Dr. Dobb's Journal, December 1994

Dr. Dobb's Software Tools for the Professional Programmer

Real-Time Scheduling Algorithms

Achieving predictability for critical applications

Alberto Daniel Ferrari

Alberto is on leave from the Laboratorio de Controle e Microinformatica at the Universidade Federal de Santa Catarina in Florianopolis, Brazil. He can be reached at alberto@uncu.edu.ar.

In a real-time computer system, correctness depends on the time at which the results are given. In practice, this means that for real-time systems to behave properly, some critical subset of a system's tasks should complete their processing *before* their deadlines. Failure to do so can lead to human, environmental, and/or economic damages. Compounding the problem is that emerging systems which are distributed, dynamic, and adaptive will put even more stringent demands on real-time systems. The success of teams of robots working in hazardous environments, on-board space-shuttle systems, and underwater or outer-space autonomous vehicles, for instance, will all be strongly dependent on the timeliness of their computational results.

The predictability of the real-time system is fundamental to achieve this temporal correctness. A predictable system has known temporal bounds for all its actions and can therefore be formally analyzed. Automated techniques can be applied in advance to guarantee the system's critical set. This way, you can have an early warning of a system's inability to meet its timing requirements, and so take appropriate corrective actions. In other words, real-time is not synonymous with "fast," but with "predictable."

Most commercial real-time executives are based in a priority-driven scheme. Current real-time practices for uniprocessor multitasking systems rely on the importance the designer subjectively perceives they have. However, there's no guarantee that critical tasks will meet their deadlines just because the "ready" process with higher priority is the one that executes at every moment; consequently, the temporal correctness of the system remains uncertain. Furthermore, ad hoc priority schemes are not effective for predictable access to shared resources due to the *unbounded priority inversion* phenomenon, in which a higher-priority task is prevented from executing by a lower-priority task for an indefinite period of time.

To alleviate these problems, real-time scheduling algorithms have been devised that force the system's scheduler to base its decisions explicitly on the temporal characteristics of the task set. For most of these algorithms, simple mathematical conditions verify the schedulability of the critical set; satisfying the conditions guarantees the set, even without the designer knowing precisely when any task will be run. This way, you isolate temporal from logical system correctness.

With real-time algorithms, you trade additional information about your system for temporal predictability. To get a more dependable system, you must know about the temporal qualities of the target environment, application, and tasks--then make this information available to the scheduler.

Algorithms for Real-Time Use

When applying real-time algorithms, you must distinguish the algorithm's heuristic--used for scheduling the system--from its schedulability test, which guarantees a given task set. That is, you can use every heuristic that comes to mind to schedule a task group, but you won't have any guarantee of meeting the set's temporal constraints unless you apply a scheduling test.

The algorithm heuristic is the criterion according to which we define a figure of merit for a task. The task with highest-merit value among all ready tasks at every moment is the task that executes (ties are broken arbitrarily). Schedulability tests are valid for a well-defined task model. In this article, I'll examine the rate-monotonic (RM), earliest deadline (EDF), least-laxity dynamic (LLF), and maximum-urgency-first (MUF) algorithms. The temporal model for the algorithms discussed in this article assumes that each task:

- Repeatedly executes at a known fixed rate (its "period").
- Must end before the beginning of its next period (its "deadline").
- Does not need to synchronize with others in order to execute.
- Can be interrupted at any point in time and replaced by another task in the CPU.
- Does not suspend voluntarily.
- Has zero preemption cost (task-switch times and scheduling-algorithm execution load are neglected).
- Is ready while its assigned processing time is not exhausted. After running out of execution units, the task blocks until its next period.

Since the task set is periodic, the base timeline will repeat itself after the LCM (least-common multiple) of the periods of the involved tasks.

A scheduling algorithm is said *static* if the task merit's value is fixed throughout its execution, and *dynamic* otherwise. Static algorithms have lower run-time costs but lack flexibility for adapting to a changing environment; they also need temporal information from all tasks *before* they're actually run.

