Real-Time Process Scheduling

郭大維 教授
ktw@csie.ntu.edu.tw
即時暨嵌入式系統實驗室
(Real-Time and Embedded Systems Laboratory)
國立臺灣大學資訊工程系

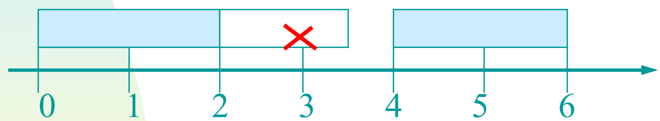
Independent Process Scheduling

Processes share nothing but CPU

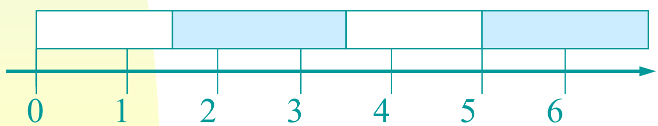
- Papers for discussions:
  - C.L. Liu and James. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," JACM, Vol. 20, No.1, January 1973, pp. 46-61.
- Major references:
  - Aloysius K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. dissertation, MIT, 1983
  - Tei-Wei Kuo and Aloysius K, Mok, "Incremental Reconfiguration and Load Adjustment in Adaptive Real-Time Systems," IEEE Transactions on Computer, 1997, IEEE 12th Real-Time System Symposium, 1991.

Motivation:

- Studying: 2 days per 4 days
  Playing Basketball: 1.5 days per 3 days
- Case 1: Playing basketball (or studying) is more important!



- Case 2: Doing whatever is more urgent!



Questions:

- Can we find an **optimal** scheduler, that always produces a feasible schedule, whenever it is possible to do so? What does ***optimal*** means?
- Can we find a quick schedulability test for a set of processes? Is it simple and accurate?
- Processes whose periods are more harmonically related is likely to be schedulable? Why? More accurate schedulability tests possible? Better system configurations can be found?
- How do we model scheduling overheads, such as the cost of context switching?

Tentative Assumptions:

- Processes are independent.
- Processes are all periodic.
    - We will show you how to model sporadic processes and solve their schedulability problem later.
- The deadline of a request is its next request time.
- A scheduler consists of a priority assignment policy and a priority-driven scheduling mechanism.

The paper of the week:

C.L. Liu and James. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," JACM, Vol. 20, No.1, January 1973, pp. 46-61.

Definitions

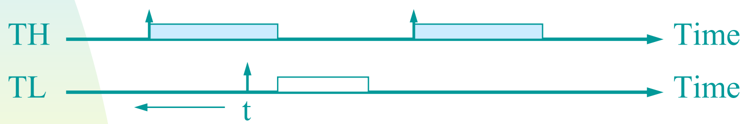
- The **response time** of a request for a process is the time span between the request and the end of the response to that request.
- A **critical instant** of a process is an instant at which a request of that process has the largest response time.
- A **critical time zone** for a process is the time interval between a critical instant and the end of the response to the corresponding request of the process.
- A **critical interval** for a process is the time interval between a critical instant and the deadline of the corresponding request of the process.

An observation: If a process can complete its execution within its critical interval, it is schedulable at all time!

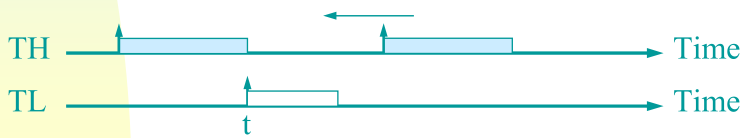
Theorem 1 [LL73] A critical instant for any process occurs whenever the process is requested simultaneously with requests for all higher priority processes.

Proof.

Case 1: TL does not run at t.



Case 2: TL runs at t.



Rate Monotonic Scheduling

- The **rate monotonic priority assignment (RMS)** assigns processes priorities according to their request rates.

Theorem 2 [LL73] If a feasible fixed priority assignment exists for some process set, then the rate monotonic priority assignment is feasible for that process set.

Proof. Start the proof from i = 1. Exchange the priorities of τ_i and τ_{i+1} if their priorities are out of RMS order.

Hint: before: $HPC_{i-1} + c_{i+1} + c_i \leqq p_i$

imply

$$HPC_{i-1}\left\lfloor \frac{p_{i+1}}{p_i} \right\rfloor + c_{i+1}\left\lfloor \frac{p_{i+1}}{p_i} \right\rfloor + c_i\left\lfloor \frac{p_{i+1}}{p_i} \right\rfloor \leq p_i\left\lfloor \frac{p_{i+1}}{p_i} \right\rfloor \leq p_{i+1}$$

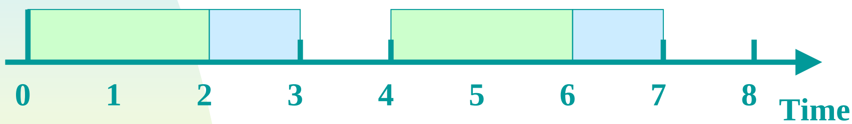
=>

after: $HPC_{i-1} + \text{floor}(p_{i+1}/ p_i)\ c_i + c_{i+1} \leqq p_{i+1}$

where HPC_{i-1} = CPU time consumed by $\{\tau_i ,..., \tau_{i-1}\} = \sum \left\lfloor \frac{p_i}{p_j} \right\rfloor c_j$

An RMS example:

Two processes with p1 = 4, p2 = 5, and c1 = 2, c2 = 1.



An observation: c2 can be increased based on the concept of critical instant!!

