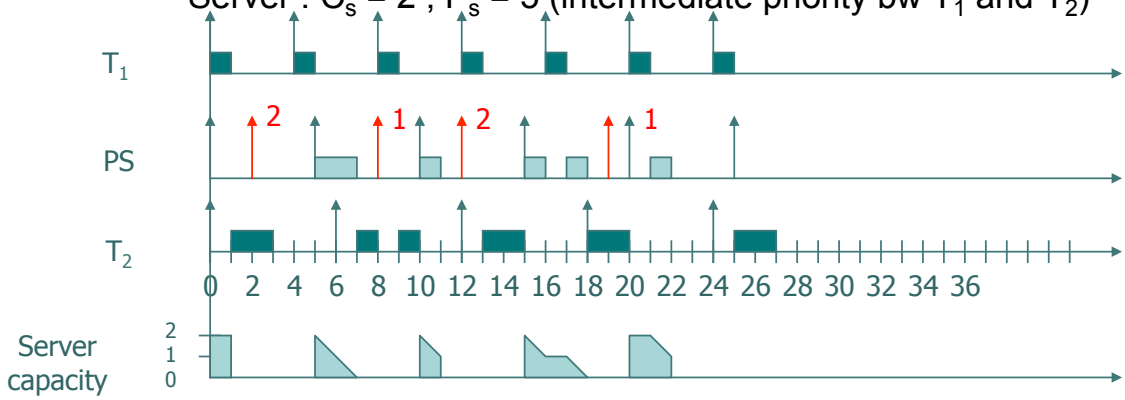
- Principle
  - Use of a server: additional periodic task dedicated to the execution of aperiodic tasks (common principle to all server based approaches)
    - Server characteristics (C_s,P_s), Cs is the server capacity
    - Server uses the same scheduling policy as periodic tasks
    - Aperiodic tasks executed during at most C_s time units per server period P_s
  - Server code (polling server - PS - only)
    - Serve any pending aperiodic requests within the limit of the capacity Cs
    - If no aperiodic request is pending, suspension until the beginning of next period
    - Capacity replenishment every Ps

Polling server (2/8)

- Example (RM priority assignment)

  $T_1 : C_1 = 1 \quad P_1 = 4$ $\qquad$ $T_2 : C_2 = 2 \quad P_2 = 6$

  Server : $C_s = 2$; $P_s = 5$ (intermediate priority bw T_1 and T_2)



Times 1, 11, 22 : server suspension (no pending aperiodic task)
Time 7 : server suspension (server capacity exhausted)
Time 16 : server preemption by T1: server capacity preserved

Polling server (3/8)

- Feasibility test for periodic tasks
  - Original condition modified (only) to consider the additional periodic tasks (server)
  - Sufficient feasibility test (RM)

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \frac{C_s}{P_s} \le (n+1)(2^{\frac{1}{n+1}} - 1)$$

  - Extension to m polling servers of different priorities, to serve aperiodic tasks with different importance

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{j=1}^{m} \frac{C_j}{P_j} \le (n+m)(2^{\frac{1}{n+m}} - 1)$$

Polling server (4/8)

- Acceptance test for an aperiodic task (firm deadline)
  - Sufficient condition for an isolated task (no other pending or running aperiodic task) J_a arriving at time A_a, with WCET of C_a and deadline D_a. No assumption on server priority.
    - Worst-case time before service: Ps (isolated task)
    - If $C_a \leq C_s$, J_a will be terminated at last 2 P_s after its arrival and the acceptance test will be: $2\ P_s \leq D_a$
    - In the general case where $C_a \leq C_s$ does not hold, J_a will be terminated at latest

$$P_s + \left\lceil \frac{C_a}{C_s} \right\rceil P_s$$

after its arrival, and the acceptance test is then

$$P_s + \left\lceil \frac{C_a}{C_s} \right\rceil P_s \leq D_a$$

Polling server (5/8)

- Remark
  - The acceptance test is a sufficient condition
  - Source
    - No assumption on the server priority
    - No knowledge of the execution instant of the server within its period

Server based hybrid scheduling for fixed priority scheduling
Polling server (6/8)

- Sufficient condition for an non isolated task (other pending or running aperiodic tasks) J_a arriving at time A_a, with WCET of C_a and deadline D_a, and a server running at the highest priority
  - Aperiodic tasks managed using preemptive EDF
  - At any time t, the total aperiodic computation that has to be served in any interval $[t,t+D_k]$ is equal to the sum of the remaining processing times of the tasks with deadlines $D_i \leq D_k$

$$C_{ape}(t,D_k) = \sum_{i=1}^{k} res_i(t)$$

$res_i(t)$ = time remaining to be executed for aperiodic task Ji

  - Let $C_s(t)$ be the residual server capacity at time t. Since PS has the highest priority, a portion of C_{ape} equal to $C_s(t)$ is immediately executed

Server based hybrid scheduling for fixed priority scheduling
Polling server (7/8)

- Finish time of an aperiodic job Jk
  - $F_k = t + C_{ape}(t,D_k)$ if $C_{ape}(t,D_k) \leq C_s(t)$
  - $F_k = (N_k + G_k) P_s + Res_k$ otherwise

