PART 1: Discussion about Cyclic and Priority-based Scheduling Policies

Project Assignment 2, ICE0610
Summer 2004

Leader- Tomás Sánchez López (20042116)
Sungjin Ahn (20042033)

Information and Communication University
tomas@icu.ac.kr

Over the years, considerations about the kind of scheduling policy that should be used for Real-Time systems has generated a heated debate between the experts. It seems like priority scheduling has received a major interest and has been the choice for the design and implementation of many systems, such as the well know and already discussed VxWorks and eCos. However, there exist still those who affirm that cyclic scheduling algorithms offer considerable advantages and insist in proposing those approaches as the best choice for Real-Time applications. In this report we'll try to draw a general picture of each approach to lately list some aspects which might influence in their advantages and disadvantages and will insist in those aspects that we consider bear a special interest. While performing our analysis, moreover, we'll try give our opinion on which considerations we consider more suitable and correct.

1. Introduction

To be specific about the target of this analysis, let's define the scheduling groups we are considering. The two major groups to distinguish are run-time and pre-run-time schedulers. The former term is referred to priority based schedulers, where the decision of which task will execute next is computed dynamically at run time. The second, denoted as pre-run-time schedulers, are those which computation work about the execution line is done previous to run-time. These are also known as cyclic or clock driven schedulers.

2. Priority-based Scheduling

Priority executives are based in the idea that each task in the systems is assigned a priority according to some decision base. Consistent with that priority, the task will be assigned for execution while the system is running. The main question here is: which factor do we use to assign the priorities? Consideration about the answer rises another question: and what happens if the factor, rather than remaining static through the execution, changes with the time? The answer is obvious. If the factor used to assign the priority to a certain task changes at run-time, the priority of that task should change as well. Here we come across the big division inside priority based algorithms, and is the distinction between dynamic-priority and static-priority scheduling.

2.1. Dynamic priority-based scheduling

Traditionally, dynamic priority scheduling has not been considered for the implementation of Real-Time systems. Is true, though, that Real Time Operating Systems such as VxWorks or eCos provide functions

for changing the priority of the tasks at run time. However, this policies can't be considered as dynamic scheduling as the default scheduler themselves don't implement the dynamic functionality. The functions are only offered to the user for programming convenience.

But… why won't the dynamic scheduling be used if in theory it should provide more flexibility (and higher schedulable utilization)? The answer here should be related with the predictability concept (also called determinism, although both terms might differ): When making a scheduling decision in a real-time system, we must assure that the task order that we have inferred is feasible, this is it, that the whole task set in consideration can meet its deadline constraints. Otherwise we might fall into system errors that could be disastrous in hard RT applications (or unacceptable performance decrease in soft RT) Is this status of being sure about the behavior of the system what makes a difference between dynamic and static schedulers. In general, computing the feasibility of a task set for dynamic scheduling systems is a more complex problem than in fixed priority policies. In concrete, it has been proved that, while sporadic and synchronous task systems are considered tractable (polynomial or pseudo-polynomial) in static priority systems, they remain intractable (exponential or NP-complete) in dynamic systems.

EDF is the "standard" scheduler that we think about when we use dynamic priority scheduling. Within the context of preemptive uniprocessor scheduling, it has been shown that the earliest deadline first algorithm (EDF), which at each instant chooses for execution the currently active job with the smallest deadline, is an optimal scheduling algorithm for scheduling arbitrary collections of independent real-time jobs[1]: If its possible to preemptively schedule a given collection of independent jobs such that all the jobs meet their deadlines, then EDF-generated schedule for this collection of jobs will meet all deadlines as well.

2.1. Static priority-based scheduling

As explain before, static priority executives are those who assign fixed priorities in relation with static factors. Those priorities, the same way as any priority based scheduling algorithm, will be used at run-time to make the decision of which task will be run next.

In recent years, Rate Monotonic scheduling algorithm, whose priority factor is related with the rate (frequency) of the tasks (increasing monotonically with it), has gained more importance in the Real-Time scheduling world. In the same way EDF was considered optimal for priority based scheduling, Rate Monotonic is considered optimal for the concrete case of fixed-priority executives, in the sense that if any fixed priority assignment can generate a successful schedule, a Rate Monotonic will generate one.