Definitions:

- The **utilization factor** of a process τ_i is c_i/p_i .
  - The fraction of CPU time spent in executing τ_i .
- The **utilization factor** of a set of m processes is

$$U = \sum_{i=1}^{m} \frac{c_i}{p_i}$$

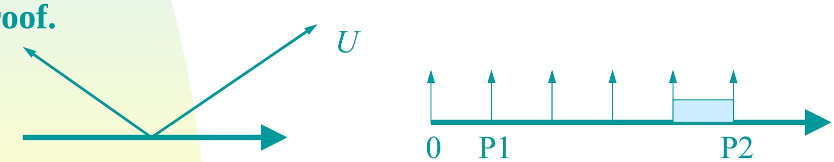
- For a given priority assignment, a process set **fully utilizes** the processor if the priority assignment is feasible for the set and if any increase in the run time of any processes in the set will make the priority assignment infeasible.

- The **achievable utilization factor (least upper bound of utilization factor)** of a scheduling policy ***Ua*** is a real number such that for any process set TΠ, U(TΠ) <= ***Ua*** implies the schedulability of the process setΠ T.

Theorem 3[LL73] For a set of two processes with a fixed priority assignment, the achievable utilization factor is

$$2(2^{1/2} - 1)$$

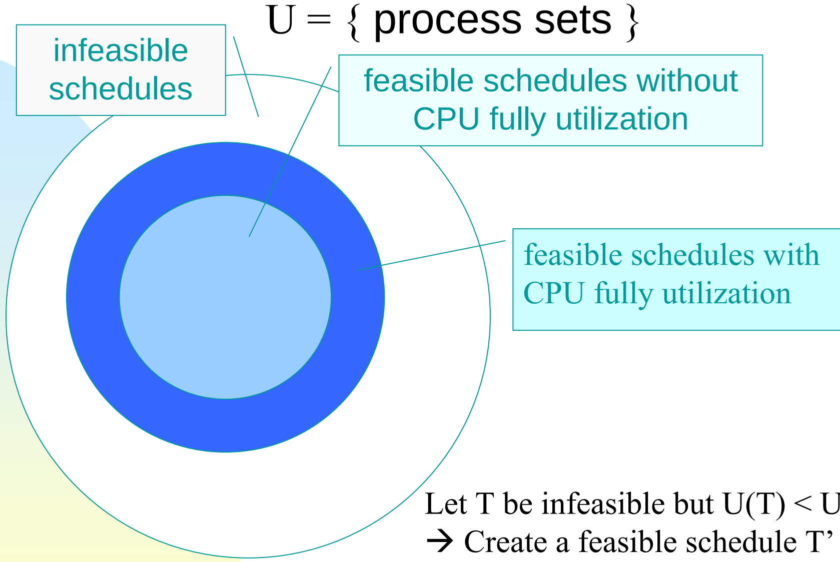
Proof.



C1 = P2 – P1 * floor(P2/P1) 已知 P1 & P2

The minimum *U* occurs when $\left\lfloor \frac{P_1}{P_2} \right\rfloor = 1$ & $\frac{P_1}{P_2} - \left\lfloor \frac{P_1}{P_2} \right\rfloor = 2^{1/2} - 1$

U = { process sets }

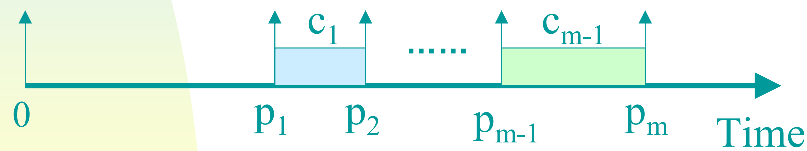


Let T be infeasible but U(T) < Ua
→ Create a feasible schedule T' from such that the CPU is fully utilized → U(T') < U(T) < Ua

Theorem 4 [LL73] For a set of m processes with a fixed priority order and the restriction that the ratio between any two request periods is less than 2, the achievable utilization factor is

$$m(2^{1/m} - 1)$$

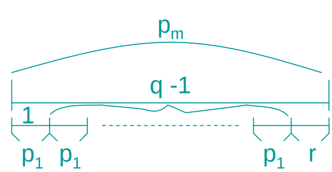
Proof.



Each process in { $\tau_1, \ldots, \tau_{m-1}$ } executes twice within P_m.

Theorem 5 [LL73] For a set of m processes with fixed priority order, the achievable utilization factor is

$$m(2^{1/m} - 1)$$



Proof.

If $p_m = q * p_i + r$, $q > 1$,

then $p_i' = q * p_i$ and increase c_m till the process set fully utilizes the processor again. (c_m <= c_m + $c_i * (q - 1)$).

Show that U is reduced!

$$U' < U + \left[(q-1)c_i/p_m\right] + c_i/p_i' - c_i/p_i$$

or

$$U' \le U + c_i (q-1)\left[1/(qp_i + r) - (1/qp_i)\right]$$

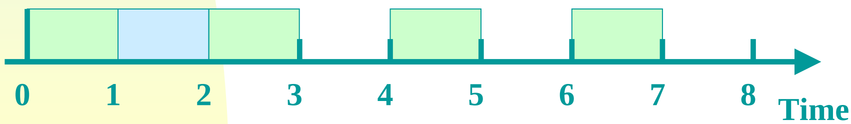
An observation: If a process set are fully harmonically related, U = 100%; otherwise U -> ln2 -> 70%!

Earliest Deadline First

- The **Earliest Deadline First algorithm (EDF)** assigns processes priorities according to the deadlines of their current request.