With N_k the number of entire period to execute Jk and G_k no period at which Jk is first executed

$$N_k = \left\lfloor \frac{C_{ape}(t,D_k) - C_s(t)}{C_s} \right\rfloor$$

$$G_k = \left\lfloor \frac{t}{P_s} \right\rfloor$$

$Res_k = C_{ape}(t,D_k) - C_s(t) - N_k C_s$

- Acceptance test of an aperiodic task : $\forall$ k$\in$[1,n_a], with n_a number of pending aperiodic tasks, check that $F_k \leq D_k$
- Remark: acceptance test complexity of $O(n_a)$ with n_a number of pending aperiodic tasks (arrived and not yet terminated)

Server based hybrid scheduling for fixed priority scheduling

Polling server (8/8)

- Pros
  - Simple
    - No modification of feasibility test for hard real time tasks
    - Low complexity acceptance test for aperiodic tasks
  - Better quality of service for aperiodics than background scheduling (resource reservation)
- Cons
  - Server capacity is lost until next period in case there is no aperiodic request pending
  - Lacks reactivity

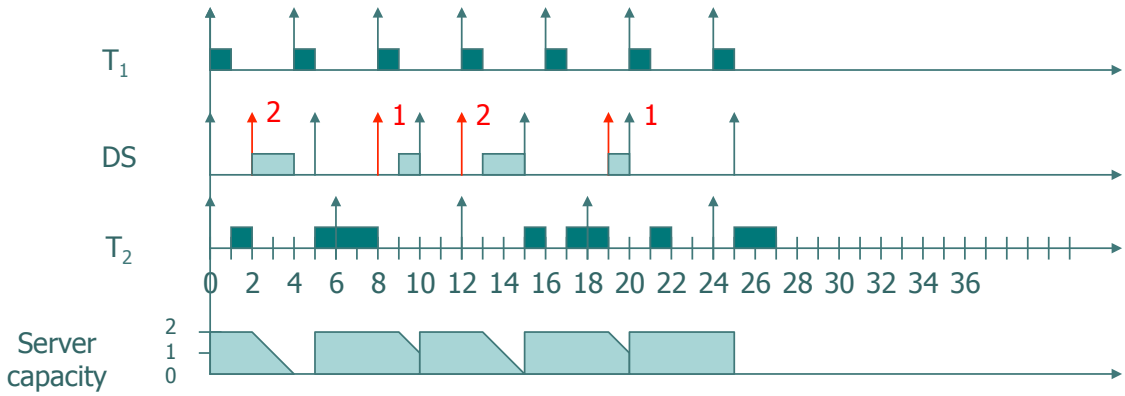
Server based hybrid scheduling for fixed priority scheduling

Deferrable Server (1/5)

- Principle
  - Server dedicated to the execution of aperiodic tasks (idem PS)
  - Difference with PS
    - The server preserves its capacity (until the end of its period) if no aperiodic request is pending

Deferrable Server (2/5)

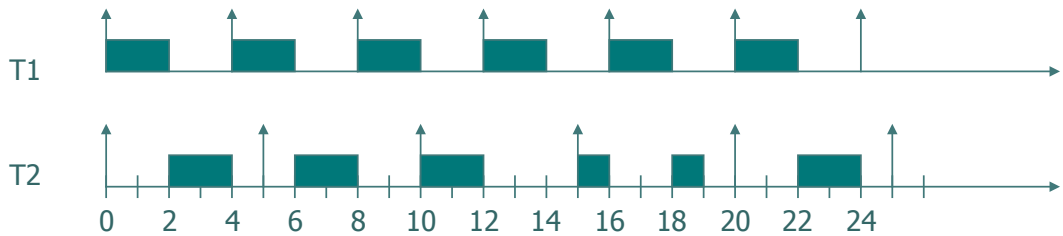
- Example (same example as PS)



Times 2,9,13,19: execution of an aperiodic task (server capacity is maintained)

Deferrable Server (3/5)

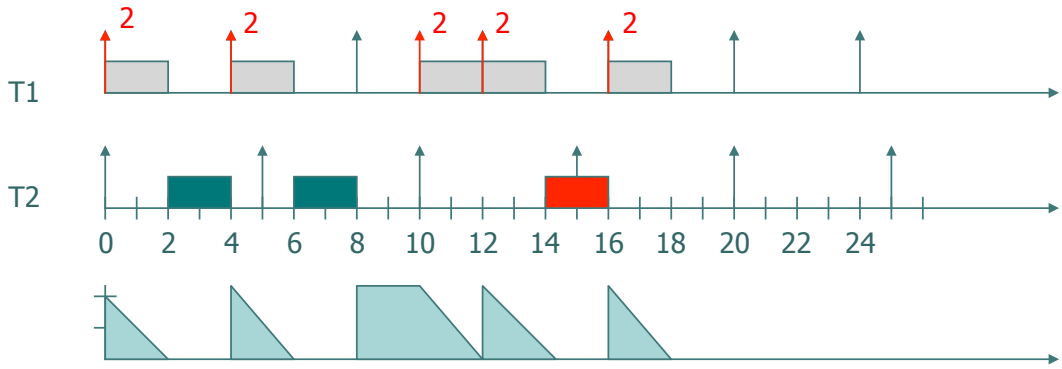
- Feasibility test for periodic tasks
  - A deferrable server is not equivalent to a periodic task anymore (it defers its execution, contrary to periodic tasks)
  - The sufficient feasibility test for RM is not applicable anymore
- Illustration
  - Case a. Two periodic tasks T1 and T2 ($C=C_2=2$, $P_1=4$, $P_2= 5$)



