Reconciling Real-Time and "Fair" Scheduling

TR88-024

May 1988

Bill O. Gallmeister

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Copyright ©1988 Bill O. Gallmeister

UNC is an Equal Opportunity/Affirmative Action Institution.

Reconciling Real-Time and "Fair" Scheduling

Bill O. Gallmeister

CB # 3175, Department of Computer Science [1]
University of North Carolina
Chapel Hill, NC 27599-3175

May 6, 1988

[1]This Research was supported by Office of Naval Research Contract N00014-86-K-0680.

Motivation

Multiprogramming operating systems which support both real-time and non-real-time processes are called "soft-real-time" systems. In such systems, process scheduling is problematic, because there are two *qualitatively* different classes of processes requiring scheduling. If the scheduling algorithm is designed to favor one class of processes, it will probably not treat the other class of processes fairly. For instance, priority-based schedulers are considered fair to non-real-time processes, but lead to unreliable real-time systems[3,2]. Conversely, deadline-based schedulers suffice to run real-time processes but will starve non-real-time processes, which have no deadline.

The common conception is that real-time responsiveness and fair multiprogramming are mutually exclusive. I hold that this concept is wrong, because our concept of "fairness" is wrong. I propose a unifying model of processes under which so-called "real-time" and "non-real-time" processes are merely different instances of the same sort of object, with different attribute values. Under this unifying scheme, all processes can be treated fairly while still preserving real-time responsiveness.

Who Needs It? Some applications do not require such a unified model of processes. For instance, hard-real-time tasks – tasks which must meet rigid time constraints – typically are not concerned with the fair treatment of processes, because the processes are carefully scheduled, as a whole, to meet the overall time constraints of the system. Fairness is not of concern because the subprocesses making up the task are not competing for the processor; instead, they are cooperating. To treat one part of that entity "fairly", at the expense of the rest of the entity, would make no sense. On the other hand, in soft-real-time systems, where real-time and non-real-time tasks compete for the processor, both fairness and responsiveness are of concern. In these machines, time-constrained tasks must run along with non-time-constrained tasks.

Systems and Schedulers

In this section, two different sorts of multiprogrammed systems – real-time and non-real-time systems – are discussed from the standpoint of their schedulers. Due to differences in objectives, the schedulers for real-time systems are completely different from the schedulers for non-real-time systems; therein lies the conflict between fairness and real-time.

1.1 Criteria for Judging Scheduling Algorithms

There are many different scheduling algorithms to meet many different requirements of the system being built. Criteria for selecting a scheduling algorithm include

- guaranteed response times
- behavior under overload
- fairness

Guaranteeing response time means that the scheduling algorithm assures that a process can be run within some specified time after it comes ready (the process deadline). This characteristic is necessary for a scheduling algorithm which can support real-time systems.

Behavior under overload refers to the algorithm's performance when the processor is overloaded – that is, too much to do in too short a time. Notice that this criterion is only important in the case of real-time systems, as well – non-real-time schedulers have no concept of deadlines, and so the idea of "overload" makes no sense in that context.

Fairness is the most widely-examined quality of non-real-time schedulers. "Fairness" is usually taken to mean that "all processes are treated the same, and no process can suffer indefinite postponement"[1]. As stated above, real-time schedulers are not usually concerned with being fair.

1.2 Non-Real-Time Systems

When a multiprogramming system has no obligation to meet real-time constraints, fairness becomes the primary criterion for judging the quality of the scheduler. If the scheduler treats every process equally, and assures that every process will run eventually, then that scheduler is called a *fair scheduler*.

Central to the notion of fairness is the idea that processes are *competing* for the computing resource. As we will see below, there are systems where processes do not compete for the processor, but instead cooperate using statically-determined rules. Such systems are not of concern to us now.

Priority-Based Schedulers. The predominant way of producing a fair scheduler is to attach priorities and quanta to each process. Processes are run in order of highest priority first, and given the processor for an amount of time not exceeding their quanta. Based on process performance (I/O- vs. CPU-boundedness), both priority and quantum can be adjusted dynamically.