- **An EDF example:**

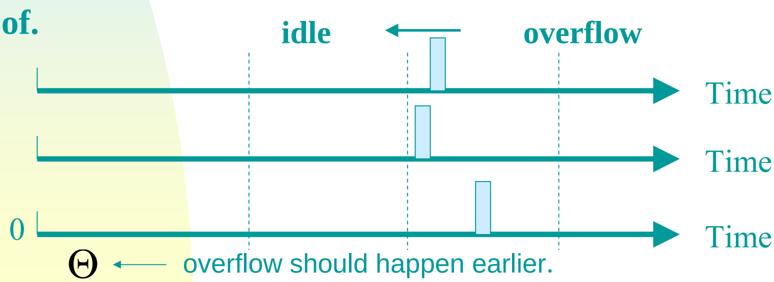
  Two processes with p1 = 2, p2 = 7, and c1 = 1, c2 = 1.



- For a set of processes scheduled according to some scheduling algorithm, we say that an **overflow** occurs at time t if t is the deadline of an unfulfilled request.

Theorem 6[LL73] When the EDF algorithm is used to schedule a set of processes on a processor, there is no processor idle time prior to an overflow.

Proof.



Theorem 7[LL73] For a given set of m processes, the EDF algorithm is feasible if and only if

$$U = (c_1/p_1) + (c_2/p_2) + \ldots + (c_m/p_m) <= 1$$

- The achievable utilization factor of the EDF algorithm is 100%.The EDF algorithm is an optimal dynamic priority scheduling policy in the sense that a process set is schedulable if its CPU utilization is no larger than 100%.
- The achievable utilization factor of the RMS algorithm is about *ln2*. The RMS algorithm is an optimal fixed priority scheduling policy in the sense that if a process set is schedulable by some fixed priority scheduling algorithm, then it is schedulable by the RMS algorithm.

Another thought of the above results?

Theorem For a set of m processes with fixed priority order , the ith process is schedulable if

$$\sum_{j=1}^{i} \frac{c_j}{p_j} \leq i\left(2^{1/i} - 1\right)$$

- Can we extend Theorem 7 in the same way ? No! What is the implication?

What is the priority of a process not reflecting its importance/criticality properly in the real world?

How far can we go from here?

Brain Damage!

Theorem Given a set of m processes, it is schedulable by some fixed priority scheduler if $U \leq 1$.

Proof.

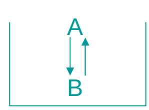
For every time slice, τ_i receive a share of is c_i/p_i .

Within p_i , τ_i receives c_i/p_i!!

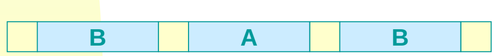
Scheduling Overheads

- **Context Switching**
  - Needed either when a process is preempted by another process, or when a process completes its execution!
  - Stack Discipline

    If process A preempts process B , process A must complete before process B can resume.

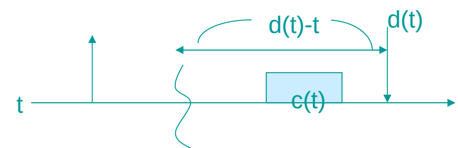


If it is obeyed, charge the cost of preemption (context switching cost) once to the preempting process!



Is there any other optimal dynamic priority scheduling algorithm beside the EDF algorithm?

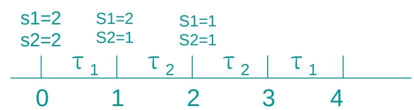
- The **least slack time algorithm (LST)**, which assigns processes priorities inversely proportional to their slack times is also optimal if context switching cost can be ignored [Mok83].
  - The slack time of a process is d(t) - t - c(t).



Proof. Swap the scheduling units of processes out of LST order.

An LST example:

c1 = c2 = 2, p1 = p2 = 4



Summary:

- Definitions
  - Critical instant and critical interval.
  - Utilization factor, achievable utilization factor, and least upper bound of utilization factor.
  - RMS and EDF.
- The achievable utilization factor of RMS and EDF are *ln2* and 100%, respectively.
- The properties and applications of RMS and EDF.
- Modeling of context switching cost, stack discipline, and the properties of least slack time (LST).

Process Synchronization

郭大維 教授
ktw@csie.ntu.edu.tw
即時暨嵌入式系統實驗室
(Real-Time and Embedded Systems Laboratory)
國立臺灣大學資訊工程系

Alternative Approaches

(1) Find and adopt a suboptimal algorithms. Note that a scheduler derived by a scheduling algorithm shall guarantee the schedulability of a process set. A suboptimal algorithm seems only for off-line computations. (Hard Real-Time constraints)

(2)Put as many restrictions on the use of the communication primitives as it is deemed reasonable for programming real-time systems and hope that the restricted scheduling problem can be efficiently solved.

Qs: In general, interprocess coordination by means of semaphores is far too unstructured for real-time analysis. Shall we have a more abstract-level language construct or more structed usages of communication primitives? Shall we provide a language construct for exclusion and synchronization?

Process Synchronization

Processes Might Share Non-Preemptible Resources or Have Precedence Constraints!

- Papers for discussions:
  - L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, 1990.
  - A.K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," IEEE Real-Time Systems Symposium, Dec 1994.

Process Synchronization

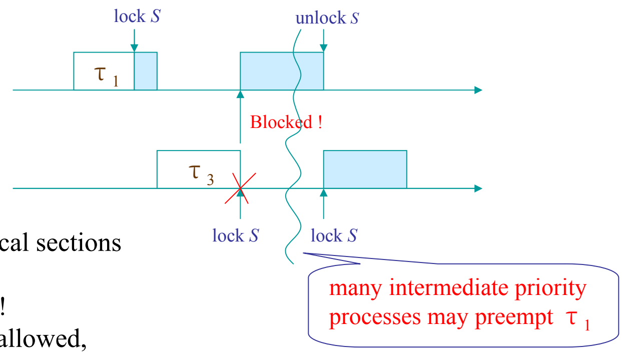
- Motivation
  - Can we find an efficient way to analyze the schedulability of a process set (systematically) ?
  - What kinds of restrictions on the use of communication primitives are needed so as to efficiently solve the restricted scheduling problem?
  - How can we control the priority inversion problem?
  - The lengths of critical sections might be quite different.