(this system is feasible under RM)

Deferrable Server (4/5)

- Illustration (continued)
  - Case b. T_1 is replaced by a deferrable server DS (other parameters unchanged)



T_2 misses its deadline at time 16

Deferrable Server (5/5)

- Sufficient feasibility test for periodic tasks
  - With Us = (Cs/Ps), deadlines are met if:

$$\sum_{i=1}^{n}\frac{C_i}{P_i} \leq n\left(\left(\frac{U_s+2}{2U_s+1}\right)^{\frac{1}{n}}-1\right)$$

  - With n the number of periodic tasks
- Acceptance test for aperiodic tasks
  - Exist
  - Same algorithmic complexity than PS
  - Accepts more aperiodic tasks than PS

Priority Exchange Server (1/3)

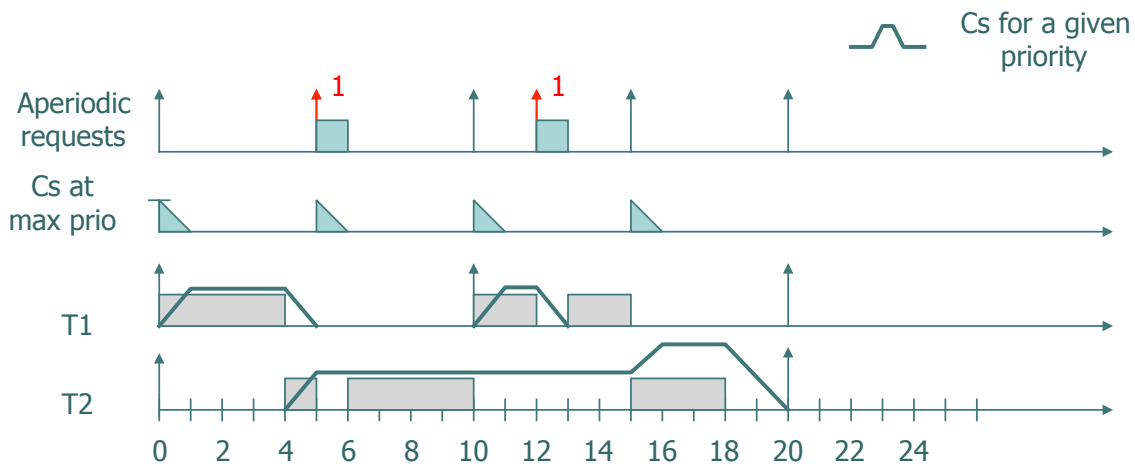
- Principle
  - Server dedicated to the execution of aperiodic tasks (idem PS and DS)
    - In the following, it is assumed that the server executes at the highest priority
  - If aperiodic requests are pending and the server executes at the highest priority, requests are serviced using the available capacity (idem PS and DS)
  - When no request is pending: priority exchange with the highest priority active task
    - The periodic task executes using the server priority
    - The server accumulates a capacity at the priority level of the periodic task
    - Server capacity is lost when no active task

Priority Exchange Server (2/3)

- Example

  $T_1 : C_1 = 4 \quad P_1 = 10 \qquad T_2 : C_2 = 8 \qquad P_2 = 20$

  Server : $C_s = 1$; $T_s = 5$ (highest priority)



Priority Exchange Server (3/3)

- Sufficient feasibility test for periodic tasks

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \ln\left(\frac{2}{U_s+1}\right)$$

Higher bound than DS
(except for very small U_s)

- Acceptance test for aperiodic task
  - Exists
  - Much more complex than under DS

Summary

- Non existence of optimal servers
  - Theorem (Tia-Liu-Shankar). For any set of:
    - periodic tasks scheduled by a given fixed priority scheme
    - Aperiodic requests ordered according to a given queuing policy

  There does not exist any valid algorithm that minimise the response time of every soft aperiodic request

- Remark: similar server based approaches exist when dynamic priority schedulers are used for periodic tasks (e.g. EDF)

Multi-core real-time scheduling

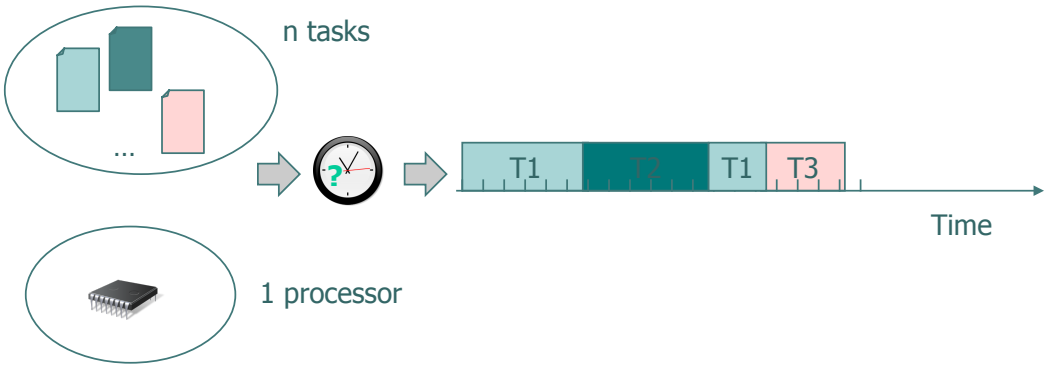
Credits: Anne-Marie Déplanche, Irccyn, Nantes (many slides come from her presentation at ETR, Brest, September 2011)