Note that even under priority-based schemes, it can be argued that the scheduling algorithm is not fair, since higher priority processes are run sooner than lower-priority processes. This example points out the error in the definition of "fair": a fair scheduler does *not* treat every process identically. Instead, a fair scheduler only *examines* each process identically, and chooses which process to run by decidedly unfair means.

Why Do We Use an Unfair Scheduler? In fact, some systems actually do schedule all processes identically: Processes are placed on a queue in some random order, and each process is taken off the queue, run for a quantum, and reinserted at the end of the queue. This scheduler is fair, and is also simple to implement and understand. These are powerful advantages. However, treating all processes identically is not "the right thing" to do in most multiprogrammed systems, because certain processes *are* more important than others. For instance, in a typical UNIX[1] system, the "update" process, which ensures that filesystems remain consistent, is more important than most other processes, since without it, the system will probably crash due to filesystem damage. The ideal of treating all processes identically is a conceptually clean thought, but not a terribly useful strategy in real life.

1.3 Real-Time Systems

Real-time systems are programs which must be run within time constraints which are part of their specification. In such systems, timely execution of code is an essential component of the system – if a process completes late (or early, in some cases), then its computation will have reduced or no benefit, and may even be detrimental to the system.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Real-time systems can be classified by the stringency of their time requirements into hard- and soft-real-time systems.

- **Hard (dedicated) Real-Time Systems:**

  In "hard-real-time" systems, timing constraints exist throughout all aspects of the system – every process has some time constraint[4]. In such systems, processes which do not complete by the proper time have zero value. It is the hard-real-time scheduler's task to ensure that all processes in the system do complete by their deadlines. In such systems, timely operation is often needed to protect property or lives, and so the proper execution of the system is a "hard" requirement. Such systems are usually run on dedicated machines using statically-determined schedulers built at programming or compile time.

  Fairness in this context has no meaning for an individual process, as the processes are not in competition. There is only one task, composed, perhaps, of multiple processes, and it has complete use of the entire machine. The component processes cooperate for the processor rather than competing.

- **Soft (non-dedicated) Real-Time Systems:**

  Soft-real-time systems are those which run on non-dedicated, multiprogrammed machines. Parts of the system may not have deadlines, but should be run simply "as soon as possible". These other tasks are competing with the real-time task for machine resources (i.e. cpu cycles). In such systems, a central scheduler decides which task will run when. Since control of the machine resource is controlled from outside the real-time task, guaranteeing hard-real-time responsiveness is impossible in all but the most lightly-loaded machines. For this reason, hard-real-time tasks should not run on non-dedicated systems. This paper discusses scheduling for soft real-time systems only.

1.3.1 Deadline-Based Schedulers

To meet time requirements of real-time processes, a deadline scheduler or a variant of a deadline scheduler can be used. Like priority-based schedulers, deadline schedulers take one process at a time from a queue, run it for some quantum, and then reinsert it to the queue. However, in the case of the deadline scheduler the queue is sorted in increasing order of deadline; the processes with the closest deadlines are at the head of the queue and those with distant deadlines are at the rear of the queue. A variant is the slack-time scheduler, where the queue is sorted in increasing order of slack time. A process's slack time is the amount of time before it absolutely must run in order to meet its deadline (given by $(deadline - quantum - currenttime)$; see figure 1).

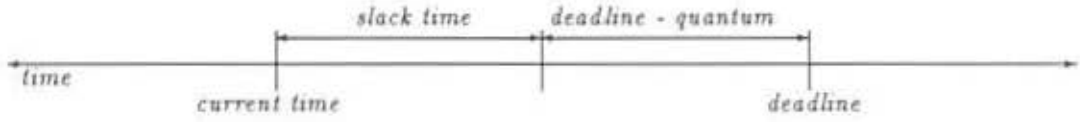


Figure 1. Definition of Slack Time.

