Real-Time Systems

Anno accademico 2009/10
Laurea magistrale in informatica
Dipartimento di Matematica Pura e Applicata
Università di Padova
Tullio Vardanega

Outline

Bibliography

- J. Liu, "Real-Time Systems", Prentice Hall, 2000
- A. Burns, A. Wellings, "Concurrent and Real-Time Programming in Ada", Cambridge University Press, 2007
- A. Burns, A. Wellings, "Real Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time C/POSIX", Addison-Wesley, 2009

5.a Task interactions and blocking

Inhibiting preemption – 1

- In many real-life situations (some parts of) jobs should not be preempted
- Typically during mutually exclusive use of non-reentrant (hence shared) resources
  - Whether directly or indirectly (e.g., within a system call primitive)
- Considerations of data integrity and/or efficiency require that some system level activities must not be preempted

Inhibiting preemption – 2

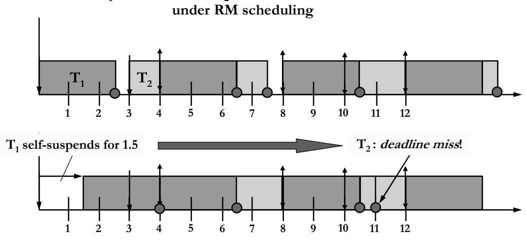
- A higher-priority job J_h that on release finds a lower-priority job J_l executing with disabled preemption gets *blocked* for a $B_i(np)$ time duration
  - Under FPS this is a flagrant case of ***priority inversion***
- The feasibility of J_h now depends on $B_i(np)$ too
  - Under FPS we have $B_i(np) = \max_{(i+1,\ldots,n)} \theta_k$ where $\theta_k \leq e_k$ is the longest non-preemptable execution of job J_k
  - This cost is paid by of J_h only *once* per activation

Self suspension

- A job J_i that invokes suspending operations or that self suspends worsens its response time
- The time penalty $B_i(ss)$ that it incurs may be captured as a degenerate form of blocking
  - $B_i(ss) = \max(\delta_i) + \Sigma_{(j=1,..,i-1)} \min(e_j, \delta_j)$
  - With δ_j the longest duration of self suspension of job J_j
  - J_i may suffer from the self suspension of higher priority jobs (!)
- For a job J_i that may self suspend K times during execution
  - $B_i = B_i(ss) + (K+1)\, B_i(np)$
  - At every resumption J_i may incur $B_i(np)$ again

Example

T_1 = {0, 4, 2.5, 4}, T_2 = {3, 7, 2, 7} ⇒ U = 0.91
under RM scheduling



$B_2(ss) = 0 + \min(2.5, 1.5) = 1.5 > slack(T_2) = 0.5$

Access contention

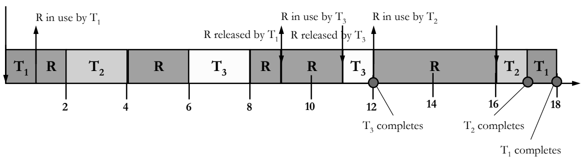
- Access to shared resources causes potential for contention that must be controlled by specialized protocols
- A *resource access control protocol* specifies
  - When and under what condition a resource access request may be granted
  - The order in which requests must be serviced
- Access contention situations may cause priority inversion to arise

Example – 1

Max use of shared resource per execution

T_1 = {-, -, 2, 18, R(4)}, T_2 = {2, -, 3, 17, R(4)} , T_3 = {6, -, 3, 14, R(2)}
under EDF

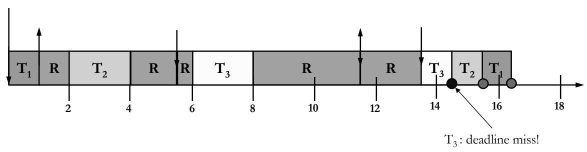
T_1 :: e; R(4); e. T_2 :: e; e; R(4); e. T_3 :: e; e; R(2); e.