Multi-core real-time scheduling

- Introduction: problem definition and classification
- Some anomalies of multiprocessor scheduling
- Model and assumptions
- Extension of uni-processor scheduling strategies
- Pfair approaches

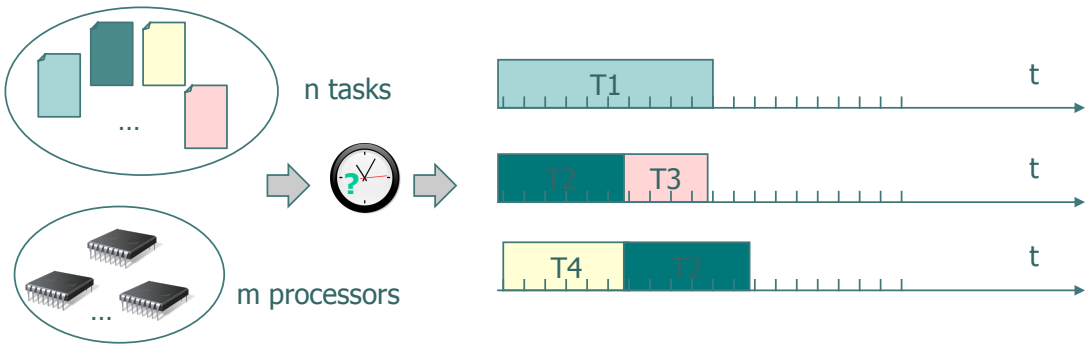
Introduction

- Mono-processor scheduling: one-dimension problem
  - Temporal organization
    - When to start, interrupt, resume every task?



Introduction

- Multi-processor (multi-core) scheduling: two-dimension problem
  - Temporal organization +
  - Spatial organization
    - On which processor execute every task?

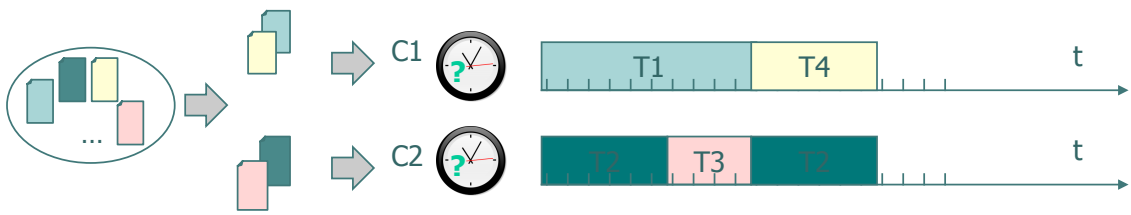


Classification

- Partitioned scheduling
  - Each of the two dimensions is dealt with separately
- Global scheduling
  - Temporal and spatial dimensions are deal with jointly
- Semi-partitioned scheduling
  - Hybrid

Classification: partitioned scheduling

- Each of the two dimensions is dealt with separately
  - Spatial organization: the n tasks are partitioned onto the m cores. No task migration at run-time
  - Temporal organization: Mono-processor scheduling is used on each core



Classification: partitioned scheduling

- Two points of view
  - Number of processors to be determined: optimization problem (bin-packing problem)
    - Bin = task, size = utilization (or other expression obtained from the task temporal parameters)
    - Boxes = processors, size = ability to host tasks
  - Fixed number of processors: search problem (knapsack problem)
- Both problems are NP-hard

Classification: partitioned scheduling

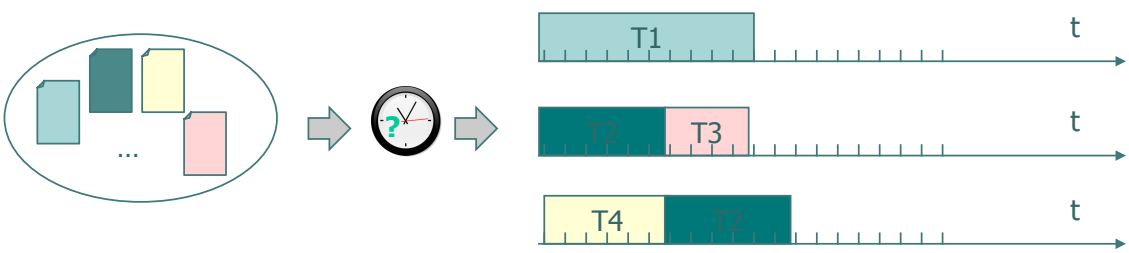
- Optimal mono-processor scheduling strategies: XX
  - RM, DM
  - EDF, LLF (see uni-processor scheduling chapter)
- Bin-packing heuristics: YY
  - FF: First-Fit
  - BF: Best-Fit
  - WF: Worst-Fit, NF: Next-Fit
  - FFD, BFD, WFD: First/Best/Worst-Fit Decreasing
- Partitioning algorithms XX-YY

Classification: partitioned scheduling

- Benefits
  - Implementation: local schedulers are independent
  - No migration costs
  - Direct reuse of mono-processor schedulability tests
  - Isolation between processors in case of overload
- Limits
  - Rigid: suited to static configurations
  - NP-hard task partitioning
  - Largest utilization bound for any partitioning algorithm [Andersson, 2001] $\frac{m+1}{2}$
    (m+1 tasks of execution time 1+ε and period 2)