Rate monotonic (RM). Developed in 1973 by Liu and Leyland, this is one of the most well-known and often-used algorithms for real-time applications. Its scheduling heuristic is shortest-period-first, so it always runs the ready task with shortest period. Since the period of a task is fixed, RM is a static algorithm. The condition for a task set to always meet its deadlines is shown in Figure 1, where C_i is task i's execution time, T_i is its period, and *n* is the number of tasks. The left term is the sum of the individual task loads; the right term defines a limit that goes from 100 percent for one task, to 69 percent for a large number of tasks. In practice, 88 percent is a more realistic value, although when task periods are harmonically related, it could still reach 100 percent. Even when the total task load is higher than the limit, the set may still be scheduled using RM: You just have to prove that every task meets its first deadline when all tasks are started simultaneously.

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_n}{T_n}\leq n*\left(2^{1/n}-1\right)$$

Figure 1 RM schedulability test for a set of n periodic, independent, preemptive tasks. The right term of the equation defines a limit which the total load should not surpass for the task set to be guaranteed.

RM is said to be "stable" because one subset of tasks remains guaranteed when the processor is overloaded. Having ordered the task set by increasing periods, its stable set is composed of the first *n* tasks whose combined load is below the limit. The RM analytical model has been extended to handle complex situations, such as task synchronization (priority-ceiling algorithms) and aperiodic task service (through appropriate servers). RM is also optimum in the sense that it can always schedule a task set if another static algorithm can.

Earliest deadline (EDF). This algorithm schedules the task with closer deadline first. Because the task with the nearest deadline varies with time, this is considered a dynamic algorithm. In this case, a task set is schedulable if the total task load is under 100 percent. Its task model was extended for the same cases as RM, too. EDF is optimal in that if a task set can be scheduled by any algorithm, it can also be scheduled using EDF. It also features the lowest number of task switches but is not stable.

Least laxity (LLF). The merit of the least-laxity (LLF) dynamic algorithm is the lesser "flexibility" to be scheduled in time. This laxity is measured as the difference between the time to deadline and the remaining computation time to finish. A task with negative laxity won't meet its deadline, so laxity provides early detection of temporal failures. This is important if specific actions (other than aborting the faulty task or warning the user) should be taken due to a timing fault: You must ensure that the exception handler has time to execute its code within a given deadline. Again, for a task set to be schedulable, its total load must be under 100 percent; LLF is also optimal, in the same sense as EDF.

Maximum-urgency-first (MUF). Developed at Carnegie Mellon University, this mixed-scheduling algorithm combines the best features of the others: predictability under overload conditions and a scheduling bound of 100 percent for its critical set. The static part of a task's *urgency* is a user-defined *criticality* (high/ low), which has higher precedence than its dynamic part.

First, tasks are ordered by increasing periods: The first *n* highly critical tasks with combined load under 100 percent form the critical set. The highly critical ready task with the least laxity is chosen to run at every moment. If no critical tasks are ready, then tasks with low criticality are selected, again with the least-laxity heuristic. Ties are broken through an optional user priority.

The period of any taski may be modified without dropping it from the critical set, provided its load (C_i/T_i) remains unchanged. This is useful in dynamic environments, where the system needs flexibility to adapt to changing situations.

Table 1 provides a summary of the RM, EDF, LLF, and MUF algorithms. All four may accept new periodic tasks at run time, but a reschedule operation is necessary to guarantee the former critical tasks.

Table 1: Summary of RM, EDF, LLF, and MUF scheduling algorithms.

Algorithm	Type	Schedulability Test	Comment
Rate monotonic (RM)	Static	Total load under limit defined by algorithm task model. Meeting of first deadline.	Stable
Earliest deadline (EDF)	Dynamic	Total load under 100%	Least number of context switches
Least laxity (LLF)	Dynamic	Total load under 100%	Early detection of timing failures
Maximum urgency first (MUF)	Mixed	Critical load under 100%	Stable; 100% schedule bound

Figure 2 shows execution timelines for a task set scheduled by each of these algorithms, with a critical load of 58.3 percent. The resulting schedules differ in the number of (potentially costly) context switches--13, 11, 13, and 13, respectively. Any timeline repeats itself every 24 time units.