1.4 The Conflict Between "Real-Time" and "Fair"

Because of the differences in their implementations and objectives, schedulers for hard-real-time systems are quite different from schedulers for non-real-time systems. Schedulers for soft-real-time systems fall in between the two extremes.

Typical real-time schedulers are evaluated with respect to overload behavior and whether deadline scheduling can be guaranteed, rather than with respect to fairness. In contrast, non-real-time schedulers do not attempt to provide any sort of deadline service, but do aim to provide "fair" scheduling behavior. Unfortunately, the intermediary soft-real-time schedulers must be evaluated with respect to both sets of desiderata, and on one count or another, they usually fail.

For instance, when typical UNIX systems are converted to support soft-real-time functionality, they usually do so by providing an elevated range of "real-time priorities". Essentially, real-time processes are given priority higher than all other processes. While this may help to guarantee response times (and guaranteeing response times is still a chancy business left to the programmer), it has the unwelcome side effect of being unfair to all other processes.

Such failures to satisfy both fairness and response time objectives has led many to assume that fairness and real-time responsiveness are mutually exclusive goals.

1.5 A Unified Model of Processes

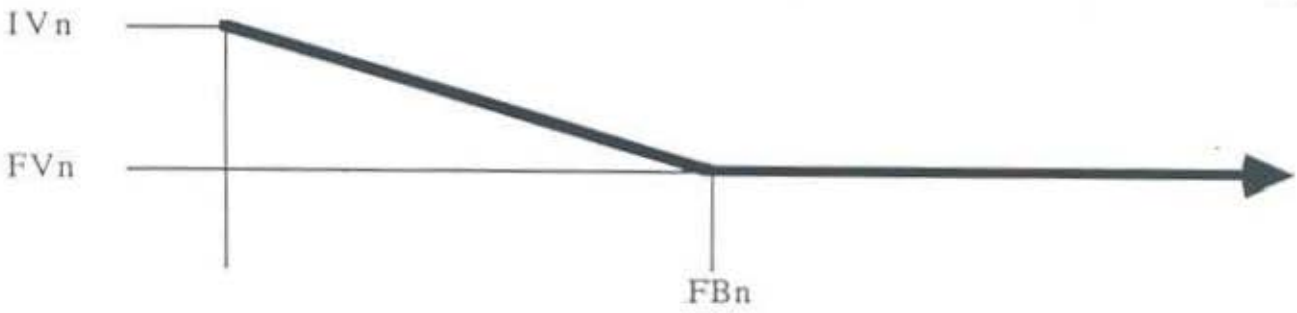
Real-time processes have been modeled before as having value functions which vary with time, and using such models has led to better scheduling performance under a variety of conditions[3]. If we wish to extend such a model to include non-real-time processes, we can consider them to be real-time processes with no deadlines. They, too, can be assigned values that vary with time. Exactly *how* such a function should behave is a question which has no exact answer. Each person probably has a different idea of how the value of a process varies with time. The particular function used corresponds to one of many different sorts of processes.

1.5.1 Non-Real-Time Process Values

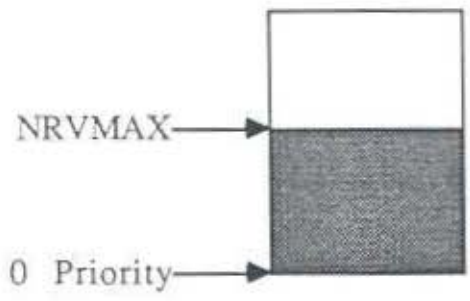
Under priority-based schemes, a process's value is modeled as a constant value, possibly modified based on process behavior and time spent waiting.

When we consider such processes, though, we don't think of their values as being constant. We would rather have the process run quickly; therefore, the process's value should be higher when it is first submitted to the system, dropping over time to some steady-state value (if the process value continues to drop, eventually it will have no value; this can lead to process starvation). Again, this basic scheme can be augmented by perks and penalties for process behavior and waiting time.

