The International Journal of Time-Critical Computing Systems, 18, 7-23 (2000)

Priority Scheduling Versus Pre-Run-Time Scheduling

JIA XU *
Department of Computer Science, York University, 4700 Keele Street, North York, Ontario M3J 1P3, Canada.

DAVID LORGE PARNAS
McMaster University, Department of Computing and Software, Faculty of Engineering, Hamilton, Ontario L8S 4K1, Canada.

Abstract. Builders of real-time systems often use priority scheduling in their systems without considering alternatives. This paper examines one alternative, pre-run-time scheduling, and show that when it can be applied it has significant advantages when compared to priority scheduling schemes.

Keywords: Real-time systems; pre-run-time scheduling; priority scheduling; application constraints; processor utilization; system overhead; schedulability.

1. Introduction

In recent years there has been an increased interest in priority scheduling. It is seen by many as the method of choice for designing real-time systems, and has been adopted or recommended by many government agencies for some very important projects. For example, it has been reported to be adopted by NASA, FAA, the U.S. Navy, the European Space Agency, among others.

Many of the intended applications are safety-critical, and there can be very serious consequences if timing constraints are not satisfied. Because of the importance of these projects, the characteristics of the priority scheduling approach needs to be carefully examined, to determine whether priority scheduling is the best possible choice.

Several papers have been published that express the view that the priority scheduling approach is better than alternative approaches, e.g., (Locke, 1992). This paper offers another view. A debate on this issue should help real-time system designers to make informed choices on which scheduling method they should use.

Some readers will find some of the observations made in this paper about the strengths and weaknesses of certain techniques to be obvious. However, (1) these issues are never discussed explicitly in other papers, and (2) people build systems in apparent ignorance of the points made in this paper. An explicit discussion of these issues will be of value to many developers of real-time systems.

 This work was supported in part by Natural Sciences and Engineering Research Council of Canada grants to J. Xu and D. L. Parnas. This paper was presented at the 23rd IFAC/IFIP Workshop on Real-Time Programming, Shantou, June 23-25, 1998.

Throughout this paper, it is assumed that the hard-real-time system includes a relatively large number of processes that have stringent timing constraints, that the processes have different timing characteristics and interdependencies, and that the processor utilization factor is not very low, i.e., there is not much spare CPU capacity available. The U.S. Navy's A-7E aircraft operational flight program is a well documented example (Heninger, Kallander, Parnas, and Shore, 1978, Faulk and Parnas, 1988).

2. Processes and Schedules

As explained in Xu and Parnas (1993), in order to provide predictability in a complex hard-real-time system, the major characteristics of the processes must be known, or bounded, in advance, otherwise it would be impossible to guarantee a priori that all timing constraints will be met.

2.1. Periodic Processes

A periodic process consists of a computation that is executed repeatedly, once in each fixed period of time. A typical use of periodic processes is to read sensor data and update the current state of internal variables and outputs.

A periodic process p can be described by a quadruple (r_p, c_p, d_p, prd_p). prd_p is the *period*. c_p is the worse case *computation time* required by process p. d_p is the *deadline*, i.e., the duration of the time interval between the beginning of a period and the time by which an execution of process p must be completed in each period. r_p is the *release time*, i.e., the duration of the time interval between the beginning of a period and the earliest time that an execution of process p can be started in each period. We assume that r_p, c_p, d_p, prd_p as well as any other parameters expressed in time have integer values. A periodic process p can have an infinite number of *periodic process executions* p_0, p_1, p_2, ..., with one process execution for each period. For the ith process execution p_i corresponding to the ith period, p_i's release time is $r_{p_i} = r_p + prd_p \times (i-1)$; and p_i's deadline is $d_{p_i} = d_p + prd_p \times (i-1)$.

2.2. Asynchronous Processes

An asynchronous process consists of a computation that responds to internal or external events. A typical use of an asynchronous process is to respond to operator requests. Although the precise request times for executions of an asynchronous process a are not known in advance, usually the *minimum amount of time between two consecutive requests* min_a is known in advance. An asynchronous process a can be described by a triple (c_a, d_a, min_a). c_a is the worse case *computation time* required by process a. d_a is the *deadline*, i.e., the duration of the time interval between the time when a request is made for process a and the time by which an execution of process a must be completed. An asynchronous process a can have an infinite number of *asynchronous process executions* a_0, a_1, a_2, ..., with one process execution for each asynchronous request. For the ith asynchronous process execu-

tion a_i which corresponds to the ith request, if a_i's request time is r_{a_i}, then a_i's deadline is $d_{a_i} = r_{a_i} + d_a$.

2.3. Schedules

We introduce the notions *process execution unit* and *processor time unit.* If a periodic process p or an asynchronous process a has a computation time of c_p or c_a, then we assume that that process execution p_i or a_i is composed of c_p or c_a process execution units. Each processor is associated with a *processor time axis* starting from 0 and is divided into a sequence of processor time units.