Name	Criticality	Period	Execution Time	Task Load
Task A	high	6	2	33.3%
Task B	high	8	2	25.0%
Task C	low	12	3	25.0%

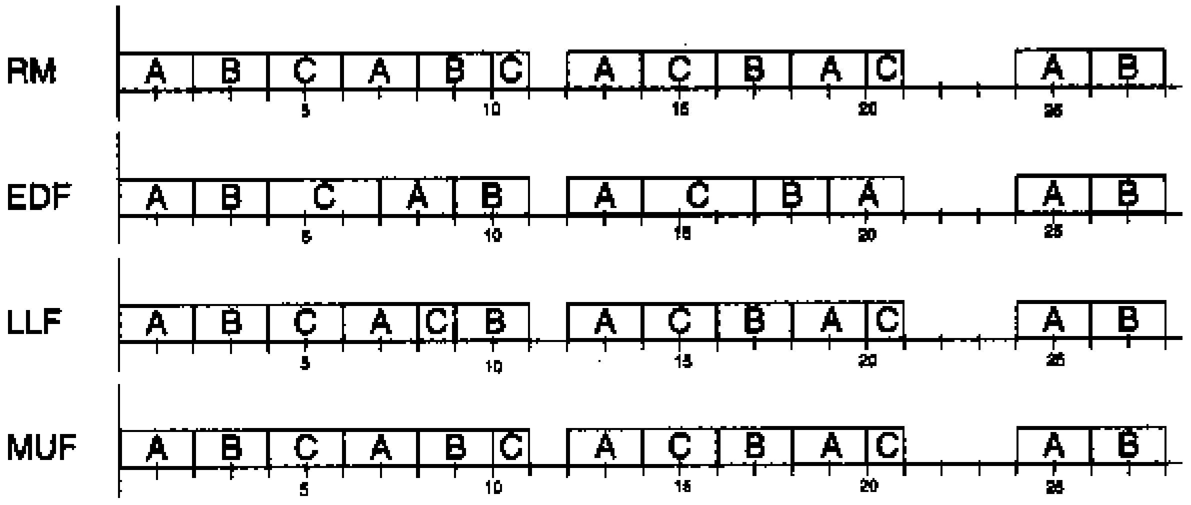


Figure 2 Task set scheduled using: RM, EDF, LLF, MUF. Although total load (83 percent) is higher than the RM limit for this case (78 percent), RM can schedule the set satisfactorily.

For nonstable algorithms such as EDF and LLF, any overload that raises the total load above 100 percent will cause at least one task to miss its deadline. We don't know which one may fail; it could be even a critical task. In RM (as in MUF), tasks outside the critical set may safely overload without any critical task losing a deadline; highly critical tasks may overload the example in up to (78.0/58.3--1)=33.8 percent and still remain guaranteed for RM. In the case of MUF, this increases to (100/58.3--1)=71.5 percent, a certainly higher value. The actual RM-scheduling limit for this case is higher than the value yielded by the formula in Figure 1 (78.0 percent).

Increasing Task B's CPU time from 3 to 5 raises the critical load in 64.3 percent (95.8/58.3--1); see Figure 3. RM cannot handle the critical set anymore, but MUF still guarantees it. Since the timeline is periodic with period 24, no task belonging to the critical set (A and B) ever loses a deadline.