Example – 2

T_1 = {-, -, 2, 18, R(2.5)}, T_2 = {2, -, 3, 17, R(4)} , T_3 = {6, -, 3, 14, R(2)}
under EDF

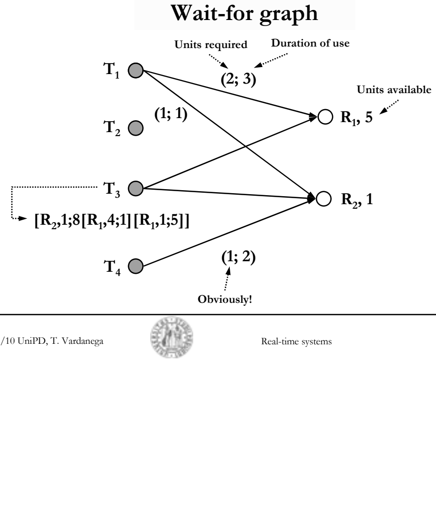
Same as before except with shorter use of R by T_1



Assumptions and notations

- It is safer for real-time design to require that
  - All jobs do not self suspend (directly or indirectly)
  - All jobs can be preempted
- We say that job J_h is ***directly blocked*** by a lower-priority job J_l when
  - J_l is granted exclusive access to a shared resource R
  - J_h has requested R and its request has not been granted
- To study the problem we may want to use a ***wait-for graph***

Example



Resource access control – 1

- ***Inhibiting preemption*** in critical sections
  - A job that requires access to a resource is always granted it
  - A job that has been assigned a resource runs at a priority higher than any other job
    - These two clauses imply each other
    - They jointly prevent deadlock situations from occurring
- They cause ***bounded*** priority inversion
  - At most once per job
    - Reason is obvious
  - For a maximum duration $B_i(rc) = \max_{(k=i+1,..,n)} C_k$
    - For job indices in monotonically non-increasing order and C_k worst-case duration of critical-section activity by job J_k

Critique – 1

- This strategy causes ***distributed overhead***
  - All jobs – including those that do not compete for resource access – incur some time penalty
  - Very unfair hence not desirable
- Better if time overhead is solely incurred by the jobs that actually compete for resource access
  - The priority of the job that is granted the resource must only be higher than that of its competitor jobs
    - The principle of the *ceiling priority*: we shall return to it
  - The resource requirements must be statically known

Resource access control – 2

- ***Basic priority inheritance protocol*** (BPIP)
  - The priority of a job varies over time from that initially assigned
  - The variation follows inheritance principles
- **Protocol rules**
  - Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they take on their *assigned priority*
  - Allocation: when job J requires access to resource R at time t
    - If R is free, R is assigned to J until release
    - If R is busy, the request is denied and J becomes *blocked*
  - Priority inheritance: when job J becomes blocked, job J_l that blocks it takes on J's *current priority* as its *inherited priority* and retains it until R is released; at that point J_l reverts to its previous priority

Critique – 2

- BPIP incurs two forms of blocking
  - ***Direct blocking***: owing to resource contention
  - ***Inheritance blocking***: owing to priority raising
- Priority inheritance is transitive
  - Direct blocking is transitive because jobs may need to acquire multiple resources
- BPIP does not prevent deadlock as cyclic blocking is a devious form of transitive direct blocking
- BPIP incurs *reducible* distributed overhead (i.e., that can be dispensed with)
  - Under BPIP a job may become blocked multiple times when competing for more than one shared resource
- BPIP does not need to have a-priori knowledge of the shared resources
  - It is inherently dynamic

Resource access control – 3

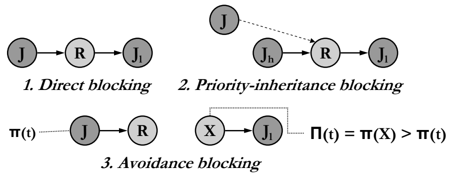
- ***Basic priority ceiling protocol*** (BPCP)
  - As BPIP but with the additional constraint that all resource requirements must be statically known
  - Every resource R is assigned a *priority ceiling* attribute set to the highest priority of the jobs that require R
    - At time t the system has a ceiling $\Pi(t)$ attribute set to the highest priority ceiling of all resources currently in use
    - Otherwise it defaults to Ω < the lowest priority of all jobs