The Priority Inversion Problem

- Priority Inversion: A phenomenon where a higher-priority process is forced to wait for the execution of a lower-priority process.

blocking vs *preemption*

Example:



Approaches:

- No preemption of critical sections is allowed!
  =>They must be short!
- If preemption may be allowed, what priority we should assign the execution of a critical section?

Priority Inheritance Protocols: An Approach to Real-Time Synchronization

L. Sha, R. Rajkumar, J.P. Lehoczky,

IEEE Transactions on Computers, 1990.

Notations and Assumptions

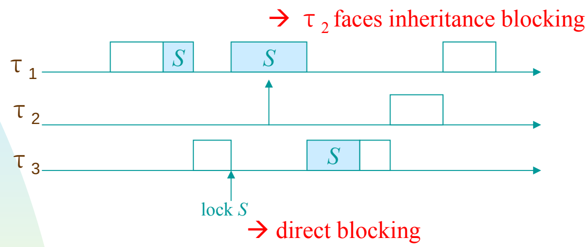
- Notation:
  - $Z_{i, j, k}$: the kth critical section in job (process) J_i guarded by semaphore S_j.
  - $P(S_i)$ and $V(S_i)$ are indivisible operations wait and signal respectively on the binary semaphore S_i.
- Assumptions:
  - Each shared data structure is guarded by a binary semaphore.
  - No job attempts to lock a semaphore that has already been locked.
  - Locks on semaphores will be released before or at the end of a job.
  - A fixed set of jobs executes on a processor.

The Basic Priority Inheritance Protocol

- Priority-Driven Scheduling
  - The process which has the highest priority among the ready processes is assigned the processor.
- Synchronization
  - Process τ_i must obtain the lock on the semaphore guarding a critical section before τ_i enters the critical section.
  - If τ_i obtains the required lock , τ_i enters the corresponding critical section ; otherwise, τ_i is blocked and said to be blocked by the process holds the lock on the corresponding semaphore.
  - Once τ_i exits a critical section, τ_i unlocks the corresponding semaphore and makes its blocked processes ready.
- Priority Inheritance
  - If a process τ_i blocks higher priority processes, τ_i inherits the highest priority of the process blocked by τ_i
  - Priority inheritance is transitive.

Properties

- Blocking



- **Lemma 1** A semaphore S can be used to cause inheritance blocking to job J only if S is accessed by a job which has a priority lower than that of J and might be accessed by a job which has a priority equal to or higher than that of J.

Properties

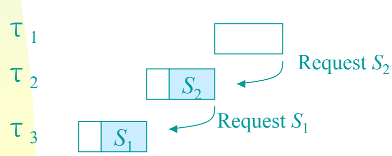
- **Lemma 2** A job J can be blocked by a lower priority job J_L only if J_L has entered and remained within a critical section when J arrives.
- **Lemma 3** A job J_L can block a high priority job J for at most the duration of a critical section regardless of the number of semaphores J and J_L share.

Properties

- **Theorem 4** Under the basic priority inheritance protocol, if there are *n* lower priority jobs , a job *J* can be blocked for at most the duration of *n* critical sections.
- **Lemma 5** A semaphore can be used by at most one lower priority job's critical section to block a higher priority process.

Properties

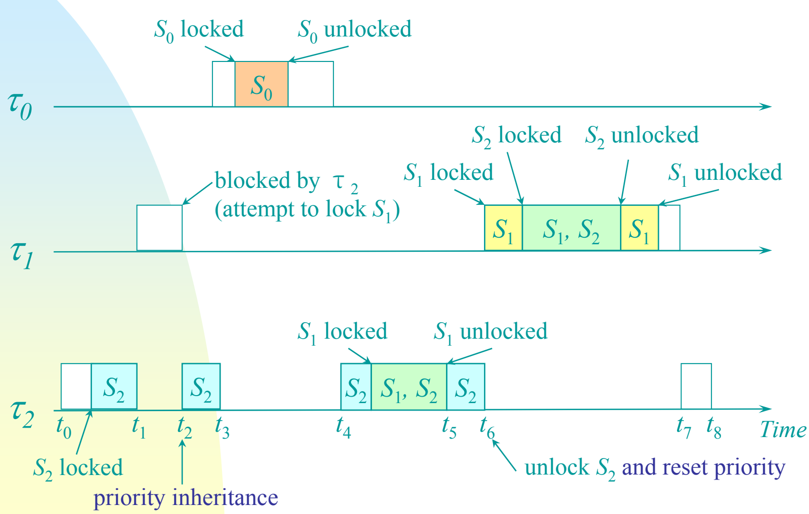
- **Theorem 6** Under the basic priority inheritance protocol , if there are *m* semaphores that can be used to block job *J* , then *J* can be blocked for at most the duration of *m* critical sections.
- Concerns:
  - ◆A chain of blocking is possible.
  - ◆A deadlock can be formed!