Name	Criticality	Period	Execution Time	Task Load
Task A	high	6	2	33.3%
Task B	high	8	5	62.5%
Task C	low	12	3	25.0%

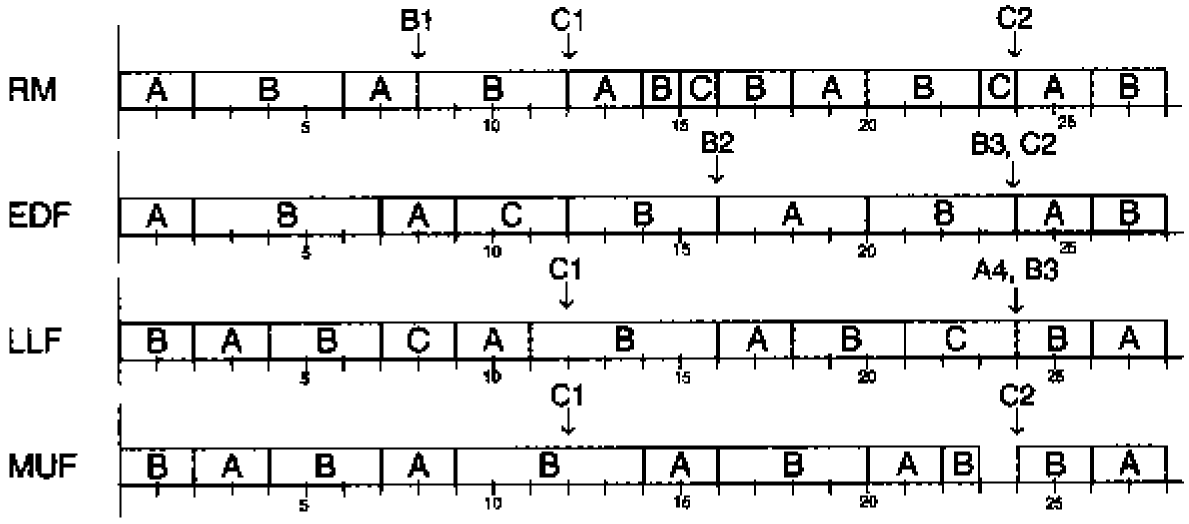


Figure 3 The same as Figure 2, but with a critical overload of 64.3 percent. Note that MUF is the only algorithm that gets the critical set (A and B) scheduled in time.

Implementation

I've included with this article a program that implements a generic execution machine that simulates a system's temporal behavior under a specified scheduling algorithm. Configuration files describe the intended task set and system parameters; see Listing One . The program is composed of a series of modules which are available electronically; see "Availability," page 3.

Listing Two is the code for the simulation machine. The selected algorithm module is called upon entry and exit of the program, and on every time tick of the system when it returns the task; in the latter case, it returns the task chosen to run next.

The system is not priority driven but managed through linked lists. The *deadline_list* holds current instances ordered by increasing deadlines and is searched for failed tasks: When a deadline is reached, its owner should be idle; otherwise, there is a deadline failure and the instance is aborted. *Request_list* contains the time for future task invocations; since the tasks are periodic, a request for the next instance is made upon starting. Finally, *merit_list* contains current instances ordered by merit value.

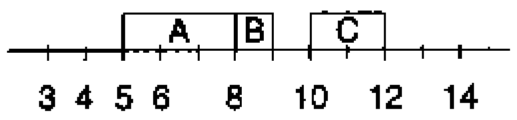
Default_dispatcher(), the *sched_alg()* function for RM and EDF, returns the first ready task from *merit_list*. If the selected task is different from the current task but has the same merit, the current task continues to execute, to save one context switch.

The first three algorithms are in Listing Three . All relevant processing for RM is done during its initialization; after that, the generic dispatcher is called, according to its static nature. Since EDF's merit list is *deadline_list*, it is always modified as instances develop and execute. As the tasks' laxities vary with time, I preferred not to reorder the merit list of LLF on every tick; instead, *merit_list* is linearly searched for the task with least value. Apart from this, *least_laxity()* has almost the same code as *default_dispatcher*.

Finally, Listing Four shows the code of the MUF algorithm. Tasks are first ordered by increasing periods; then the critical set is defined. The code returns the highly critical task with the least laxity, if it exists, or a task with lower criticality and least laxity otherwise. Ties are broken through user priority, which is determined by the order in which tasks appear in the configuration file (preceding tasks have higher user priority than a given task).