A *schedule* is a mapping from a possibly infinite set of process execution units to a possibly infinite set of processor time units on one or more processor time axes. The number of processor time units between 0 and the processor time unit that is mapped to by the first unit in a process execution is called the *start time* of that process execution. The number of time units between 0 and the time unit subsequent to the processor time unit mapped to by the last unit in a process execution is called the *completion time* of that process execution. A *feasible schedule* is a schedule in which the start time of every process execution is greater than or equal to that process execution's release time or request time, and its completion time is less than or equal to that process execution's deadline.

2.4. Process Segments

Each process p may consist of a finite sequence of *segments*[1] $p[0], p[1], ..., p[n[p]]$, where $p[0]$ is the first segment and $p[n[p]]$ is the last segment in process p. Given the release time r_p, deadline d_p of process p and the computation time of each segment $p[i]$ in process p, one can easily compute the release time and deadline for each segment (Xu and Parnas, 1993).

2.5. Precedence and Exclusion Relations

Various types of relations such as *precedence* relations and *exclusion* relations may exist between ordered pairs of processes segments. A process segment i is said to *precede* another process segment j if j can only start execution after i has completed its computation. Precedence relations may exist between process segments when some process segments require information that is produced by other process segments. A process segment i is said to *exclude* another process segment j if no execution of j can occur between the time that i starts its computation and the time that i completes its computation. Exclusion relations may exist between process segments when some process segments must prevent simultaneous access to shared resources such as data and I/O devices by other process segments.

3. Introduction to Priority Scheduling and Pre-Run-Time Scheduling

In the following, brief descriptions of the priority scheduling and pre-run-time scheduling approaches are provided.

3.1. The Priority Scheduling Approach

Most priority scheduling approaches share two basic assumptions: *a) the scheduling is performed at run-time; b) processes are assigned fixed priorities and whenever two processes compete for a processor, the process with the higher priority wins.*

Rate Monotonic Scheduling (Liu and Layland, 1973) is now the best known representive of the priority scheduling approach. It assumes that all processes are periodic, and that the major characteristics of the processes are known before run-time, that is, the worst case execution times and periods are known in advance. Fixed priorities are assigned to processes according to their periods, — the shorter the period, the higher the priority. At any time, the process with the highest priority among all processes ready to run, is assigned the processor. A *schedulability analysis* is performed before run-time based on the known process characteristics. If certain equations are satisfied, the actual scheduling is performed during run-time, and it can be assumed that all the deadlines will be met at run-time.

The Priority Ceiling Protocol (Sha, Rajkumar, and Lehoczky, 1990), makes the same assumptions as Rate Monotonic Scheduling, except that in addition, processes may have critical sections guarded by semaphores, and a protocol is provided for handling them. Each semaphore is assigned a *priority ceiling*, which is equal to the priority of the highest priority process that *may* use this semaphore. The process that has the highest priority among the processes ready to run, is assigned the processor. Before any process p enters its critical section, it must first obtain the lock on the semaphore S guarding the critical section. If the priority of process p is not higher than the priority ceiling of the semaphore with the highest priority ceiling of all semaphores currently locked by processes other than p, then process p will be blocked and the lock on S denied. When a process p blocks higher priority processes, p *inherits* the highest priority of the processes blocked by p. When p exits a critical section, it resumes the priority it had at the point of entry into the critical section. A process p, when it does not attempt to enter a critical section, can preeempt another process p_2 if its priority is higher than the priority, inherited or assigned, at which process p_1 is executing.

A set of n periodic processes using the Priority Ceiling Protocol can be scheduled (i.e., all deadlines will be met) by Rate-Monotonic Scheduling (the shorter the period, the higher the priority) if the following conditions are satisfied:

$\forall i, 1 \leq i \leq n : \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_i}{T_i} + Bi/Ti \leq i(2^{\frac{1}{i}} - 1)$

where C_i is the execution time, T_i is the period, B_i is the worst case blocking time of p_i due to any lower priority process (Sha, Rajkumar, and Lehoczky, 1990).

3.2. Pre-Run-Time Scheduling Approaches

With pre-run-time scheduling, the schedule for processes is computed off-line; this approach requires that the major characteristics of the processes in the system be known in advance. It is possible to use pre-run-time scheduling to schedule periodic processes. This consists of computing off-line a schedule for the entire set of periodic processes occurring within a time period that is equal to the least common multiple of the periods of the given set of processes, then at run-time executing the periodic processes in accordance with the previously computed schedule (Xu and Parnas, 1993).

In pre-run-time scheduling, several alternative schedules may be computed off-line for a given time period; each such schedule corresponding to a different "mode" of operation. A small run-time scheduler can be used to select among the alternative schedules in response to external or internal events. This run-time scheduler can also be used to allocate resources for a small number of asynchronous processes that have very short deadlines (Xu and Lam, 1998).

It is possible to translate an asynchronous process into an equivalent periodic process, if the minimum time between two consecutive requests is known in advance, and the deadline is not very short. Thus it is also possible to schedule such asynchronous processes using pre-run-time scheduling (Mok, 1984, Xu and Lam, 1998).