Classification: global scheduling

- Temporal and spatial dimensions are dealt with jointly
  - Global unique scheduler and run queue
  - At each scheduling point, the scheduler decides when and where schedule at most m tasks
  - Task migration allowed



Classification: global scheduling

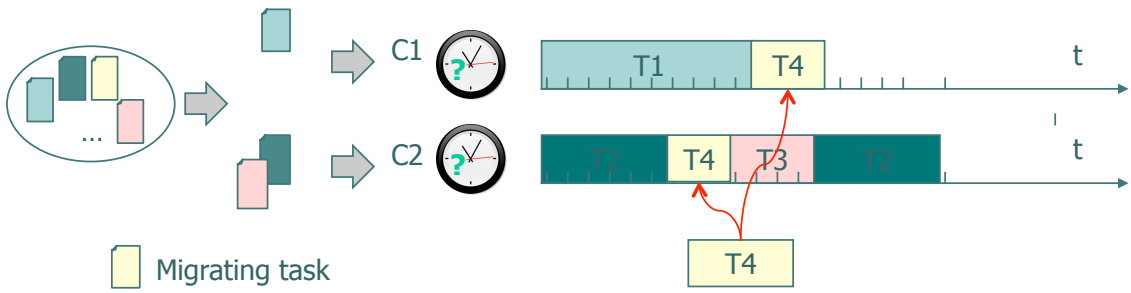
- Benefits
  - Suited to dynamic configurations
  - Dominates all other scheduling policies
    - (if unconstrained migrations + dyn. priorities – see later)
  - Optimal schedulers exist
  - Overloads/underloads spread on all processors
- Drawbacks
  - System overheads: migrations, mutual exclusion for sharing the run queue

Classification: global scheduling

- (Preemptive) global RM/DM/EDF: definition
  - Task priorities assigned according to RM/DM/EDF
  - Scheduling algorithm: the m higher priority tasks are executed on the m processors

Classification: semi-partitioned scheduling

- Partitioned scheduling as far as possible
- Some statically determined tasks may migrate
  - Constraint: migrating tasks (T4 on the example) must execute on a single processor at a time



Terminology

- A task set is schedulable if there exists a scheduling policy such that all deadlines are met
- A task set is schedulable by a scheduling policy if under that scheduling policy all deadlines are met
- A scheduling policy is optimal if it is able to correctly schedule all schedulable task sets
  - Different from the optimality defined before
- Utilization bound of a scheduling policy: utilization U_{lim} below which all task sets meet their deadline

Terminology

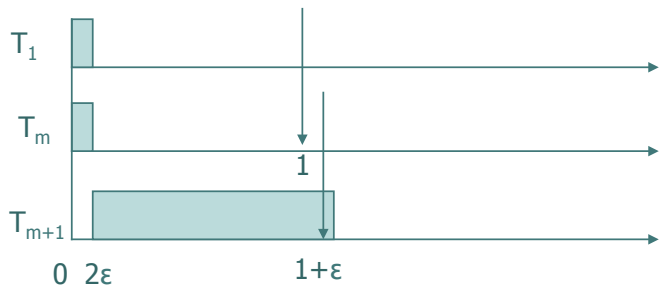
- Priorities
  - Fixed per task (FTP)
  - Fixed per job (FJP)
  - Dynamic per job (DJP)

Overview of global scheduling policies

- Assumptions
  - Tasks
    - Periodic tasks (Pi)
    - Implicit deadlines (Di=Pi)
    - Synchronous tasks (Oi=0 for all i)
    - Independent tasks
    - A single job of a task can be active at a time
  - Architecture
    - Identical processors
    - System costs are neglected (preemption, migration, scheduling policy)

Scheduling anomalies (1/3)

- Dhall's effect [Dhall & Liu, 1978]
  - Periodic task sets with utilization close to 1 are unschedulable using global RM / EDF
  - $n = m+1$, $P_i = 1$, $C_i = 2\varepsilon$, $u_i=2\varepsilon$ for all $1 \le i \le m$
  - $P_{m+1}=1+\varepsilon$, $C_{m+1}=1$, $u_{m+1}=1/(1+\varepsilon)$
  - Task m+1 misses its deadline although U very close to 1

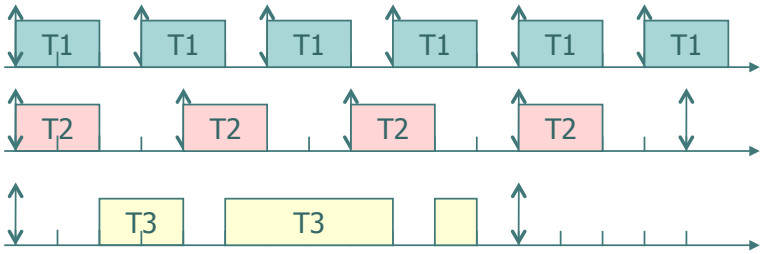


Scheduling anomalies (2/3)