Figure 4 shows a sample timeline, plus the program output generated. When a task executes continuously, you capture just its beginning and not its subsequent time periods: This is the case for *a*, which executes from the beginning of 5 to the end of 7. There exists a special task, *idle_task*, whose execution represents the processing of background tasks or tasks without hard deadlines. When a task terminates normally, it blocks until the beginning of its next period.

(a)



```
Z a  bZc Z
345678901234
       1
```

(b)

Name	Criticality	Period	Execution Time	Task Load
Task A	high	6	2	33.3%
Task B	high	8	2	25.0%
Task C	low	12	3	25.0%

```
Timeline (Rate Monotonic):
a b c a b cZa c b a cZ  e b
0123456789012345678901234567
0         1         2
```

Figure 4 Sample timeline and its program's representation. Background processing (or soft) real-time tasks are symbolized by (Z); (b) program output for the example in Figure 2, scheduled with RM.

For linked-list management, I adapted Pugh's skip lists, described and coded in the article "Skip Lists" by Bruce Schneier (*DDJ*, January 1994). The program supports an interface to this library that modifies some calls, mainly because the original doesn't have search-by-value functions for duplicated items. Consequently, I combined the key and the *task_id* into one new key and didn't allow for duplicates.

Conclusion

The main difficulties in applying real-time scheduling techniques stem from the restricted task model for which they're valid. Those models often miss communication relations, precedence orders, and mutual-exclusion problems among tasks that exist in real-world situations. As tasks interact, integrated resource scheduling is also necessary. Algorithms exist that support special cases, in which decisions deal with imprecise results, task-completion value, and so on. However, no algorithm is good for all cases: You'd have to look for the one most suitable to your system's task model.

Additionally, you must know the precise temporal attributes of each process in advance. State-of-the-art real time lacks tools for getting them automatically. The problem is exacerbated when modules synchronize or share resources (such as data). Modern hardware, with cache, pipelining, and the like, also adds temporal uncertainty to the system.

Real-time scheduling may seem unnecessary, but as the project's complexity and size increase, it's the only way to guarantee proper system behavior. It is certainly more predictable than ad hoc techniques. If you're still not convinced, just imagine scheduling 20--30 processes by hand. Even worse, imagine modifying the resulting schedule to accommodate a change in the specifications of one or two processes. Remember, however, that the benefits of real-time scheduling depend on the quality of the temporal information (actual execution-time upper limit of each task, among others) on which you base your analysis.

Bibliography

Liu, C. and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the ACM* (January 1973).

Natarajan, S. and W. Zhao. "Issues in Building Dynamic Real-Time Systems." *IEEE Software* (September 1992).

Sha, L. and J. Goodenough. "Real-Time Scheduling Theory and Ada." *IEEE Computer* (April 1990).

Stankovic, J. "Misconceptions about Real-Time Computing." *IEEE Computer* (October 1988).

Stewart, D. and P. Koshla. "Real-Time Scheduling of Sensor-Based Control Systems." *Proceedings of Eighth IEEE Workshop on Real-Time Operating Systems and Software* (May 1991).

Listing One

```
configuration file: TASK DESCRIPTION FILE
; Task set descriptions, from which tasks are instantiated. Keywords
; are at the line's beginning, and end with ':'.
; Everything in the line after the keywords or values is ignored. Lines
; beginning with '*' are also ignored. No line can be
; longer than MAX_STRING characters, and no name longer than
; MAX_NAME_LENGTH. Please, maintain the order of the parameters in the
; task descriptions.
; start:

test set: Article's Example

MAXTIME= 27    /* timeline's upper limit (starting at 0) */
Number of Application Tasks=3

APPLICATION TASKS DESCRIPTION:
Name       Criticality  Period      Execution_time
Task A,       HIGH,        6,           2.
Task B,       HIGH,        8,           2.
Task C,        LOW,       12,           3.
end.
```