4. A Comparison of Pre-Run-Time Scheduling and Priority Scheduling

Below, the drawbacks of the priority scheduling approach will be briefly described and illustrated. We show how the priority scheduling approach, either Rate Montonic Scheduling or the Priority Ceiling Protocol, may fail to provide a feasible solution to a problem that could otherwise be solved with an optimal algorithm[2], using the pre-run-time scheduling approach. These drawbacks are direct consequences of the basic assumptions inherent in fixed priority scheduling, and not just peculiarities of one particular algorithm or formula based on the approach. Consequently, these drawbacks are not amenable to some ingenious quick fix.

4.1. It is Difficult to Use the Priority Scheduling Approach to Handle Complex Application Constraints

The schedulability analysis given in (Sha, Rajkumar, and Lehoczky, 1990) assumed that all tasks are independent tasks, that there are no precedence relations, that their release times are equal to the beginning of their periods. It is difficult to extend the schedulability analysis for priority scheduling to take into account application constraints that frequently exist in real-time applications, such as precedence constraints, release times that are not equal to the beginning of their periods, low jitter requirements[3], etc.

There are two main reasons for this difficulty:

(a) Additional application constraints are likely to conflict with the priorities that are assigned to processes. It is not generally possible to map the many different execution

orderings of processes that are required by the different application constraints in a large complex system onto a fixed hierarchy of priorities.

Attempts to reconcile additional application constraints with process priorities can result in subtle errors. For example, some authors have suggested using process priorities to enforce precedence constraints. But the priority assignment needed for the precedence constraints may conflict with a priority assignment determined by other criteria, e.g., the lengths of the periods of the processes. Even if there is no conflict, using priorities alone is inadequate for controlling the execution order of processes. Suppose that it is required that process A precede process B. If B happens to arrive and request execution before A, and B is the only process to request execution at that time, then B will be executed before A, even if A had been assigned a priority that is higher than the priority of B.

In contrast, with a pre-run-time scheduling approach, the schedule is computed before run-time, and it is relatively easy to take into account many kinds of additional constraints, such as arbitrary release times and deadlines, and precedence constraints, and it is possible to avoid using sophisticated run-time synchronization mechanisms by directly defining precedence relations and exclusion relations (Belpaire and Wilmotte, 1973) on pairs of process segments to achieve process synchronization and prevent simultaneous access to shared resources (Xu and Parnas, 1990, Xu and Parnas, 1992, Xu and Parnas, 1993, Xu, 1993). For example, the precedence relation A precedes B, can be enforced simply by ordering A before B in the pre-run-time schedule, thus guaranteeing that A will always be executed before B.

(b) Additional application constraints increase the computational complexity of scheduling problems, which already have high computational complexity whenever processes contain critical sections. Significant amounts of time are required by the scheduler to find good solutions for problems with high computational complexity. With the priority scheduling approach, scheduling is performed at run-time, and the scheduler does not have the time necessary to find good solutions.

In contrast, *with pre-run-time scheduling, schedules are computed off-line and time efficiency of the scheduling algorithm is not a critical concern. thus one is free to use optimal scheduling algorithms or any algorithm* (Xu and Parnas, 1993), *to schedule processes,* which would in most cases provide a better chance[4], of satisfying all the timing and resource constraints.

Because of the inherent constraints built into the fixed priority scheduling model, (e.g. fixed priorities) and because scheduling is performed at run-time, attempts to take into account additional constraints typically result in suggestions that either are only applicable to a few very special cases, or make drastically simplifying assumptions, which significantly reduce schedulability, or are extremely complicated, making the run-time behavior of the system very difficult to analyze and predict.

4.2. In General, the Priority Scheduling Approach Achieves Lower Processor Utilization Than the Pre-Run-Time Scheduling Approach

4.2.1. Scheduling strategies that are based on priorities, are heuristics that have less chance of finding a feasible schedule than optimal pre-run-time scheduling algorithms.

(a) Even if all processes are completely preemptable — an unlikely situation in a complex hard real-time system, scheduling processes according to priorities, is still not optimal. In Example 1 below, the Rate Monotonic Scheduling algorithm fails to meet the deadlines of two completely preemptable processes A, and B, but these two processes could be successfully scheduled by an algorithm that is optimal for scheduling completely preemptable processes, such as the earliest-deadline-first (EDF) algorithm (Liu and Layland, 1973).

Example 1. Process *A*: release time $r_A = 0$; computation time $c_A = 3$; deadline $d_A = 6$; period $prd_A = 6$; $priority_A = 0$.
Process *B*: release time $r_B = 0$; computation time $c_B = 4$; deadline $d_B = 8$; period $prd_B = 8$; $priority_B = 1$.

Schedule generated by the Rate Monotonic Scheduling algorithm: *B* misses its first deadline.

```
| A | B | A | B | A | B | A | B |
0   3   6   9   12  15  18  21  24
```

Schedule generated by an optimal algorithm, such as EDF: all deadlines are met.

```
| A | B  | A |B | A |B  | A | B |
0   3    7   10 12  15  18  21  24
```

□

(b) When some processes are not preemptable, scheduling processes according to priorities is even less likely to meet deadlines. For example, in such cases there are situations where, in order to satisfy all given timing constraints, it is necessary to let the processor be idle for an interval, even though there are processes that are ready for execution. An example is shown below:

Example 2. Process *A*: release time $r_A = 0$; computation time $c_A = 10$; deadline $d_A = 12$.
Process *B*: release time $r_B = 1$; computation time $c_B = 1$; deadline $d_B = 2$. *B* is not allowed to preempt *A*.