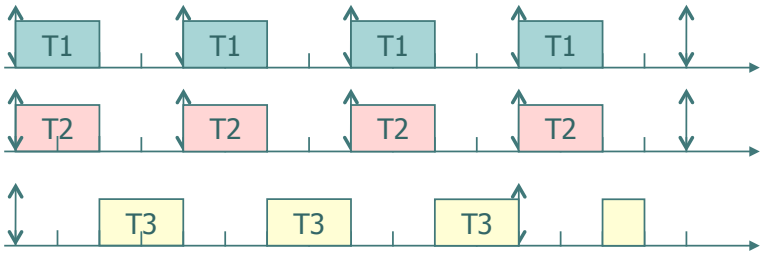
- Period increase for periodic tasks and fixed priorities [Anderson, 2003]
  - $n = 3$, $m=2$, ($P_1= 3$, $C_1=2$), ($P_2=4$,$C_2=2$), ($P_3=12$,$C_3=7$)
  - Schedulable under global RM
  - If P_1 is increased to $P_1=4$ and priorities stay the same, T_3 misses its deadline

Scheduling anomalies (2/3)

- (P_1= 3, C_1=2), (P_2=4,C_2=2), (P_3=12,C_3=7)

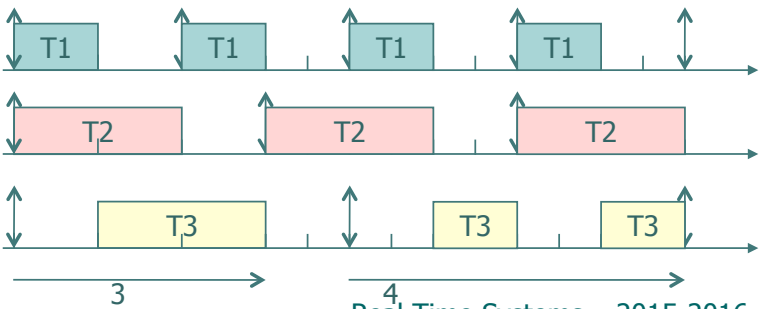


- (P_1= 4, C_1=2), (P_2=4,C_2=2), (P_3=12,C_3=7)



Scheduling anomalies (3/3)

- Critical instant not necessarily the simultaneous release of higher priority tasks
  - n=3, m=2
  - (P_1=2, C_1=1), (P_2=3,C_2=2), (P_3=4,C_3=2)
  - Under RM scheduling
    - Response time of T_3 higher at time 4 than at time 0

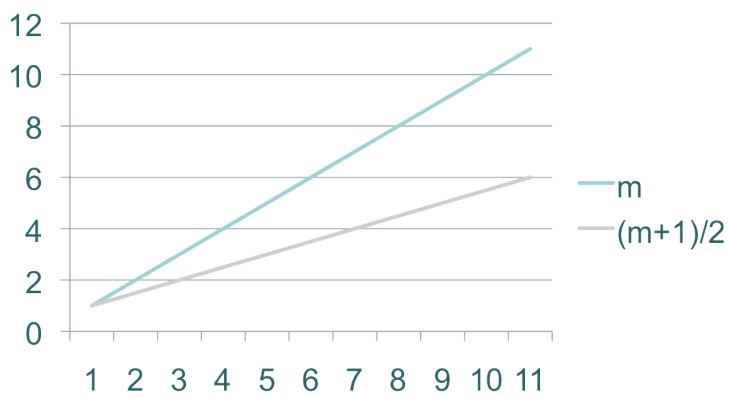


General properties of multiprocessor scheduling (1/2)

- Exact schedulability condition
  - $U \leq m$ and $u_{max} \leq 1$
  - U = total utilization
  - U_{max} = maximum utilization
  - Does not tell for which scheduling algorithm!
- Schedule is cyclic on the hyperperiod H (PPCM(P_i)) for:
  - Deterministic
  - "Without memory" scheduling algorithms

General properties of multiprocessor scheduling (2/2)

- Theorem [Srinavasan & Baruah, 2002]
  - Non existence of FJP (FJP+FTP) scheduling with utilization bound strictly larger than (m+1)/2 for implicit deadline periodic task sets



Global multiprocessor scheduling: detailed outline

- Transposition of uni-processor algorithms
- Extensions of uni-processor algorithms
  - US (Utilization Threshold)
  - EDF(k)
  - ZL (Zero Laxity)
- Pfair approaches (Proportional Fair)

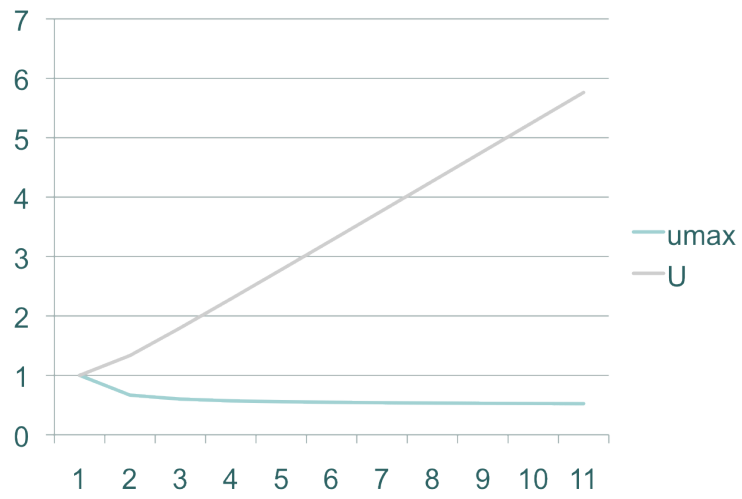
Transposition of uni-processor algorithms (1/2)

- Main algorithms
  - RM (Rate Monotonic) ➔ G-RM, Global RM
  - EDF (Earliest Deadline First) ➔ G-EDF, Global EDF
- Not optimal anymore
- Sufficient schedulability tests (depend on u_{max})