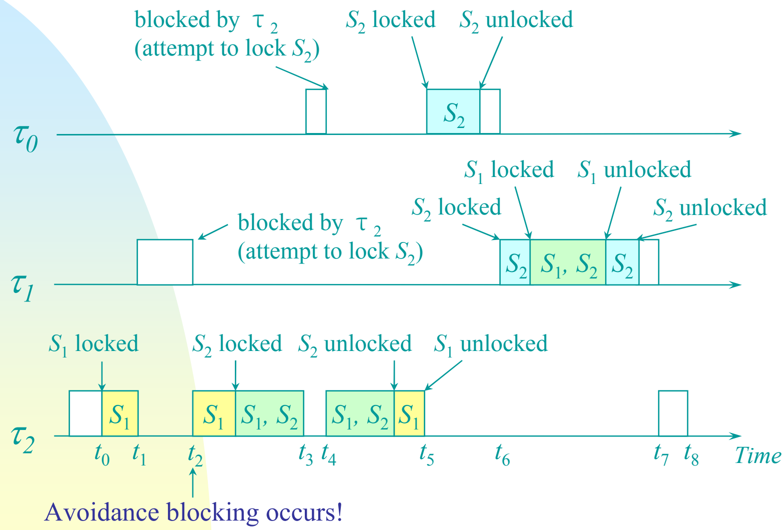
The Priority Ceiling Protocol

- The priority ceiling of a semaphore is the priority of the highest priority job that may lock the semaphore.
- The Basic Priority Inheritance Protocol + Priority Ceiling
- A job *J* may successfully lock a semaphore *S* if *S* is available, and the priority of *J* is higher than the highest priority ceiling of all semaphores currently locked by jobs other than *J*
- Priority inheritance is transitive

Example: Deadlock Avoidance



Example: Chain Blocking Avoidance



* Price paid for deadlock avoidance & chain blocking avoidance!

Properties

- **Lemma 7** A job J can be blocked by a lower priority job J_L only if the priority of job J is no higher than the highest priority ceiling of all the semaphores that are locked by job J_L when J arrives.
- **Lemma 8** Suppose that job J_i preempts job J_j which enters a critical section. Under the priority ceiling protocol, job J_j cannot inherit a priority level which is higher than or equal to that of J_i until J_i completes.

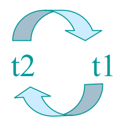
Properties

- **Lemma 9** The priority ceiling protocol prevents transitive blockings.

  Proof. Follows Lemma 8. $t_3 \rightarrow t_2 \rightarrow t_1$

- **Theorem 10** The Priority ceiling Protocol prevents deadlock.

  Proof. Lemma 9 shows that the number of jobs in the blocking cycle can only be 2.



Properties

- **Lemma 11**No job can be blocked for more than one critical section of a lower priority job J_L.

  Proof. Follows Lemma 2 and Theorem 10.

- **Theorem 12** No job can be blocked for more than one critical section of any lower priority job.

  Proof. Lemma 11 suggests that job *J* can only be blocked by *n* different processes' critical sections if $n > 1$. The correctness of the proof them follows Lemma 7.

Properties

- **Theorem 15**: A set of n periodic tasks under the priority ceiling protocol can be scheduled by the rate monotonic algorithm if the following conditions are satisfied:

$$\forall i, \quad 1 \le i \le n, \quad \sum_{j=1}^{i-1} \frac{c_j}{p_j} + \frac{c_i + B_i}{p_i} \le i\left(2^{1/i} - 1\right)$$

where B_i is the worst-case blocking time for τ_i

Proof. Consider B_i as an additional computation requirement.

Properties

$$B_i = \underset{\tau_j \in \beta_i}{Max} \left|\text{critical section}_{j,k}\right|$$

$$\beta_i = \{\, \tau_j \mid Pri(\tau_i) > Pri(\tau_j)\ \&\ \underset{s \in S_j}{Max}(\text{Priority - Ceiling}(s)) \ge Pri(\tau_i)\}$$

$$S_j = \left\{\, \text{s} \,\middle|\, \text{Semaphore } \ s \text{ is accesses by } \tau_j \right\}$$

- More accurate calculations can be derived by considering the relationship between critical sections and their corresponding semaphores.

Properties – Schedulability Analysis

- **Theorem 14** [Loc87] A set of n periodic tasks scheduled by the rate-monotonic algorithm will meet all their deadlines (for all task phasings) iff

$$\forall i,\ 1 \le i \le n, \qquad \min_{(k,l)\in y_i} \sum_{j=1}^{i} c_j \frac{1}{lP_k} \left\lceil \frac{lP_k}{P_j} \right\rceil$$

$$= \min_{(k,l)\in y_i} \sum_{j=1}^{i} U_j \frac{P_j}{lP_k} \left\lceil \frac{lP_k}{P_j} \right\rceil \le 1$$

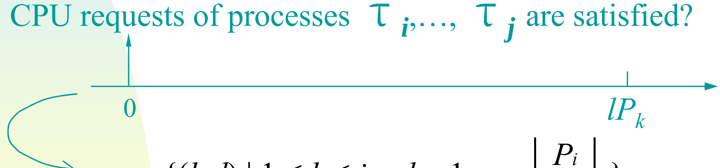
where

$$y_i = \{(k,l) \mid 1 \le k \le i,\quad l = 1, \ldots, \left\lfloor \frac{P_i}{P_k} \right\rfloor \}$$

Properties – Schedulability Analysis

Another Thought:

$$\min_{(k,l)\in y_i} \underbrace{\sum_{j=1}^{i} c_j \left\lceil \frac{lP_k}{P_j} \right\rceil - lP_k}_{\text{CPU requests of processes } \tau_i, \ldots, \tau_j \text{ are satisfied?}} \le 0$$



$$y_i = \{(k,l) \mid 1 \le k \le i,\quad l = 1, \ldots, \left\lfloor \frac{P_i}{P_k} \right\rfloor \}$$

Check every request time of processes, i.e, lP_k.