Listing Two

```
/***** RTALGS.C -- Algorithms for Real-Time Scheduling  *****
 ***** Alberto Ferrari -- aferrari@uncu.edu.ar          *****/
/* Simulates the execution of a task set in a multitasking environment,
under several real-time scheduling algorithms. The objective is to
obtain a timeline of the execution, and to show if tasks meet their
deadlines or not. Tasks are assumed to be hard real-time, preemptive,
periodic, with deadline equal to the next instance's arrival
time, and independent (they do not need to syncronize with others in
order to execute). They also do not suspend its execution voluntarily.
All tasks start execution at the same time in the simulation.
Available algorithms for scheduling are Rate Monotonic (RM),
Earliest-Deadline-First (EDF), Least-Laxity-First (LLF), and
Maximum-Urgency-First (MUF), selected in command line. Also selected
in command line is the configuration file, containing the task set
description and the system parameters. Usage:
rtalgs -[e|E|l|L|m|M|r|R] <testvect.name> where: e,
Earliest-Deadline-First (EDF); l, Least-Laxity-First (LLF); m,
Maximum-Urgency-First (MUF), r, Rate Monotonic (RM) *****/
void main( int argc, char *argv[])
{
    node n;
    task_t *task, *new;
  init( argc, argv);
  printf( "\nSelected Scheduling Algorithm: %s,\n", labels[ alg]);
  (sched_alg_init)();
  /* select which task to run next */
  for( sys_time=0;
      (merit_list->header->forward[0]!=NIL || request_list->
                                                     header->forward[0]!=NIL)
          &&  sys_time <= max_time;
      sys_time++){

            /* and if current task emptied its allocated time... */
            if( current!= idle_task  &&  -- current->remaining == 0){
             current->state=DEAD;
             current->cycles++;
             delete_task( deadline_list, current->deadline, current);
             current= idle_task;
          }
          /* Look out for deadline failures */
          while(  key_of( n=first_node_of( deadline_list)) <= sys_time){
              if( (task= n->v)->state != DEAD){
                  printf( "At %d: task %c (\"%s\"),
                                   instance %d, Deadline Failure%s\n",
                      sys_time, task->sys_id, task->name,
                                          task->instance, bell);
              }
              delete( deadline_list, n->key);
          }
          /* if it is time to launch a task... */
          while(  key_of( n=first_node_of( request_list)) <= sys_time){
              task_init( (task= n->v) );
              delete( request_list, n->key);
              insert_task( deadline_list, task->deadline,task);
              insert_task(  request_list, task->deadline, task);
          }
          new = (sched_alg)();
          /* swap and register who's using the processor */
          if( current!=new){
              context_switches++;
              current->state=READY;
              current=new;
              current->state=RUNNING;
          }
          timeline.history[ sys_time]= current->sys_id;
          #ifdef DEBUG
          printf( "%d: %s\n", sys_time, timeline.history);
          #endif
  }
  (sched_alg_end)();

  draw_timeline();
}
  ......
task_t *default_dispatcher( void)
{
    task_t *task;
    if(  (task=first_ready( merit_list))==NULL)
        return( idle_task);
    else if( current== idle_task)
        return( task);
    else /* current task prevails other tasks with same merit */
        return(  (*task->merit == *current->merit)? current : task);
}
```