Under the unified scheme, non-real-time process values can be represented by a tuple (IV_r, FV_r, FB_r). When first readied, the non-real-time process has value IV_r. The value changes linearly with time to FV_r, which value it achieves at time FB_r. The value remains constant thereafter. Changes to the value may be made based upon process waiting time and behavior.



Range of Non-Real-Time Values. Non-real-time process values must lie in the range $[0..NRVMAX]$.

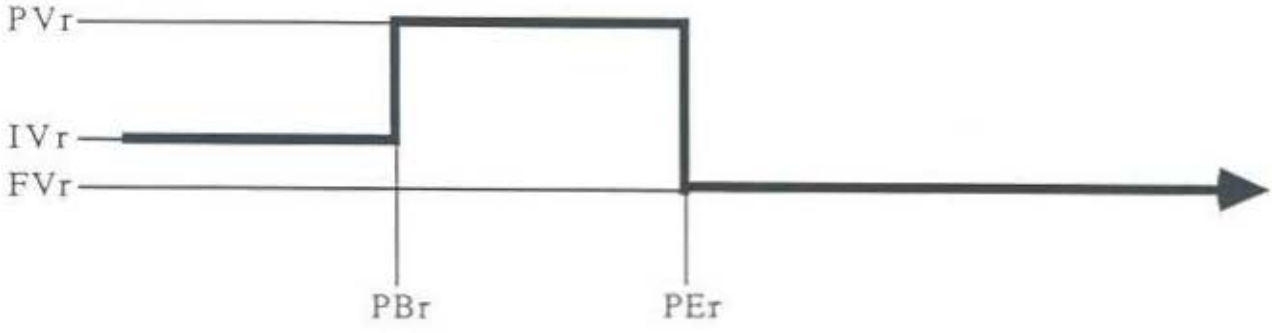


1.5.2 Real-Time Process Values

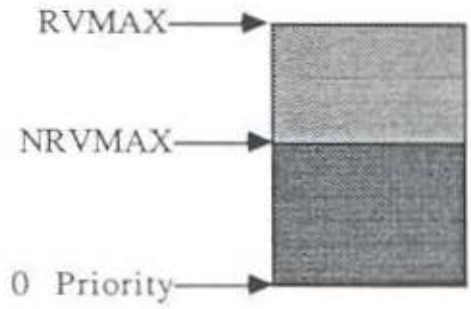
In deadline-based schedulers, a real-time process's value is represented by its deadline, and looks something like a step function.

In other real-time systems, processes may be constrained to run within a time *window* rather than just by a certain time:

Under the unified scheme, real-time process values are represented by a tuple $(IV_r, PV_r, FV_r, PB_r, PE_r)$. When first readied, the process has value IV_r. At time PB_r, the process value rises to PV_r. At time PE_r, the value drops down to value FV_r.



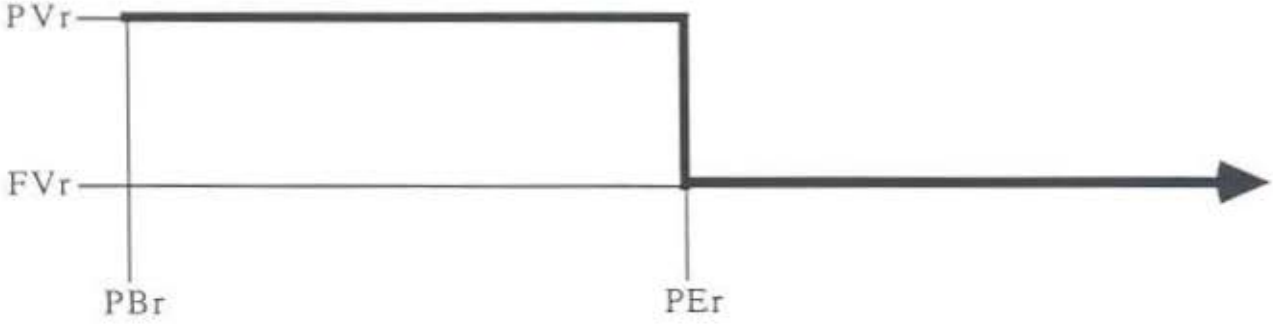
Range of Real-Time Values. Real-time process initial and final values must lie in the range $[0..NRVMAX]$. Real-time process peak values must lie on the range $[0..RVMAX]$, where $RVMAX > NRVMAX$.