A priority scheduling algorithm will give the following schedule, in which *B* misses its deadline:

```
|          A       |B|
0                  10 11
```

In contrast an optimal algorithm, such as the algorithm in (Xu and Parnas, 1990), would be able to find the following feasible schedule in which all processes meet their deadlines:

```
|B|          A        |
1 2                   12
```

The timing constraints require that the processor must be left idle between time 0 and 1, even though *A*'s release time is 0; if *A* starts it would cause *B* to miss its deadline.
□

In most real situations, there will be process segments that cannot be preempted because a critical section has been entered. Thus, this example illustrates a very common situation. Priority-driven schemes are not capable of dealing properly with such situations.

The priority scheduling approach has a much smaller chance of satisfying timing constraints, because priority-driven schemes are only capable of producing a very limited subset of the possible schedules for a given set of processes. This severely restricts the capability of priority-driven schemes to satisfy timing and resource sharing constraints at run-time.

In general, the smaller the set of schedules that can be produced by a scheduling algorithm, the smaller the chances are of finding a feasible schedule, and, the lower the level of processor utilization that can be achieved by that algorithm. For example, if one insists on using a priority-driven scheme to schedule the set of processes given in Example 2 given above, then one would have to increase the processor capacity by a ratio of $(c_A + c_B)/d_B = 11/2$, and would achieve a much lower processor utilization than with an optimal scheduling scheme capable of producing the feasible schedule shown in Example 2. By using optimal algorithms that compute the schedule off-line, it is possible to achieve higher levels of resource utilization than those achievable by priority-driven schemes. Hence, using priority-driven schemes may increase the cost of a system to non-competitive levels.

4.2.2. When processes are scheduled at run-time, the scheduling strategy must avoid deadlocks. In general, deadlock avoidance at run-time requires that the run-time synchronization mechanism be conservative, resulting in situations where a process is blocked by the run-time synchronization mechanism, even though it could have proceeded without causing deadlock. When combined with the use of priorities, this reduces further the level of processor utilization. In Example 3, an example is shown where the Priority Ceiling Protocol (Sha, Rajkumar, and Lehoczky, 1990) blocks a process from executing and causing it to miss its deadline, even though that process could have proceeded and met its deadline without actually causing a deadlock.

Example 3. Process A: becomes ready at t_1, the whole process is one critical section guarded by semaphore s_0; computation time $c_A = t_2 - t_1$; $priority_A = 0$.
Process B: becomes ready at t_3, the whole process is one critical section guarded by semaphore s_1; computation time $c_B = t_5 - t_4$; deadline is $t_3 + c_B$; $priority_B = 1$;
Process C: becomes ready at t_2; the whole process is one critical section guarded by semaphore s_0; computation time $c_C = t_4 - t_2$; deadline is t_5; $priority_C = 2$.
Priority Ceiling of s_0 is 0; Priority Ceiling of s_1 is 1.

According to the Priority Ceiling Protocol, because both A and C use critical sections guarded by the same semaphore s_0, when C enters its critical section at time t_2, it executes at priority 0, and blocks B at time t_3, causing B to miss its deadline:

```
|A |        C           |B |
t1 t2 t3                t4 t5
```

But B could have preempted C and met its deadline without actually causing a deadlock:

```
|A |C | B |    C           |
t1 t2 t3                    t5
```

□

In the example above, the extremely conservative strategy of requiring a process to execute a critical section at the highest priority of any process that has a critical section guarded by the same semaphore, is necessary in order to avoid potential deadlocks when processes are scheduled at run-time.

In contrast, when a pre-run-time scheduling approach is used, the scheduling algorithm can look at all possible ways of constructing a deadlock-free, feasible schedule.

Note that the ratio:

$$\frac{processor\ utilization\ of\ the\ Priority\ Ceiling\ Protocol}{processor\ utilization\ of\ an\ optimal\ scheduling\ algorithm}$$

can be arbitrarily low! (In Example 3, this ratio can be made arbitrarily low by increasing ϵ.)

4.2.3. When using the priority scheduling approach, the amount of system overhead, that is, the amount of time spent on activities that are not part of the real-time application, is much greater than when a pre-run-time scheduling approach is used.

(a) The priority scheduler needs run-time resources to compute the schedule, that is, calculate at which moment which process should be executed. With a pre-run-time scheduling approach, the schedule is determined in advance.

(b) Since the priority scheduler does not know the schedule before run-time, it has to assume the worst case and save/restore complete contexts each time a process is preempted by another process. With a pre-run-time scheduling approach, one can determine in advance the minimum amount of information that needs to be saved and restored, and thus significantly reduce the time required for context switching.

(c) The priority scheduler also consumes significant run-time resources in order to perform various process management functions, such as suspending and activating processes, manipulating process queues, etc (Burns, Tindell, and Wellings, 1995). In comparison, with a pre-run-time scheduling approach, automatic code optimization is possible; one can switch processor execution from one process to another process through very simple mechanisms such as procedure calls, or simply by catenating code when no context needs to be saved or restored, which greatly reduces the amount of run-time overhead.