Resource access control – 4

- **Protocol rules**
  - Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they take on their *assigned priority*
  - Allocation: when job J requests access to resource R at time t
    - If R is assigned to another job, the request is denied and J becomes blocked
    - If R is free and J's priority $\pi(t)$ is > $\Pi(t)$, the request is granted
    - If J owns the resource that has priority ceiling = $\Pi(t)$, the request is granted
    - Otherwise the request is denied and J becomes blocked
  - Priority inheritance: when job J becomes blocked, job J_l that blocks it takes on J's current priority $\pi(t)$ until it releases all resources with priority ceiling ≥ $\pi(t)$; then J_l's priority reverts to the level that preceded resource access

Critique – 3

- BPCP is not greedy (whereas BPIP is)
  - Under BPCP a request for a free resource may be denied
- Under BPCP each job J incurs <u>three</u> distinct forms of blocking caused by lower-priority job J_l



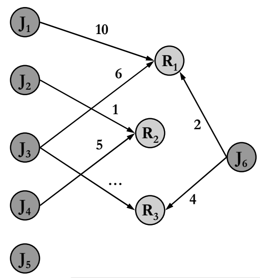
Critique – 4

- ***Avoidance blocking*** is what makes BPCP not greedy and prevents deadlock from occurring
  - If at time t job J has current priority $\pi(t) > \Pi(t)$ then it must be that
    - J will never use any of the resources currently used at time t
    - So won't all jobs with higher priority than J
  - The value of the system ceiling $\Pi(t)$ determines the partition of jobs to which a resource free at time t can be assigned without risk of deadlock
    - All jobs with priority higher than the system ceiling $\Pi(t)$
- **Caveat**
  - To stop job J from blocking itself in the attempt of acquiring multiple resources, BPCP must grant its request if $\pi(t) \leq \Pi(t)$ but J holds the resources {X} with *priority ceiling* = $\Pi(t)$

Critique – 5

- BPCP does not incur reducible distributed overhead because it does not permit transitive blocking
- **<u>Theorem</u>** [Sha & Rajkumar & Lehoczky, 1990]: under BPCP a job may become blocked for at most the duration of one critical section
  - Under BPCP when a job becomes blocked, its blocking can only be caused by a single job
  - The job that causes others to block cannot itself be blocked
    - Hence BPCP does not permit transitive blocking
  - Demonstration: by exercise
- The maximum possible value of that duration is termed the *blocking time* $\mathbf{B_i(rc)}$ due to resource contention
  - $B_i(rc)$ must be accounted for in the schedulability test for J_i

Computing the BPCP blocking time – 1



Directly blocked by

	J2	J3	J4	J5	J6
J1		6			2
J2			5		
J3					4
J4					
J5					

Priority-inheritance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					4

Avoidance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					

B_i(rc) = max value in row *i* across all tables

Computing the BPCP blocking time – 2

- Table "*directly blocked by*" is straightforward
- Table "*priority-inheritance blocked by*"
  - The value set in cell [i, k] is the maximum value found in rows 1, …, i-1; column k in Table "*directly blocked by*"
- Table "*avoidance blocked by*"
  - In the (desirable) case that jobs are assigned distinct priorities, the cells here are identical to those in Table "*priority-inheritance blocked by*" except for the jobs that do not request resources (whose cell value is set to zero)

Resource access control – 4

- ***(Stack-based) ceiling priority protocol***
  - Improves over BPCP in terms of
    - Saving resources especially precious to embedded systems by sharing stack space across jobs
      - To prevent preemption from ever fragmenting a job's stack space we must ensure that no job request for resources may be denied during execution
        - Which BPCP instead allows
      - And of course we must require that jobs do not self suspend
    - Lower algorithmic complexity
      - To reduce the run-time overhead in space and time (e.g., from the dynamic computation of the system ceiling)

Ceiling priority protocol – 1

- ***Stack-based* version** [Baker, 1991]
  - Computation of and updates to ceiling $\Pi(t)$: when all resources are free, $\Pi(t)$ evaluates to Ω; the ceiling value is updated any time a resource is assigned or released
  - Scheduling: on its release time a job stays blocked until its assigned priority $\pi(t) > \Pi(t)$
    - Jobs that are not blocked are dispatched to execution by preemptive priority-driven scheduling
  - Allocation: whenever a job issues a request for a resource, the request is granted