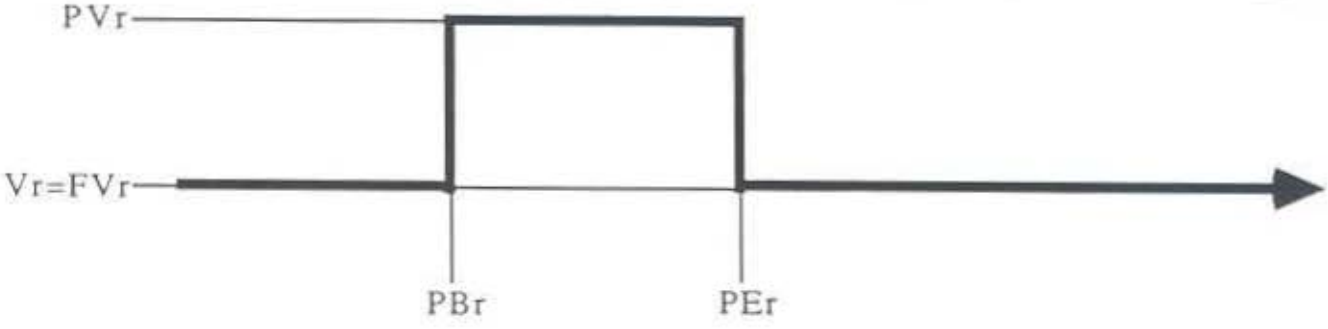
1.6 Advantages of the Unified Process Model

The unified process model allows us to represent a large number of different process value profiles. For instance, a non-real-time process can start out at a high value and drop to a lower one, or it can rise to a higher value over time. It can remain at a constant value for all time.

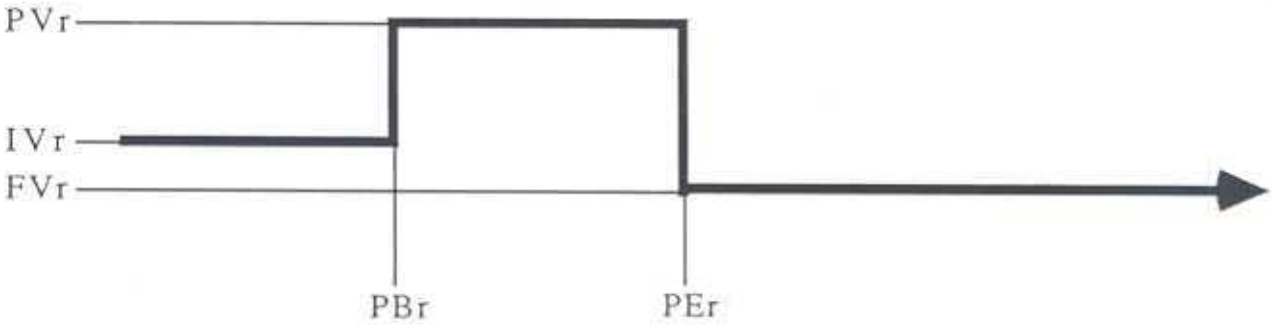
The flexibility in representing real-time processes is more pronounced. We can represent processes with single deadlines by setting PB_r and FV_r to zero.