When the process periods are relatively prime, the Least Common Multiple (LCM) of the process periods and the length of the pre-run-time schedule may become inconveniently long. However, in practice, one can adjust the period lengths in order to obtain a satisfactory length of the LCM of the process periods. While this may result in some reduction in the processor utilization, the reduction should be insignificant when compared to the decrease in processor utilization with priority scheduling.

With the priority scheduling approach, when there are many different process periods, many priority levels are normally required; this would greatly increase the run-time scheduling overhead. Compromises that reduce the number of priority levels, by assigning identical priorities to subsets of processes with different periods, have the effect of increasing the response times of processes, and further reducing both schedulability and processor utilization (Sha, Klein, and Goodenough, 1991).

4.3. The Run-Time Behavior of the System is Much More Difficult to Analyze and Predict With the Priority Scheduling Approach Than With the Pre-Run-Time Scheduling Approach

The priority scheduling approach requires the use of complex run-time mechanisms in order to achieve process synchronization and prevent simultaneous access to shared resources. The run-time behavior of the scheduler can be very difficult to analyze and predict accurately. For example, in one study fixed priority scheduling was implemented using priority queues, where tasks were moved between queues by a scheduler that was ran at regular intervals by a timer interrupt. It has been observed that, because the clock interrupt handler had a priority greater than any application task, even a high priority task could suffer long delays while lower priority tasks were moved from one queue to another. Accurately predicting the scheduler overhead proved to be a very complicated task, and the estimated scheduler overhead was substantial, even though it was assumed that the system had a total of only 20 tasks, tasks do not have critical sections, and priorities do not change (Burns, Tindell, and Wellings, 1995).

In contrast, with a pre-run-time scheduling approach, it is possible to avoid using sophisticated run-time synchronization mechanisms by directly defining precedence relations and exclusion relations on pairs of process segments to achieve process synchronization and prevent simultaneous access to shared resources (Xu and Parnas, 1993). As mentioned above, one can switch processor execution from one process to another process through very simple mechanisms such as procedure calls, or simply by catenating code when no context needs to be saved or restored, which not only greatly reduces the amount of run-time overhead, but also makes it much easier to analyze and predict the run-time behavior of the system. Compared with the complex schedulability analysis required when run-time synchronization mechanisms are used, it is straightforward to verify that all processes will meet their deadlines in an off-line computed schedule.

It was reported that during the July, 1997 Mars Pathfinder mission, the spacecraft experienced repeated total system resets, resulting in losses of data. The problem was reported to be caused by "priority inversion" when the priority inheritence mechanism was turned off in the VxWorks real-time systems kernel that used priority scheduling. It is noted that such problems would never have happened if a pre-run-time scheduler had been used by the Mars Pathfinder (Wilner, 1997).

4.4. The Priority Scheduling Approach Provides Less Flexibility in Designing and Modifying the System to Meet Changing Application Requirements Compared With the Pre-Run-Time Scheduling Approach

The priority scheduling approach provides less flexibility than the pre-run-time scheduling approach, because the execution orderings of processes are constrained by the rigid hierarchy of priorities that are imposed on processes, whereas with the pre-run-time scheduling approach, there is no such constraint — the system designer can switch from any pre-run-time schedule to any other pre-run-time schedule in any stage of the software's development. Here are a few examples.

(a) It has been frequently claimed that the priority scheduling approach has superior "stability" compared with other approaches, because "essential" processes can be assigned high priorities in order to ensure that they meet their deadlines in transient system overload situations (Locke, 1992). The Rate Monotonic Scheduling approach assigns higher priorities to processes with shorter periods, because it has been proved that, if processes with longer periods are assigned higher priorities, then the schedulability of the whole system will be severely reduced. However, essential processes may not have short periods. While suggestions like cutting essential processes into smaller processes that are treated as processes with short periods have been made[5], these suggestions not only increase run-time overhead, but also add new artificial constraints to the problem, which increase the complexity and reduce the schedulability of the whole system. In real-time applications, under different circumstances, different sequences of process execution may be required, and sometimes different sets of processes become "essential." This is a problem which cannot easily be solved by assigning a rigid hierarchy of priorities to processes.

A pre-run-time scheduling approach can guarantee essential processes just as well, or better, than a priority scheduling approach. When using the pre-run-time scheduling approach, in the case of system overload, an alternative pre-run-time schedule which only includes the set of processes that are considered to be essential under the particular circumstances can be executed. As pre-run-time schedules can be carefully designed before run-time, the designer has the flexibility to take into account many different possible scenarios in overload situations, and tailor different strategies in alternative schedules to deal with each of them.

(b) A frequently mentioned example of the "flexibility" of the priority scheduling approach, is the fact that there exists a schedulability analysis for the priority scheduling approach that is based only on knowledge of the total processor utilization of the task set. This is claimed to provide more flexibility because "determining the schedulability of a system when an additional task is added requires recomputing only the total schedulability bound and determining whether the new utilization caused by the additional functionality causes the new bound to be exceeded (Locke, 1992)."