Comments

- Under SB-CPP a job can only begin execution when the resources it needs are free
  - Otherwise $\pi(t) > \Pi(t)$ could not hold
- Under SB-CPP a job that may get preempted does not become blocked
  - The preempting job does certainly not share any resources with the preempted job
- SB-CPP prevents deadlock from occurring
- Under SB-CPP $B_i(rc)$ is computed in the same way as with BPCP

Ceiling priority protocol – 2

- ***Base version***
  - CPP does not use the system ceiling $\Pi(t)$ although the resources continue to have a ceiling priority attribute
  - Scheduling:
    - Each job that does not hold any resource executes at the level of its assigned priority
    - Jobs with the same priority are scheduled in a FIFO ordering (`FIFO_within_priorities`)
    - The current priority of a job that holds any resources takes on the highest value among the ceiling priority of those resources
  - Allocation: whenever a job issues a request for a resource, the request is granted

Summary

- Issues arising from task interactions under preemptive priority-based scheduling
- Survey of resource access control protocols
- Critique of the surveyed protocols

5.b Task interactions and blocking (recap, exercises and extensions)

Credits to A. Burns and A. Wellings

Task interactions and blocking

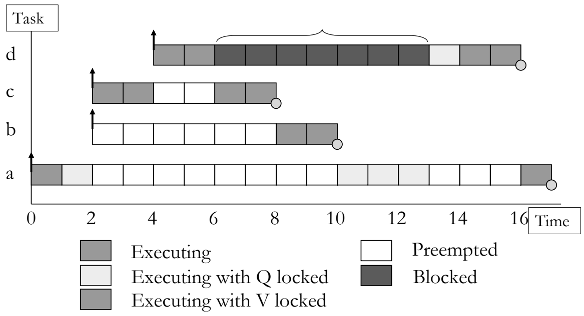
- If a task is suspended waiting for a lower-priority task to complete some required computation then the priority model is, in some sense, being undermined
- It is said to suffer *priority inversion*
- If a task is waiting for a lower-priority task, it is said to be *blocked*

Priority inversion – 1

- To illustrate an extreme example of priority inversion, consider the execution of four periodic tasks: `a`, `b`, `c` and `d`; and two resources: `Q` and `V`; under *simple locking*

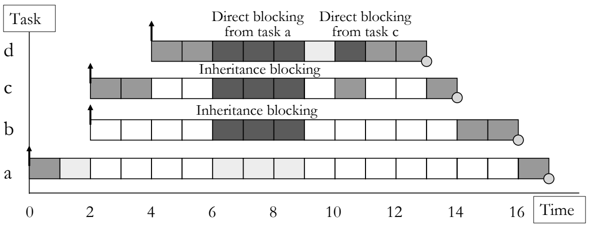
Task	Priority	Execution sequence	Release time
a	1 (low)	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Priority inversion – 2



Priority inheritance (basic version)

- If task `p` is blocking task `q`, then `q` runs with `p`'s priority



Calculating BPI blocking

- If the system has `m` critical sections that can lead to a task being blocked then the maximum number of times that the task can be blocked is `m`
- The upper bound on blocking time B for task `i` with `K` critical sections in the system is given by

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

- With *`usage(k,i)={1 | 0}`* depending on task i's use of the critical section *`k`* and *`C(k)`* the duration of use

Incorporating blocking in response time

$$R_i = C_i + B_i + I_i$$

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

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Ceiling priority protocols

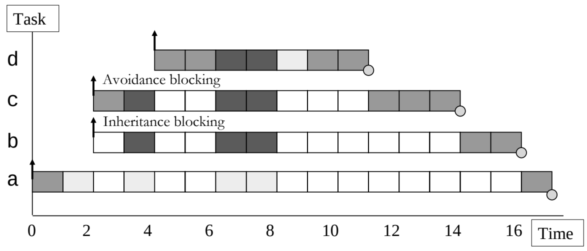
- Two variants
  - *Original* ceiling priority protocol (basic priority ceiling)
  - *Immediate* ceiling priority protocol
- With them on a single processor
  - A high-priority task can be blocked by lower-priority tasks at most once per job
  - Deadlocks are prevented
  - Transitive blocking is prevented
  - Mutual exclusive access to resources is ensured by the protocol itself so that locks are not needed