Listing Three

```
/***** Rate Monotonic (RM) Algorithm *****/
void monotonic_rate_init( void)
{
    int i;
    node n;
    task_t *task;      /* 'task' is the task with 'lesser' period */
    float task_load=0.0, critical_task_load=0.0, schedulability_bound;
    /* in RM case, 'deadline_list' is different from the 'merit_list' */
    deadline_list= newList(); deadline_list ->sys_id= 'D';
    /* calculate n*(2^1/n - 1) */
    schedulability_bound= num_tasks * ( pow( 2.0, 1.0/num_tasks) -1.0);
    printf( "which has a schedulability bound of %.1f%% for %d tasks.\n",
            100.0 * schedulability_bound, num_tasks);
    /* insert tasks in merit_list by increasing periods */
    /* If two tasks with equal period, order them by original sequence */
    for( i=1; i<=num_tasks; i++){
        task->merit= &(task=task_set+i)->period;
        insert_task( merit_list, *task->merit, task);
        insert_task( request_list, 0, task);
    }
    puts( "Critical set is composed of");
    for( task= (n= merit_list->header->forward[0])->v; n!=NIL;
         task= (n= n->forward[0])->v){
        task_load+= (float )task->cpu_time / (float )task->period;
        if( task_load <schedulability_bound){
            critical_task_load= task_load;
            printf( "\t%s,\n", task->name);
        }
    }
    printf( "which accounts for a critical load of %.1f%%, over a total
      system load of %.1f%%\n", 100.0 * critical_task_load, 100.0 * task_load);

    if( task_load<=schedulability_bound) printf( "So, the whole task set IS");
    else if( task_load>1.0) printf( "WARNING: the whole task set IS NOT");
    else printf( "WARNING: the whole task set MAY NOT be");
    printf( " schedulable under RM\n\n");
}
void monotonic_rate_end( void){}
/***** Earliest-Deadline-First (EDF) algorithm *****/
void earliest_deadline_init( void)
{
    task_t *task;
    float task_load=0.0;
    int i;
    printf( "which has a schedulability bound of 100%%\n");
    /* in the EDF case, 'deadline_list' is the same as 'merit_list' */
    deadline_list= merit_list;
    /* insert tasks in merit_list by increasing deadlines */
    for( i=1; i<=num_tasks; i++){
        task->merit= &(task=task_set+i)->deadline;
        task_load+= (float )task->cpu_time / (float )task->period;
        insert_task( request_list, 0, task);
    }
    printf( "Total system task load = %.1f%%\n", 100.0 * task_load);
    if( task_load<=1.0) printf( "So, the whole task set IS");
    else printf( "WARNING: the whole task set IS NOT");
    printf( " schedulable under EDF\n\n");
}
void earliest_deadline_end( void) {}
/***** Least Laxity Algorithm (LLA)  *****/
void least_laxity_init( void)
{
    task_t *task;
    float task_load=0.0;
    int i;
    printf( "which has a schedulability bound of 100%%\n");
    /* in the LLF case, 'deadline_list' is not the same as 'merit_list' */
    deadline_list= newList(); deadline_list ->sys_id= 'D';
    for( i=1; i<=num_tasks; i++){
        task->merit= &(task=task_set+i)->laxity;
        task_load+= (float )task->cpu_time / (float )task->period;
        insert_task( merit_list, *task->merit, task);
        insert_task( request_list, 0, task);
    }
    printf( "Total system task load = %.1f%%\n", 100.0 * task_load);
    if( task_load<=1.0) printf( "So, the whole task set IS");
    else printf( "WARNING: the whole task set IS NOT");
    printf( " schedulable under LLF\n\n");
}
task_t *least_laxity( void)
{
    task_t *least;
    /* all tasks (except 'current') now have one less 'laxity' unit */
    if(  (least=update_laxity_and_get_least( merit_list)) ==idle_task)
        return( idle_task);
    else if( current== idle_task)
        return( least);
    else /* current task prevails other tasks with same merit */
        return( (*least->merit == *current->merit)? current : least);
}
void least_laxity_end( void){}
/* returns idle_task if 'l' is empty */
task_t *update_laxity_and_get_least( list l)
{
    task_t *task, *least;
    node n;
    least= idle_task;
    for( task= (n= l->header->forward[0])->v; n!=NIL;
         task= (n= n->forward[0])->v){
        /* task->laxity(t) = task->deadline - t - task->remaining(t);
         * but now(t)= now(t-1)+1,
         * and task->remaining(t)=  task->remaining(t-1), if task!=current,
         * ==> task->laxity(t) = task->laxity(t-1) -1;
         *****************************************************************/
        /* look out! task->laxity is decremented only if its state is
                                                     READY, because of && */
        if( task->state ==READY  &&  -- task->laxity<0){
                                                   /* if it's eligible... */
            printf( "At %d: task %c (\"%s\"), instance %d, will
                                lose its deadline at %d%s\n",
                sys_time, task->sys_id, task->name,
                                task->instance, task->deadline, bell);
            task->state=BLOCKED;
        }
        if( (task->state ==READY  ||  task->state ==RUNNING)
         &&  task->laxity < least->laxity)
                  least=task;
    }
    return( least);
}
```