What is perhaps less well known about processor utilization based schedulability analyses, is the fact that the use of such analyses may cause the system designer to under-utilize the system's capacity. Processor utilization based analyses are invariably pessimistic; they give sufficient but not necessary conditions for schedulability. In other words, if one were to rely on the schedulability analysis, one may be forced to conclude that the fixed priority scheduling algorithm cannot be used, and take measures that further reduce the processor utilization in order to meet the processor utilization conditions provided by the schedulability analysis, even in simple cases where the fixed priority scheduling algorithm may have been able to schedule the processes under the original conditions.

Example 4.

Suppose that the system consists of 20 processes $p_1, p_2, ..., p_{20}$ all having an execution time of 1, and a period of 28.

The Rate Monotonic Scheduling schedulability analysis would not be able to guarantee that this set of processes is schedulable, because the total processor utilization is 20*(1/28) = 0.71 which is greater than the processor utilization limit of $20 * (2^{\frac{1}{20}} - 1)$

required by the schedulability analysis (Liu and Layland, 1973).

In this case, this set of processes would be rejected as the schedulability analysis would not be able to guarantee that they are schedulable, even though actually the Rate Montonic Scheduling algorithm would have been able to meet their deadlines:

```
|p1 |p2 | ... |p20|   | ... |   |
0   1   2     19  20  21    27  28
```

□

Note that when all processes are completely preemptable, the processor utilization based schedulability analysis of the Priority Ceiling Protocol is identical to that of Rate Monotonic Scheduling. Thus this example applies equally well to the Priority Ceiling Protocol.

Worst-case response time schedulability analyses for static priority scheduling, e.g., (Burns, Tindell, and Wellings, 1995), provide slightly more relaxed conditions for schedulability, compared with processor utilization based schedulability analyses. However, worst-case response time schedulability analyses for static priority scheduling share the same fundamental weakness with processor utilization based schedulability analyses. That is, they also only give sufficient but not necessary conditions for schedulability. Although worst-case response time schedulability analyses for static priority scheduling can guarantee the schedulability of the very simple set of processes in Example 4, they would reject the sets of processes in the examples shown earlier as not schedulable, even though those sets of processes can be scheduled when a pre-run-time scheduling approach is used.

It has also been claimed that, with a pre-run-time scheduling approach, it is more difficult to handle asynchronous processes when compared with using priority scheduling schemes. In the paper by Locke (Locke, 1992), where a particularly rigid version of a pre-run-time scheduling approach, the cyclic executive, was applied to an example problem, in order to show the difficulties in applying the cyclic executive, the author did not illustrate how the fixed priority executive could solve the same example scheduling problem. The fixed priority executive would have an equal or even greater difficulty in handling that same example problem. In another paper (Audsley, Tindell, and Burns, 1993) which attempts to show that a priority scheduling approach can solve an example problem that was given (Xu and Parnas, 1990), the example problem parameters were changed. If the original problem parameters in our paper are used, the proposed solution fails. In addition, the proposed solution used offsets but no algorithm to systematically compute those offsets was given.

In the following, we will provide an example in which the reverse is true. This example shows that, with a pre-run-time scheduling approach, once the pre-run-time schedule has been determined for all the periodic processes, the run-time scheduler can use this knowledge to achieve higher schedulability by scheduling asynchronous processes more efficiently, e.g., it would be possible to completely avoid blocking of a periodic process with a shorter deadline by an asynchronous process with a longer deadline.

Example 5.

Asynchronous process A: computation time $c_A = 3$; deadline $d_A = 15$; minimum time between two consecutive requests $min_A = 15$.
Process B: release time $r_B = 0$; computation time $c_B = 3$; deadline $d_B = 3$; period $prd_B = 8$; B is not allowed to preempt A.

Priority scheduling schemes are not able to guarantee that *A* and *B* will always be able to meet their deadlines. *B* misses its deadline, no matter what priority scheduling scheme is used.

Suppose that *A* arrives at time 7. A priority scheduling scheme will put *A* into execution immediately at time 7, thus *A* will block the second instance of *B* causing *B* to miss its deadline.

```
|      B      |                |      A      |      B      |
0             3                7             10            13
```

When a priority scheduling scheme is used, no matter whether the schedulability analysis is based on the total processor utilization, or the worst-case response time, the processes *A* and *B* will be rejected as unschedulable.