In the context of *schedulable utilization*[2], is clear that the higher this value it is, the better the algorithm (as the utilization of the processor can be potentially higher). According to this criterion, optimal dynamic-priority algorithms outperform fixed-priority ones, but was at this point where the problems of predictability aroused. In concrete, as the results described by Liu and Layland back in 1973, when the

[1] To define the optimality of algorithms and the framework for their complexity analysis, several conditions must be considered. First, necessary conditions for optimality are preemptiveness and uniprocessor scheduling. The former states that preemption is allowed for all jobs and jobs do not contend for resources (are independent). The second one requires single processor execution environments, as multiple processors would increase scheduling consideration and will produce a different starting point. To end with, it is also considered that the set of jobs for the analysis have arbitrary release times and deadlines.

[2] Defined as the upper bound of the total utilization of the task set considered for a certain scheduling algorithm

Rate Monotonic scheduler is used for a set of tasks, the schedulable utilization value adjust to the formula $n(2^{1/n}-1)$, which in practice means that is dependent on the periodic relationship of the tasks and its value is generally between 88% and 98%.

Aside the discard of the dynamic priority scheduling policies due to the complexity problems exposed before, also the fixed priority executives (and, in concrete, Rate Monotonic) have not been considered traditionally (and, by some, still are not) practical for the Real-Time implementations. This rejection has been induced by the assumptions taken by the model, which state the need for independent periodic execution of all tasks, considered inappropriate for existing real-time applications. However, a big number of later studies extend the work undertook 30 years ago and include dependent tasks and aperiodic jobs. The feasibility of this work can be observed in the decision of modern Real-Time systems to apply fixed priority scheduling rather than cyclic executives.

3. Cyclic Scheduling

Opposing priority based schedulers, cyclic scheduling (also known as clock-driven, pre-run time or off-line scheduling) typically makes use of a pre-computed schedule of all real-time jobs. This schedule is computed off-line before the system begins to execute, and the computation is based on the knowledge of the release times and resource requirements of all the jobs for all times.

Cycling scheduling offers, at a first sight, general advantages for deterministic systems, in which release times and job demands do not vary or vary slightly with time. Optimal schedulers can be computed off-line in this cases without a worry of the complexity of the problem, because this computation time won't affect the system at run-time. But the problem here is obvious: what happens with those systems in which the future work load is relatively unpredictable? Is in this cases where the application of this kind of inflexible policies might drive the system to a disaster, because although is true that some techniques exist to manage small amounts of unpredictable load, the system is inherently static and incapable of offering efficient management to meet unexpected job deadlines. At this point, although waiting for further analysis, we are able to affirm that cyclic scheduling efficiency is application dependent.

Another important limitation to take in account is the one inherited from the cyclic design. As we computed the schedule off-line, the execution line is formed by a sequence of non-preemptible tasks invoked by order. This task list will be repeated cyclically. The problem arises when the frequency of the periodic task is not the same, as the scheduler has to figure out a sequence such that each task is repeated sufficiently often that its frequency requirement is met. In concrete, serious performance drawbacks may appear in cases where the frequencies are not harmonic, as the adaptation into a harmonic sequence by the scheduler will fall into higher processor utilization for the same work. Derived from this, also, we have the cases in which functions exist whose execution time is long compared to the period of the highest rate cyclic task, as this will preempt the later to be able to execute when needed. Solutions to this go though the division of the long task in small parts, which executions is located between the execution of the highest frequency one. At this point we might face problems related with preemption, such as shared resources managed by both tasks, turning the system into what we tried to avoid not using priority preemptable schedulers.

4. Comparison issues and decision points

Now that we have sketched the different scheduling policies that we are going to consider and given some hints about the advantages or disadvantages, is time for a more detailed analysis of the different factors that might led us to decide between them.

4.1. Complexity consequences and overhead

As it was exposed in section 2.1, the complexity of the calculus to decide whether a certain scheduler is feasible for certain task set is very important for policies in which those calculations have to be made at run-time. This consequences where indeed the reason we discarded dynamic priority scheduling.

In cyclic scheduling the complexity of the calculus is irrelevant, as that computation will be maid off-line. This way, the overhead is kept low at run-time, as not even interrupts need to be attended since no task will be processed the event until its cycle begins. Also no problem with shared resources or unexpected context switches should appear. However, overhead coming from the limitations regarding the cyclic nature of the design might cover up these benefits. These limitations where exposed in section 3.