G-RM	G-EDF
$u_{max} \leq m/(3m-2)$ and $U \leq m^2/(3m-2)$	$u_{max} \leq m/(2m+1)$ and $U \leq m^2/(2m+2)$
$u_{max} \leq 1/3$ and $U \leq m/3$	$u_{max} \leq 1/2$ and $U \leq (m+1)/2$
$U \leq m/2 * (1-u_{max}) + u_{max}$	$U \leq m – (m-1)\ u_{max}$

Transposition of uni-processor algorithms (2/2)

(G-EDF, premier test)



Extensions of global RM/EDF: US (Utilization Threshold) policies

- Priority assignment depend on an utilization threshold ξ
  - If $u_i > \xi$, then T_i is assigned maximal priority
  - Else, T_i's priority assigned as in original algorithm (RM/EDF)
  - Arbitrary deterministic tie resolution
- Remarks
  - Still non optimal,
  - Outperforms the base policy
  - Defies Dhall's effect

Extensions of global RM/EDF: US (Utilization Threshold) policies

- Example: RM-US[ξ=1/2]

	Ci	Pi	Ui	Prio
T1	4	10	2/5	2
T2	3	10	3/10	2
T3	8	12	2/3	∞
T4	5	12	5/12	1
T5	7	12	7/12	∞

Extensions of global RM/EDF: US (Utilization Threshold) policies

- Utilization bounds

RM-US		EDF-US	
ξ=m/(3m-2)	$U \le m^2/(3m-2)$	ξ=m/(2m-1)	$U \le m^2/(2m-1)$
ξ=1/3	$U \le (m+1)/3$	ξ=1/2	$U \le (m+1)/2$

- Remarks
  - Utilization bounds do not depend on u_{max} anymore
  - EDF-US[1/2] attains the best utilization bound possible for FJP

Extensions of global RM/EDF: EDF(k)

- Task indices by decreasing utilization
  - $u_i >= u_i+1$ for all i in [1,n]
- Priority assignment depends on a threshold on task index
  - i < k, then maximum priority
  - Else, priority assignment according to original algorithm

Extensions of global RM/EDF: EDF(k)

- Example, EDF(4)

	Ci	Pi	Ui	Prio
T1	4	10	2/5	EDF
T2	3	10	3/10	EDF
T3	8	12	2/3	∞
T4	5	12	5/12	∞
T5	7	12	7/12	∞

Extensions of global RM/EDF: EDF(k)

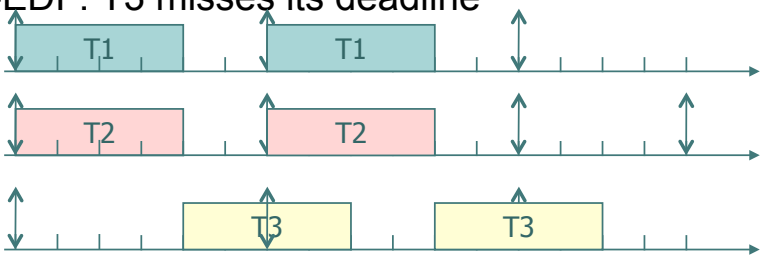
- Sufficient schedulability test $$m \geq (k-1) - \left\lceil \frac{\sum_{i=k+1}^{n} u_i}{1-u_k} \right\rceil$$
  - k_{min} = value minimizing right side of the equation
  - With $k=k_{min}$, utilization bound of (m+1)/2 (best possible for FJP)
  - Comparison with EDF[1/2]
    - Same utilization bound
    - EDF(k_{min}) dominates EDF[1/2]

Extensions of global RM/EDF: ZL (Zero Laxity) policies

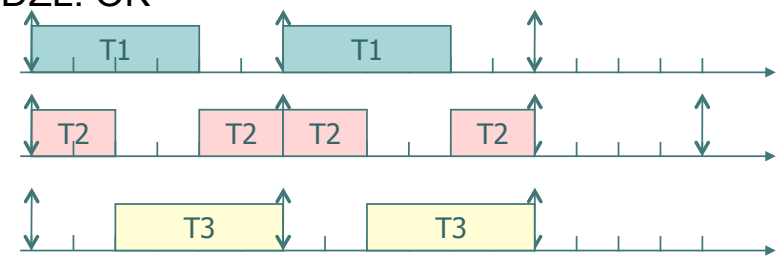
- XX-ZL: apply policy XX until Zero Laxity
  - Maximal priority when laxity reaches zero (regardless of the currently running job), original priority assignment for the others
  - In category DJP (dynamic job scheduling)
- Policies: EDZL [Lee, 1994], RMZL [Kato & al, 2009], FPZL [Davis et al, 2010]
- Utilization bound: (m+1)/2
- Dominates G-EDF

Extensions of global RM/EDF: ZL (Zero Laxity) policies

- Example: m=3,m=2; all Pi to 2, all Ci to 2
  - G-EDF: T3 misses its deadline



  - EDZL: OK



Pfair algorithms

- Principle
- Construction of a Pfair schedule
- Pfair scheduling policies

Pfair algorithms: principle

- Pfair: "Proportionate Fair"
  - [Baruah et al, 1996]
  - Allocate time slots to tasks as close as possible to a "fluid" system, proportional to their utilization factor
- Example
  - $C_1=C_2=3$, $P_1=P_2=6$ ($u_1=u_2=1/2$)
  - Each task will be "approximately" allocated 1 slot out of 2 (whatever the processor)