Original ceiling priority protocol

- Each task has an assigned *static* priority
  - Perhaps determined by deadline monotonic assignment
- Each resource has a static ceiling attribute defined as the maximum priority of the tasks that may use it
- A task has a *dynamic* priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks
- A task can only lock a resource if its dynamic priority is higher than the highest ceiling of any currently locked resource (excluding any that it has already locked itself)

$$B_i = \max_{k=1}^{k} usage(k,i)C(k)$$

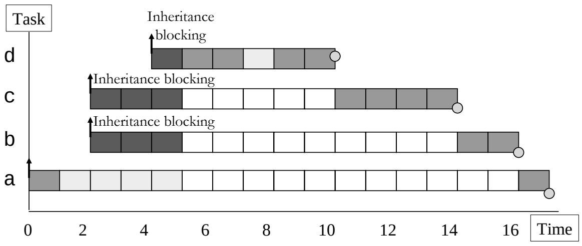
Inheritance with OCPP



Immediate ceiling priority protocol

- Each task has an assigned *static* priority
  - Perhaps determined by deadline monotonic assignment
- Each resource has a static ceiling attribute defined as the maximum priority of the tasks that may use it
- A task has a *dynamic* priority that is the maximum of its own static priority and the ceiling values of any resources it is currently using
- Any job of that task will only suffer a block at release
  - Once the job starts executing all the resources it needs must be free
  - If they were not then some task would have priority ≥ than the job's hence its execution would be postponed

Inheritance with ICPP



OCPP versus ICPP

- Although the worst-case behavior of the two ceiling schemes is identical (from a scheduling view point), there are some points of difference
  - ICPP is easier to implement than OCPP as blocking relationships need not be monitored
  - ICPP leads to less context switches as blocking is prior to job activation
  - ICPP requires more priority movements as they happen with all resource usage
  - OCPP changes priority only if an actual block has occurred
- ICPP is called Priority Protect Protocol in POSIX and Priority Ceiling Emulation in Ada and Real-Time Java

An extendible task model

- Our workload model so far allows
  - Deadlines that can be less than period (D<T)
  - Periodic and sporadic tasks
    - As well as aperiodic tasks under some server scheme
  - Task interactions with the resulting blocking being factored in the response time equations

Extensions

- Cooperative scheduling
- Release jitter
- Arbitrary deadlines
- Fault tolerance
- Offsets
- Optimal priority assignment

Cooperative scheduling – 1

- Unrestrained preemptive behavior is not always acceptable for safety-critical systems
- Cooperative or deferred preemption splits tasks into *slots*
- Mutual exclusion is via non-preemption
- The use of deferred preemption has two important benefits
  - It increases the timing feasibility of the system as it can lead to lower response time values
  - With deferred preemption no interference can occur (by definition) during the last slot of execution

Cooperative scheduling – 2

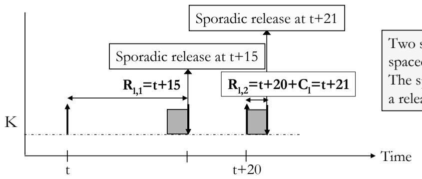
- Let the execution time of the final slot be F_i

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- When the response time equation converges, that is, when $w_i^n = w_i^{n+1}$, the response time is given by $R_i = w_i^n + F_i$

Release jitter – 1

- A big issue for distributed systems and now for multi-core too
- Consider a periodic task K with period 20 releasing at end of job activation a sporadic task on a different processor
  - What is the time between any two subsequent sporadic releases?