In contrast, when using a pre-run-time scheduling approach, there are at least two possible ways of scheduling the above processes.
(a) Convert asynchronous process *A* into a new periodic process *newA* (Mok, 1984, Xu and Parnas, 1993, Xu and Lam, 1998): release time $r_{newA} = 0$; computation time $c_{newA} = 3$; deadline $d_{newA} = 8$; period $prd_{newA} = 8$. Then the following pre-run-time schedule can be constructed using the algorithm in (Xu and Parnas, 1990), in which periodic process *newA* serves asynchronous process *A* in a manner similar to polling, will guarantee that both *A* and *B*'s deadlines will never be missed.

```
|      B      |    newA     |         |
0             3             6         8
```

(b) Another possible solution is to use an approach described in (Xu and Lam, 1998), in which a pre-run-time schedule is constructed for all the periodic proceses using the algorithm in (Xu and Parnas, 1990). In this case the following pre-run-time schedule will be constructed for *B*.

```
|      B      |                        |
0             3                        8
```

In this approach a run-time scheduler can use the information about the pre-run-time schedule to schedule asynchronous processes. In this case, *A* can be scheduled in the following way: at any time *t*, whenever there is a possibility that the immediate execution of *A* will cause *B* to miss its deadline, that is, whenever the end of the time slot reserved for *B* in the pre-run-time schedule minus *t* is less than the sum of the computation times of *A* and *B*; or whenever a periodic process with a deadline that is smaller or equal to *A*'s deadline, in this example *B*, is currently executing, then delay *A* until *B* is completed. By using the information in the pre-run-time schedule, one should be able to construct a table of "safe start time intervals" for asynchronous processes. For *A*, the safe start time interval is [3, 5], and the deadlines of *A* and *B* can be guaranteed by only allowing *A* to start its execution within that interval in each repetition of the pre-run-time schedule at run-time. The worst-case response time for each asynchronous process can be computed before run-time using the information about the pre-run-time schedule. In this case, *A*'s worst-case response time happens when *A* arrives at time 6, and is delayed until *B* has completed, and *A*

completes its execution at time 14. So A's worst-case response time is 14 - 6 = 8, which is less than its original deadline $d_A = 15$. If B is always executed within its reserved time slot in the pre-run-time schedule, then both A and B will be guaranteed to always meet their deadlines.
□

If one compares the potential run-time overhead of using a method similar to (b) in the example above that integrates run-time scheduling of a subset of the asynchronous processes with pre-run-time scheduling of periodic processes; with the overhead of priority scheduling schemes that schedule all the tasks at run-time:

(1) With the integration approach, the number of processes that the asynchronous process scheduler needs to handle, should be very small. This is because, in most real-time systems, the bulk of the computation is performed by periodic processes, while the number of asynchronous processes with hard deadlines is usually very small (Xu and Parnas, 1993). In addition a significant portion of the asynchronous processes can be transformed into periodic processes when using this approach. (One can determine which asynchronous processes should be converted into new periodic processes, and which should remain asynchronous, based on the reserved processor capacity required by each method (Xu and Lam, 1998).)

(2) A significant portion of the parameters used by the asynchronous process scheduler to make scheduling decisions, are known before run-time, so one can pre-compute major portions of the conditions that are used for decision making, hence the amount of computation that needs to be performed for scheduling purposes at run-time can be minimized. For example, one may create before run-time, a table of "safe start time intervals" for asynchronous processes, similar to the interval [3, 5] for process A in Example 5 (b) above, and substantially reduce run-time overhead by allowing the asynchronous processes to be scheduled by simple table lookup.

(3) Most of the important scheduling decisions have already been made before run-time. In particular, the relative ordering of periodic processes that usually form the bulk of the computation in most real-time applications, was determined before run-time when the pre-run-time schedule was computed.

When the system designer uses the priority scheduling approach, no matter whether the schedulability analysis is based on the total processor utilization, or the worst-case response time, the designer has less, not more, flexibility in adding new functionality to the system, because the schedulability analysis is more likely to reject a new set of processes as unschedulable.

In contrast, real-time system designers using the pre-run-time scheduling approach have the freedom to use any optimal scheduling algorithm to construct new pre-run-time schedules that include new processes and add new functionality to the system. Performing modifications to the system online is also not difficult. One can easily insert code in pre-run-time schedules that, when activated by an external signal, will cause processor execution to switch from a previously designed pre-run-time schedule to a newly designed pre-run-time schedule during run-time. The system designer is not constrained to use a rigid hierarchy of process priorities, and has more flexibility in designing new pre-run-time schedules to meet changing requirements.

From the discussion above, one may observe that with the priority scheduling approach, the system designer has very few options at his/her disposal for meeting application requirements. Priorities are used for too many purposes in the priority scheduling approach. It has been suggested that higher priorities be assigned to processes with:

i) shorter periods;

ii) higher criticality;

iii) lower jitter requirements;

iv) precedence constraints, etc.

Consequently, the system designer is faced with the impossible task of trying to simultaneously satisfy many different application constraints with one rigid hierarchy of priorities.

5. Conclusions

This paper explains and illustrates the following facts.

(1) Compared with the pre-run-time scheduling approach, the priority scheduling approach:

— does not handle complex application constraints well;

— results in lower processor utilization;

— has much greater system overhead;

— makes it significantly more difficult to analyze and predict the run-time behavior of the system.

(2) In contrast, with a pre-run-time scheduling approach:

— it is much easier to verify that all the deadline and other constraints will be satisfied;

— one can use better algorithms that can take into account a great variety of application constraints and provide the best chance of finding a feasible schedule;

— the run-time overhead required for scheduling and context switching can be greatly reduced.

(3) The main reasons for these differences are:

— the execution orderings of processes are constrained by the rigid hierarchy of priorities when using the priority scheduling approach, thus priority schemes are only capable of producing a very limited subset of the possible schedules;

— schedules are computed off-line when using the pre-run-time scheduling approach, but computed on-line when using the priority scheduling approach.