Properties – Schedulability Analysis

- **Theorem 16** A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm for all task phasings if

$$\forall i,\ 1 \le i \le n,\quad \min_{(k,l)\in y_i}\left[\sum_{j=1}^{i-1}\left(U_j \frac{P_j}{lP_k}\left\lceil \frac{lP_k}{P_j} \right\rceil\right) + \frac{c_i + B_i}{lP_k}\right] \le 1$$

$$\Rightarrow \min_{(k,l)\in y_i}\left[\sum_{j=1}^{i-1} c_j \left\lceil \frac{lP_k}{P_j} \right\rceil + c_i + B_i - lP_k\right] \le 0$$

Example – Schedulability Analysis

τ_1: $C_1 = 40,\ P_1 = 100,\ B_1 = 20,\ U_1 = 0.4$

block

τ_2: $C_2 = 40,\ P_2 = 150,\ B_2 = 30,\ U_2 = 0.267$

block

τ_3: $C_3 = 100,\ P_3 = 350,\ B_3 = 0,\ U_3 = 0.286$

Apply Theorem 15.

$U_1 + B_1 / P_1 = 0.4 + 0.2 = 0.6 < 1$

$U_2 + B_2 / P_2 + U_1 = 0.267 + 0.2 + 0.4 = 0.867$

$U = U_1 + U_2 + U_3 = 0.953$ is far too large!

Example – Schedulability Analysis

1. τ_1 $y_1 = \{(1, 1)\}$ Apply Theorem 16.

   ˅ $C_1 + B_1 \leqq P_1$?? => $40 + 20 < 100$ schedulable !

2. τ_2 $y_2 = \{(1, 1), (2, 1)\}$

   $C_1 + C_2 + B_2 \leqq P_1$?? => $40 + 40 + 30 > 100$

   ˅ $2C_1 + C_2 + B_2 \leqq P_2$?? => $80 + 40 + 30 \leqq 150$ schedulable !

3. τ_3 $y_3 = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)\}$ * $B_3 = 0$

   $C_1 + C_2 + C_3 \leqq P_1$?? => $40 + 40 + 100 > 100$

   $2C_1 + 2C_2 + C_3 \leqq 2P_1$?? => $80 + 80 + 100 > 200$

   ˅ $3C_1 + 2C_2 + C_3 \leqq 3P_1$?? => $120 + 80 + 100 \leqq 300$

   $2C_1 + C_2 + C_3 \leqq P_2$?? => $80 + 40 + 100 > 150$

   ˅ $3C_1 + 2C_2 + C_3 \leqq 2P_2$?? => $120 + 80 + 100 \leqq 300$ schedulable !

   $4C_1 + 3C_2 + C_3 \leqq P_3$?? => $160 + 120 + 100 > 350$

Other Issues

- Implementation Considerations
  - Priority Inheritance Protocol (PIP)
    - Priority queues: one for each semaphore
    - ***P*( )** & ***V*( )** operations must support priority inheritance
  - Priority Ceiling Protocol (PCP)
    - A single priority queue ∵ avoidance blocking
    - A list of blocked semaphores and their respective owners and ceilings

Other Issues

- Other synchronization mechanisms
  - Monitor (used for scheduling mutual exclusion)
    - Define ceiling for each monitor.
    - Operations on monitors are executed in an order complying with PCP.
  - Server-Client Model
    - Server runs at the lowest priority unless it is servicing or requested by some clients.
    - PCP is implemented as usual, when servers are treated as “monitors”!

Other Issues

- ADA rendezvous
  - Priority queues: one per entry call.
  - When there are multiple entries for a server task, the server task must select the highest priority task waiting on one of the entries.
  - PCP is implemented as usual.

Summary

- Definitions
  - Critical instant, critical interval, achievable utilization factor
  - Harmonic base, harmonic chain, adaptive process optimal scheduler, optimal partition of a process set, deferrable server, polling
- RMS & EDF are optimal fixed-priority & dynamic-priority schedulers, respectively. RMS is stable, but EDF has a high achievable utilization factor.
- Process sets with a smaller harmonic base tend to be more schedulable and have better achievable utilization factor.

Summary

- Consider the trade-off between the minimization of the priority inversion problem & the maximization of the concurrency level of a system.
- Consider the ideas behind the schedulability tests of PCP.

After all,

we only have limited pieces of knowledge in predicting the schedulability of a system. However, we begin to understand and find out better ways in allocating resources for processes !

Process Synchronization

Processes Might Share Non-Preemptible Resources or Have Precedence Constraints!

- Papers for discussions:
  - L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, 1990.
  - A.K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," IEEE Real-Time Systems Symposium, Dec 1994.

The Design of Real-Time Programming Systems Based on Process Models

A.K. Mok,

IEEE Real-Time Systems Symposium,

Dec 1994.

Definitions

- A **totally on-line** scheduler makes scheduling decisions independent from *a priori* knowledge of the future request-times of the processes.
- A **run-time** scheduler is the code for allocating resources in response to requests generated at run time, e.g., timer or external device interrupts.
- A run-time scheduler is **clairvoyant** if it has an oracle which can predict with absolute certainty the future request times of all processes.
- A run-time scheduler is **optimal** if it always generates a feasible schedule whenever it is possible for a clairvoyant scheduler to do so.

  Remark: Check definitions in previous transparencies.

Theorem 2 [Mok84] When there are mutual exclusion constraints, it is impossible to find a totally on-line optimal run-time scheduler.

Proof.

Consider two mutually exclusive processes:

$\tau_s : c_s=1, d_s=1, p_s=4$

$\tau_p : c_p=2, d_p=4, p_p=4$

Select the request time of τ_s to fail any totally on-line optimal run-time scheduler. Let τ_p occur at time 0, and τ_s occur at time 1.

The result can be trivially generalized to the cases of multiprocessor by creating a periodic process with $c = p$ for each additional processor.

- **Theorem 4** [Mok84] The problem of deciding whether it is possible to schedule a set of periodic processes which use semaphores only to enforce mutual exclusion is NP-hard.

Proof.