Pfair algorithms: principle

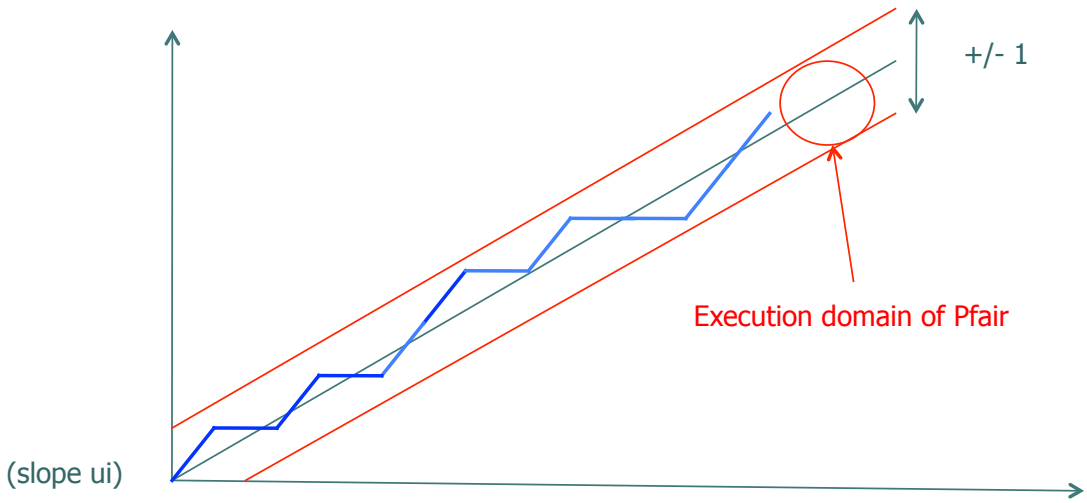
- Lag function: difference between real and fluid execution
  - Discrete time, successive time slots [t,t+1[
  - Weight of a task: $\omega_i=u_i$
- Lag

$$lag(T_i,t) = \omega_i t - \sum_{u=0}^{t-1} S(T_i,u)$$

  - First term: fluid execution
  - Second term: real execution, with $S(T_i,u)=1$ if T_i executed in slot u, else 0
- Pfair schedule: for all time t, lag in interval]-1,1[

Pfair algorithms: principle

- Example



Pfair algorithms: principle

- Property
  - If a Pfair schedule exists, deadlines are met
- Exact test of existence of a Pfair schedule

$$\sum_{i=1}^{n} u_i \leq m$$

  - Full processor utilization!

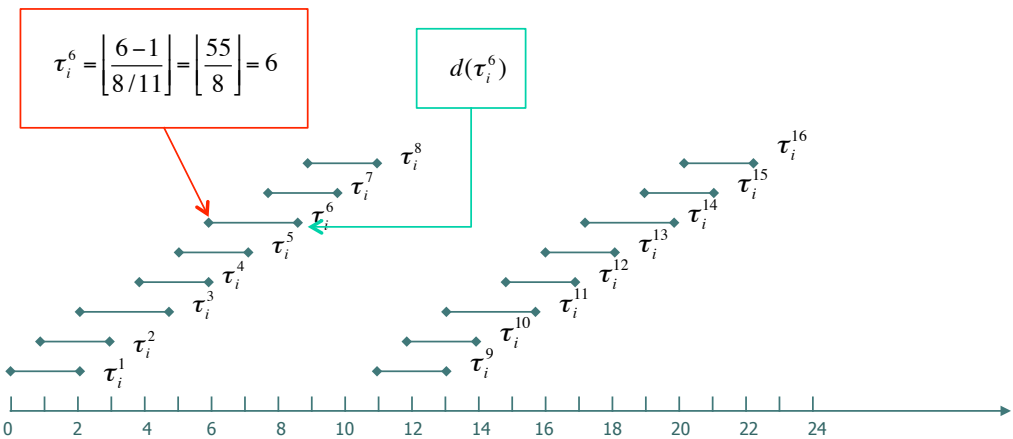
Pfair algorithms: construction of a Pfair schedule

- Divide tasks in unity-length sub-tasks
  - Pfair condition: each subtask j executes in a time window between a pseudo-arrival and a pseudo-deadline
  - Pseudo-arrival: $$r(T_i^j) = \left\lfloor \frac{j-1}{\omega_i} \right\rfloor$$
  - Pseudo-deadline: $$d(T_i^j) = \left\lceil \frac{j}{\omega_i} \right\rceil$$

Pfair algorithms: construction of a Pfair schedule

- Example

$Ci = 8, Ti = 11$



Pfair algorithms: scheduling algorithms

- EPDF (Earliest Pseudo-Deadline First)
  - Apply EDF to pseudo-deadlines
  - Optimal only for m=2 (2 processors)
- PF, PD, PD^2
  - EPDF with non-arbitrary tie breaking rules in case of identical pseudo-deadlines
  - All of them are optimal
  - Most efficient one: PD^2
- Ongoing works
  - Reduce numbers of context switches and migrations while maintaining optimality

Conclusion

- Multi-processor scheduling is an active research area
- Ongoing works
  - Global multi-core scheduling
  - Semi-partitioned scheduling
  - Determining upper bounds of practical factors (preemption, migration, …)
  - Implementation in real-time operating systems