One may combine both techniques and use a small run-time scheduler to schedule a small number of asynchronous processes when they have (a) long inter-arrival times, (b) short deadlines and computation times, or (c) deadlines that are long compared with most periodic processes. Although the pre-run-time scheduling approach has many advantages over the priority scheduling approach, there are situations where a ”pure” pre-run-time scheduling approach is not practical because the deadlines for rarely invoked asynchronous processes are very short. In such situations, translating an asynchronous process to a periodic process (using a worst-case approach) may require too much processor capacity. The strategies used in the Priority Ceiling Protocol for preventing ”priority inversion” and deadlocks can be adapted for scheduling any asynchronous processes that could not be translated into periodic processes. The run-time scheduling should be integrated with the pre-run-time scheduling so that one can take advantage of the known process characteristics. The known characteristics should be used to determine which asynchronous processes are to be scheduled at run-time, and to reserve processor capacity methodically for asynchronous processes so that they can interrupt other processes in time to meet their deadlines (Xu and Lam, 1998).

• As only a small number of asynchronous processes with hard deadlines are scheduled at run-time, the amount of run-time resources required for scheduling and context switching is minimized.

• As the majority of processes are scheduled before run-time, this allows one to use better algorithms to handle more complex constraints, and achieve higher schedulability.

• As the run-time scheduler is aware of the future start times of the processes in the pre-run-time schedule, the run-time scheduler can make more informed decisions at run-time. This will reduce system overhead and maximize processor utilization, and consequently increase the chances that one can guarantee satisfaction of all constraints before run-time.

Notes

1. Parallel computations can be represented by several processes, with various types of relations defined between individual segments belonging to different processes, and processes can be executed concurrently; thus requiring each process to be a sequence of segments does not pose any significant restrictions on the amount of parallelism that can be expressed.
2. When scheduling algorithms are discussed, a distinction between *optimal algorithms* and *heuristics* will be made. An optimal scheduling algorithm is capable of finding a feasible schedule for a given mathematical scheduling problem if one exists. In contrast, for a given mathematical scheduling problem, a heuristic may not find one even if one exists.
3. *Jitter* refers to the variation in time a computed result is output to the external environment from cycle to cycle.
4. If there exists no other algorithm that has a better chance of finding a feasible schedule than an algorithm that uses priorities — an unlikely situation, one may still use the pre-run-time scheduling approach in combination with an algorithm that uses priorities as a last resort to find a pre-run-time schedule. Priority scheduling approaches assume that scheduling is done at run-time, and do not consider pre-run-time schedules at all.
5. A similar suggestion is described in (Locke, 1992) that suggests assigning high priorities to processes with low-jitter requirements. The difficulty is similar: processes with low jitter requirements may not have short periods.

References

Audsley, N., Tindell, K. and Burns, A. 1993. The end of the line for static cyclic scheduling. *Proc. Fifth Euromicro Workshop on Real-Time Systems* 36-41.

Belpaire, G. and Wilmotte, J. P. 1973. A semantic approach to the theory of parallel processes. *Proc. 1973 European ACM Symposium* 217-222.

Burns, A., Tindell, K. and Wellings, A. 1995. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Software Eng.*, 21: 475–480.

Faulk, S. R. and Parnas, D. L. 1988. On synchronization in hard-real-time systems. *Communications of the ACM*, 31: 274–287.

Heninger, K., Kallander, J., Parnas, D. L. and Shore, J. 1978. Software Requirements for the A-7E Aircraft, NRL Report No. 3876, Nov.

Liu, C. L. and Layland, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM 20: 46–61.

Locke, C. D. 1992. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems* 4: 37–53.

Mok, A. K. 1984. The design of real-time programming systems based on process models. Proc. *1984 IEEE Real-Time Systems Symposium* 5–17.

Sha, L., Klein, M. H. and Goodenough, J. B. 1991. Rate Monotonic Analysis for Real-Time Systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, van Tilborg, A. M., and Koob, G. M., eds. Kluwer Academic Publishers, 129–155.

Sha, L., Rajkumar, R. and Lehoczky, J. P. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers* 39: 1175–1185.

Wilner, D. 1997. Keynote address, 1997 IEEE Real-Time Systems Symposium.

Xu, J. and Lam, K. 1998. Integrating run-time scheduling and pre-run-time scheduling of real-time processes. Proc. *23rd IFAC/IFIP Workshop on Real-Time Programming*, Shantou, China, June.

Xu, J. and Parnas, D. L. 1993. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Trans. on Software Engineering* 19: 1-17, Jan.

Xu, J. 1993. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering* 19, Feb.

Xu, J. and Parnas, D. L. 1992. Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections. Proc. *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC-92)*, Scottsdale, Arizona, April 1-3.

Xu, J. and Parnas, D. L. 1990. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering* 16: 360-369, Mar. Reprinted in *Advances in Real-Time Systems*, Stankovic, J. A., and Ramamrithan, K., eds. IEEE Computer Society Press, 1993, pp. 140-149.