A 3-partition problem:

$A = \{a_1, a_2, \ldots, a_{3m}\}$, B a positive integer, and $w_1, w_2, \ldots, w_{3m}$ be integral weights of elements of A respectively such that $B/4 \leq w_i \leq B/2$, and the sum of w_i is equal to mB. The decision is whether A can be partitioned into m disjoint sets such that each of which has weight B.

Create $3m$ processes, where the ith process has computation time w_i, and period $(mB+m)$, deadline $(mB+m)$. In addition, we create a process τ_{3m+1} with period $(B+1)$, deadline 1, and run-time 1.

The Deterministic Rendezvous Model

- The Primary Purpose:
  Establish precedence constraints between the scheduling blocks of processes.
- Issues:
  - Rendezvous overheads
    Charge it in the scheduling blocks right before the rendezvous.
    => Rendezvous can be preempted!
    No mutual exclusion!
  - Compatibility of processes that rendezvous with each another.
    1. Periodic processes
       → Having periods being an exact multiple of each another.
    2. Otherwise
       → “Server” is the made ready whenever it is needed.

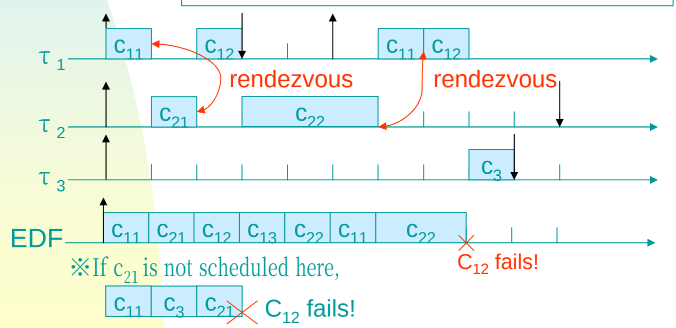
Example

- EDF is not optimal for the process model
  - Example:

τ_1: $C_{11}^* = C_{12} = 1$, $d_1 = 3$, $p_1 = 5$

τ_2: $C_{21}^* = 1$, $C_{22}^* = 3$, $d_2 = p_2 = 10$

τ_3: $C_3 = 1$, $d_3 = 9$, $p_3 = 10$



Deadline Modification

- A deadline-modification technology
  - Consider the computations performed in the interval [0, L]
  - Sort the scheduling blocks generated in [0, L] in a reverse topological order.
  - Set the deadline of the kth instant of τ_{ij} to $(k-1) * p_i + d_i$
  - Revise the deadlines in reverse topological order by the formula:

$$d_s = \min\left(d_s, \{d_{s'} - c_{s'} : s \rightarrow s'\}\right)$$

  - Run-time scheduler repeats the revised deadlines every L time units
    (L is the longest period among processes that rendezvous with each another.)

Deadline Modification

- **Lemma 5** [Mok 84] The feasibility of an instance of the process model is not violated by the above revising technology. The technology will not violate or damage the precedence constraints involving any two processes.

  ＊check the previous example!

- **Theorem 6** [Mok 84] If a feasible schedule exists for an instance of a process model restricted by rendezvous constraints, then it can be scheduled by EDF modified to schedule the ready process which is not blocked by a rendezvous and which has the nearest dynamic deadline.

※Remarks: A pseudo-polynomial-time approach is presented!

The Kernelized Monitor Model

- Idea:

  Processes are given processor time in an uninterruptible quantum, say q.

  - If sporadic processes are of length multiple of q, and each q is treated as critical sections, the process model is reduced to an independent process scheduling with the provision that a process may be interrupted only after it has received an integer number of q. Sporadic processes are called monitors.

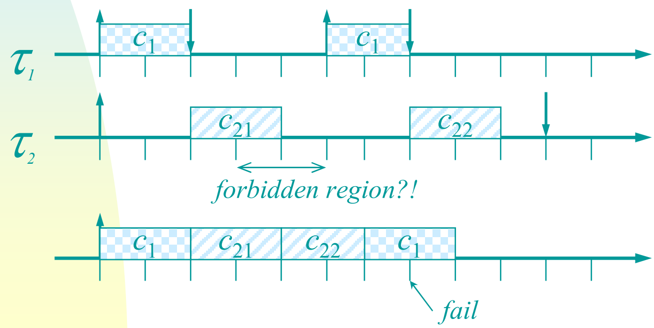
- EDF is still not optimal!

The Kernelized Monitor Model

- An EDF Example: $q = 2$

  $\tau_1 : c_1 = 2, d_1 = 2, p_1 = 5$

  $\tau_2 : c_{21} = c_{22} = 2, d_2 = p_2 = 10$



- How to find the forbidden regions in the interval $[0, L]$ where L is the longest period among the compatible processes? ($L = \text{LCM}(p_i)$)
  1. Each process is considered as a chain of mini scheduling blocks: each of which is of a quantum.
  2. Revise request times and deadlines of the block.

request-time revising
- (1) Sort the blocks in $[0, L]$ in a forward topological order.
- (2) Initialize the request time of the kth instance of each mini scheduling block of T_i to $(k-1)*p_i$.
- (3) Revise the request times in forward topological order by the formula: $r_s = \text{Max}(r_s, \{r_{s'} + q : s' \rightarrow s\})$

deadline revising
- (4) Sort the blocks in $[0, L]$ in a reverse topological order.
- (5) Initialize the deadline of the kth instance of each mini scheduling block of T_i to $(k-1)*p_i + d_i$
- (6) Revise the deadlines in reverse order by the following formula: $d_s = \text{Max}(d_s, \{d_{s'} - q : s \rightarrow s'\})$

3. Compute the set of forbidden regions in $[0, L]$ in the following way:

(1) Sort the request times in a reverse chronological order and determine the forbidden region associated with each request time as follows: Initially, there are no forbidden regions.