Two sporadic releases spaced by 21-15 = 6 (!) The sporadic task has a release jitter $J_s = R_{1,1}$

Release jitter – 2

- Sporadic task `s` released at `0, T-J, 2T-J, 3T-J`
- Examination of the derivation of the schedulability equation implies that task `i` will suffer
  - One interference from task s if $R_i \in [0, T-J)$
  - Two interferences if $R_i \in [T-J, 2T-J)$
  - Three interferences if $R_i \in [2T-J, 3T-J)$
- This can be represented in the response time equation

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

- If response time is to be measured relative to the real release time then the jitter value must be added

$$R_i^{periodic} = R_i + J_i$$

Arbitrary deadlines – 1

- To cater for situations where D > T

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

$$R_i(q) = w_i^n(q) - qT_i$$

- The number of releases is bounded by the lowest value of `q` for which $R_i(q) \le T_i$
- The worst-case response time is then the maximum value found for any `q` $\quad R_i = \max_{q=0,1,2,\ldots} R_i(q)$

Arbitrary deadlines – 2

- When formulation is combined with the effect of release jitter, two alterations to the RTA must be made
- First, the interference factor must be increased if any higher priority tasks suffers release jitter:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j$$

- Second, if the task under analysis can suffer release jitter then two consecutive windows could overlap if response time plus jitter is greater than period

$$R_i(q) = w_i^n(q) - qT_i + J_i$$

Fault tolerance – 1

- Fault tolerance via either forward or backward error recovery always results in extra computation
  - This could be an exception handler or a recovery block.
- In a real-time fault-tolerant system, deadlines should still be met even when a certain level of faults occur
  - This level of fault tolerance is known as the fault model
- If the extra computation time that results from an error in task `i` is C_i^f

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f$$

  - where `hep(i)` is set of tasks with priority equal to or higher than `i`

Fault tolerance – 2

- If F is the number of faults allowed

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f$$

- If there is a minimum arrival interval T_f

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

Offsets

- So far we assumed all tasks share a common release time (the critical instant)

Task	T	D	C	R	U=0.9
a	8	5	4	4	
b	20	10	4	8	Deadline miss!
c	20	12	4	16	

- What if we allowed offsets (phase?)

Task	T	D	C	O	R
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	**8**

Arbitrary offsets are not amenable to analysis!

Non-optimal analysis – 1

- In most realistic systems, task periods are not arbitrary but are likely to be related to one another
- In the previous example two tasks have a common period
- In these situations we can give one of such tasks an offset (of `T/2`) and then we analyze the resulting system using a transformation technique that removes the offset so that critical instant analysis applies
- In the example, tasks `b` and `c` (which has the offset of 10) are replaced by a single *notional* task with period T/2, computation time 4, deadline equal to period and no offset

Non-optimal analysis – 2

- This notional task has two important properties
  - If it is feasible (when sharing a critical instant with all other tasks) then the two real tasks that it represents will meet their deadlines when one is given the half-period offset
  - If all lower priority tasks are feasible when suffering interference from the notional task (and all other high-priority tasks) then they will remain schedulable when the notional task is replaced by the two real tasks (one of which with the offset)
- These properties follow from the observation that the notional task always has no less CPU utilization than the two real tasks

Task	T	D	C	O	R	U=0.9
a	8	5	4	0	4	
n	10	10	4	0	**8**	

Notional task parameters

$$T_n = \frac{T_a}{2} = \frac{T_b}{2}$$
$$C_n = Max(C_a, C_b)$$
$$D_n = Min(D_a, D_b)$$
$$P_n = Max(P_a, P_b)$$

Can be extended to more than two tasks

Priority assignment (simulated annealing)

- **Theorem**: If task p is assigned the lowest priority and is feasible then, if a feasible priority ordering exists for the complete task set, an ordering exists with task p assigned the lowest priority

```
procedure Assign_Pri (Set : in out Task_Set;
                      N   : Natural; -- number of tasks
                      OK  : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, OK); -- is task K feasible now?
      exit when OK;
    end loop;
    exit when not OK; -- failed to find a schedulable task
  end loop;
end Assign_Pri;
```


Summary

- Completing the survey and critique of resource access control protocols using some examples
- Relevant extensions to the simple workload model
- A simulated-annealing heuristic for the assignment of priorities