4.2. Predictability

In cyclic scheduling all the future execution line of the system is predetermined before executions time. Assuming that the system is deterministic and no unexpected load occurs, it is clear that all the deadlines of all the tasks in the systems will be met, which fulfills the requirements for hard Real-Time systems. In such a system, cyclic scheduling will, in general, perform better and will be more predictable than any priority based executive.

4.3. Flexibility

As a direct consequence of the off-line computation of its scheduler, cycling scheduling policies offer less flexibility when facing non deterministic systems. On the other hand, for on-line schedulers such as the priority based ones, the price of the flexibility and adaptability is a reduced ability for the scheduler to make the best use of the system resources. Without a prior knowledge of all the life time of the system, it is unlikely for these schedulers to make optimal scheduling decisions, something that a pre-run-time scheduler can certainly do.

But the lack of flexibility doesn't reduce to a gain on better performance. Reduced flexibility of cycling schedulers running in non fully deterministic systems can be disastrous. For example, frame overrun problems might appear in systems even when they are considered predictable (i.e. unanticipated high load stress). If the scheduler allows the task to overrun its frame, it could potentially slip the entire remainder of the time line. On the other hand, the abortion of the task by the scheduler might not be an option at all, mainly when we talk about hard Real-Time systems.

Contrary to the cyclic scheduling, priority-based schedulers flexibility allow handling of unpredicted load, such as the one mentioned previously of a frame overrun. First of all, the problem of the overload is not isolated to an arbitrary time frame or task, but rather is expressed as an overall utilization constrained by the whole system execution bounds. Second, the existence of priorities themselves provides stronger

flexibility when deciding what to do with the overload work and can guaranty that high priority tasks will be executed anyway.

Apart form frame overrun problems, the existence of aperiodic and sporadic tasks might compromise the correct execution of the system. In this cases, also the flexibility is an important factor to handle the extra work. In the case of cyclic scheduling, for example, the limitation of an inflexible execution line makes very difficult dealing with this kind of jobs. Solutions such as the *slack stealing* or the *sporadic servers* have been proposed. The former proposes a relocation of the excessive work into the unused slack time of some of the frames. The later, suggests a periodic task running with the cyclic scheduler (and thus scheduled off-line with the rest of the system tasks) which will attend any aperiodic task that might come into the system. The first solution is obviously risky, as the feasibility of the system's new state depends on the finding of enough slack time to meet the aperiodic task deadline. Moreover, if the execution of the aperiodic task takes more than one cycle and new aperiodic tasks come, note of the positions of its execution period has to be taken to try to relocate the second periodic task, resulting in an extra overhead. About the *sporadic serve* approach, in cycling scheduling with few aperiodic tasks we face a decrease in performance coming from the need to run an server in each cycle that very seldom will be occupied.

The problem of aperiodic and sporadic tasks in the priority based schedulers is easily handled using also the sporadic server approach. Contrary to the use in cyclic scheduling and also the so called *deferrable servers* in priority scheduling, there is no overhead for this server until its execution budget has been consumed. In particular, there is no overhead if there are no aperiodic arrivals.

4.4. Jitter

Jitter refers to the variation in the time a computed result is output to the external environment from cycle to cycle. This is it, low jitter computation at the start of each cycle will end with a highly precise time relationship between its previous and next succeeding completion times. Controlling jitter turns out to be quite important for certain classes of Real-Time systems such as feedback control systems

Its easy to deduce, according to the previous discussion on predictability, that cycling schedulers will produce minimum jitter as the execution line is known beforehand (and no unexpected problems occur). However, preemptive priority based schedulers an inherently more prone to cause higher levels of jitter, as the start and completion times of a task may be delayed arbitrarily due to preemption by higher priority tasks. Although solutions for specific problems might be addressed (such as providing enough high priorities to those jobs with low jitter requirements), it is true that jitter lack of control is one of the main disadvantages of preemptive priority-based schedulers.

5. Conclusion

Although we would have liked to provide further analysis, such as discussion on the needs for different kind of applications, we consider that we have in our hands enough facts to produce a conclusion. To our understanding, fixed priority scheduling groups the major number of advantages for general Real-Time systems, while dynamic scheduling, although producing potentially more schedulable utilization, might be non desirable for systems due to its complexity and exponential cost. However, we would like to note that cycling scheduling is a good approach for those application specific situation in which the system can be guaranteed to be deterministic, but not in any case for general Real-Time applications.