(2) For each request time r_s and any deadline d for which $L \geq d \geq d_s$, let $n_{r,d}$ be the number of mini blocks which must be scheduled in $[r_s, d]$. ($r_{s'} \geq r_s$ and $d_{s'} \leq d$)

If $S_{r,d}$ is the latest time at which the first mini blocks must be scheduled, (stack blocks close to each anothor from d)

(I) $S_{r,d} < r_s \rightarrow$ System fails

(II) $S_{r,d} < r_s + q \rightarrow (S_{r,d} - q, r_s)$ is a forbidden region.

(III) otherwise, no action is taken !

*Complexity: $O(n)$ for each request time

- There are no more than n forbidden regions when n is the number of mini scheduling blocks generated in $[0, L]$

  $W = \{(x_i, y_i) : y_i$ is the revised request-time of some mini scheduling block in $[0, L]$, and no process should start pass x_i before $y_i\}$

- The kernelized monitor scheduler:

  At any time t, if t does not lie in a forbidden region, the scheduler allocates the next quantum to the ready process which is not blocked by a rendezvous and has the earliest deadline. If t is in a forbidden region, the process is allowed to be idle until the end of region.

- **Theorem 7**[Mok 84] If a feasible schedule exists for a process set with rendezvous and monitor communication primitives, the kernelized monitor scheduler can schedule it.

A Related Theorem

- **Theorem** [Mok, et al. 87] If EDF is applied to schedule a set of independent periodic processes whose utilization factor is not more than 1, and the scheduler is subject to restriction that every process must be allowed to run for at least q time units before it can be preempted, then no process will ever miss its deadline by more than $q - 1$ time units.
- Schedulability Test:

  Add $q - 1$ time units to the computation time of each process & check the total utilization factor.
- Applications:

  Put conflicting resource accesses in the q-time-unit code for applications based on dataflow graphs.

Overload Detection for Sporadic Processes

- A sporadic process is one whose request times are not known a priori.
- Model a sporadic process by a pair $(p\ ,\ c\ ,\ d)$ where

  c = computation time

  p = minimum separation time between two instances.

  d = deadline

  How should we set the separation parameter p for a sporadic process?

  - It is a tradeoff. If p is too big, we risk having two more requests within p time units . If p is set too small , we might have large gaps between requests and waste CPU capacity.

Overload Detection for Sporadic Processes

Is there an analysis method to evaluate the hypothesis of the choice of timing parameters on system reliability?

- **Theorem 1** [Baruah, Mok, Rosier, RTSS90]
  A set T of n sporadic processes is not feasible (or schedulable by EDF) if either

  (1) $\sum_{i=1}^{n} \frac{c_i}{p_i}$ or,

  (2) $\exists t : t < \min \left\{ P + \max \{d_i\}, \frac{u}{1-u} \max \{p_i - d_i\} \right\}$

  s.t. T fails at or before t, where $P = \mathrm{lcm}\{ p_i \}$

Overload Detection for Sporadic Processes

Define

$$h_R(t) = \sum_{i=1}^{n} c_i \max\{0, \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\}$$

T is feasible iff

$$\forall t \quad h_R(t) \le t$$

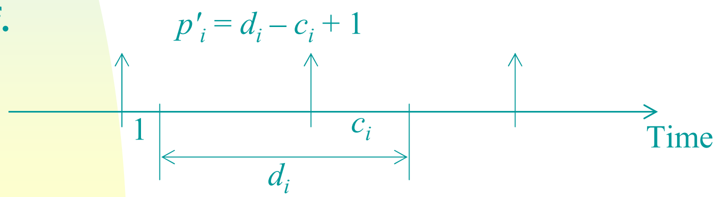
Lemma 3 [Baruah, Mok, Rosier, RTSS90] The minimum t that fails the above formula is the earliest time that the EDF algorithm can report a failure.

Copyright: All rights reserved, Prof. Tei-Wei Kuo, Real-Time and Embedded System Lab, National Taiwan University.

Modeling of Sporadic Processes

Lemma 3[Mok, RTSS84]: Suppose we replace every sporadic process $\tau_i = (c_i, p_i, d_i)$ with a periodic process $\tau'_i = (c'_i, p'_i, d'_i)$ with $c'_i = c_i$, $p'_i = min(p_i, (d_i - c_i + 1))$, and $d'_i = c_i$. If the result set of all periodic processes can be successfully scheduled, then the original set of processes can be scheduled without *a priori* knowledge of the request times of the sporadic processes.

Proof.



Modeling of Sporadic Processes

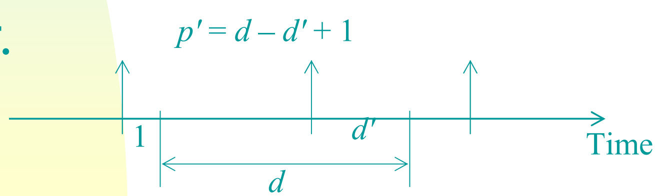
In general, we can replace each sporadic process $\tau_i = (c_i, p_i, d_i)$ with a periodic process $\tau'_i = (c'_i, p'_i, d'_i)$ if the following conditions are satisfied: [Mok, RTSS84]

(1) $d \geq d' \geq c$;

(2) $c' = c$;

(3) $p' \leq d - d' + 1$

Proof.



Summary

- Definitions:
  - totally on-line scheduler, clairvoyant scheduler, optimal run-time scheduler
  - critical region
  - compatibility, rendezvous model monitor model
- The difficulty in finding a totally on-line optimal run-time scheduler.
- The NP-hard nature of the problem in scheduling a set of periodic processes that use semaphores to enforce mutual exclusion.
- EDF with the deadline-revising technology for the deterministic rendezvous model.
- EDF with forbidden regions technology for the kernelized monitor model.