Listing Four

```
/***** Maximum-Urgency-First (MUF) Algorithm *****/
list high_crit_l, low_crit_l;
task_t *first;
void maximum_urgency_first_init( void)
{
    node n;
    list temp_list;
    task_t *task;
    float critical_task_load=0.0, task_load=0.0, temp=0.0, load;
    int i, critical_set=TRUE;
    printf( "which has a schedulability bound of 100%%\n");
    /* in the MUF case, 'deadline_list' is not the same as 'merit_list' */
    deadline_list= newList(); deadline_list->sys_id= 'D';
    temp_list= newList(); temp_list->sys_id= 'T';
    high_crit_l= merit_list; high_crit_l->sys_id= 'H';
    low_crit_l= newList(); low_crit_l->sys_id= 'L';
    for( i=1; i<=num_tasks; i++){
        task=task_set+i;
        task->merit= &task->laxity;
        /* use temp_list to order tasks by increasing periods */
        insert_task( temp_list, task->period, task);
        insert_task( request_list, 0, task);
    }
    /* insert tasks in both (high_crit_l and low_crit_l) lists */
    puts( "Critical set is composed of"); /* the first 'n' tasks in
                         'high_crit_l' with combined load less than 100% */
    for( task= (n= temp_list->header->forward[0])->v; n!=NIL;
                                             task= (n=n->forward[0])->v){
        task_load+=(load=(float )task->cpu_time/(float )task->period);
        if( task->criticality ==HIGH){
            if(  (temp+=load)<=1.0  &&  critical_set==TRUE){
                critical_task_load= temp;
                printf( "\t%s,\n", task->name);
                insert_task( high_crit_l, task->period, task);
            }else{
                critical_set= FALSE;
                printf( "WARNING at %d: Highly critical task
                               %c (\"%s\"),\ found NOT Schedulable!!%s",
                               now(), task->sys_id, task->name, bell);
                printf( "\nContinue anyway? (y/[N]) ");
                if( (i=getchar()) == '\n' || i=='n' || i=='N')
                                                                exit(0);
                else insert_task(low_crit_l,task->period,task);
            }
        }else{    /* task->criticality ==LOW */
            insert_task( low_crit_l, task->period, task);
        }
    }
    freeList( temp_list);
    printf( "which accounts for a critical load of %.1f%%, over a total
                        system load of %.1f%%\n",
                        100.0 * critical_task_load,
                        100.0 * task_load);
    if( task_load<=1.0) printf( "So, the whole task set MAY BE");
    else printf( "WARNING: the whole task set IS NOT");
    printf( " schedulable under MUF\n\n");
}
task_t *maximum_urgency_first( void)
{
    task_t *least, *leasth, *leastl;
    /* all tasks (except 'current') now have one less 'laxity' unit */
    leasth= update_laxity_and_get_least( high_crit_l);
    leastl= update_laxity_and_get_least( low_crit_l);
    least= (leasth==idle_task)? leastl : leasth;
    /* all tasks (except 'current') have one less 'laxity' time unit */
    if( least==idle_task)
        return( idle_task);
    else if( current== idle_task)
        return( least);
    else /* current task prevails other tasks with same merit */
        return( (*least->merit == *current->merit)? current : least);
}
void maximum_urgency_first_end( void){}
```