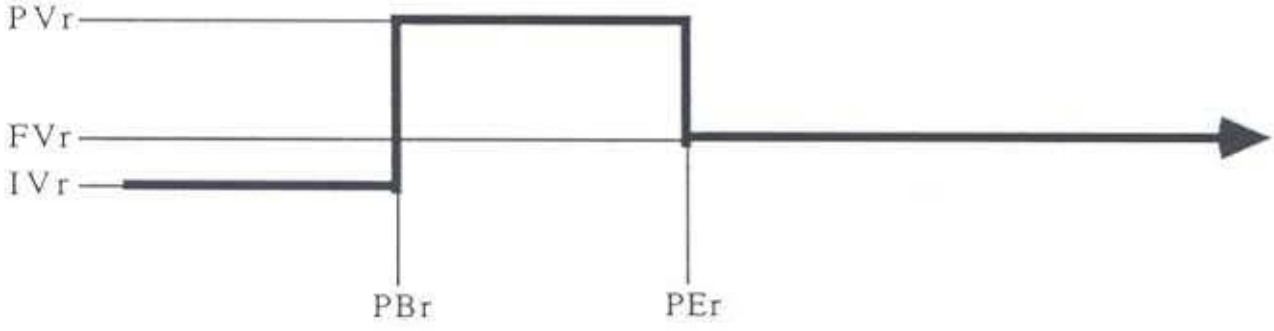
We can represent processes that must be run within a time window by making PB_r nonzero and FV_r and IV_r zero.



We can represent processes that should be run within a window but which can be run before the window by allowing IV_r to be nonzero.



By setting FV_r to a nonzero value, we can specify a process which should run within a window but can run after the window.



1.7 Fair Real-Time Scheduling in the Unified Process Model

Fair real-time scheduling can be achieved in this unified process model by simply running, at any given time, the process with the greatest value.

Real-Time Scheduling is Performed. Because real-time processes have peak values which can be greater than any non-real-time value, we are assured that real-time processes will be run within their windows unless there is an overload condition.

Fairness is Preserved. Because real-time processes have values outside their windows which are no greater than those of non-real-time processes, we will possibly run several non-real-time processes in the time interval before a real-time process assumes its peak value.

Note that processes are not examined exactly equally – the class of real-time processes is treated differently than the class of non-real-time processes. This is a result of the fact that the two classes of processes have different characteristics and requirements. It is the duty of the scheduling algorithm to ensure that the processes are all treated "separately but equally."

The performance bottleneck of the scheduling algorithm is in the decision process. Some clever data structures can be used to greatly speed the determination of "most valuable" process at any given time. First, we split processes into real-time and non-real-time processes; each set is stored on a separate priority queue which can be sorted especially for that class of processes. Efficient search and insertion in these priority queues then becomes the central problem, but efficient priority queues have been well-researched.

This general scheme is similar to [2]. In fact, a fair real-time scheduler can be built as an extension of the scheduler mentioned in that work[2]. The major drawback of such a system is the extensive scheduling overhead incurred due to the finer granularity of scheduling and the more complex model being used. The scheduler proposed herein is of a much coarser grain, and therefore the overhead involved should be much less.

The scheduler tends to schedule non-real-time tasks whenever it can, behaving in a "just-in-time" manner. This is in contrast to typical soft-real-time schedulers, which run real-time processes as soon as possible because of the elevated priorities these processes *always* run at. This scheduler will tend to schedule processes more tightly near their deadlines.

1.8 Problems With Scheduling Unified Processes

This complicated model of processes allows us to solve the problem of combining responsiveness and fairness. Nothing is free, though, and this system has drawbacks of its own. Most importantly, the more complex process value model makes for more complicated scheduling algorithms. Simplicity is lost in moving to this more realistic model. Along with the loss of simplicity we will suffer a gain in the amount of data and code required, and an attendant loss in speed. The exact tradeoffs involved will be more clear when the system is actually implemented.

Avoiding Starvation In order to avoid starvation of non-real-time processes, we need some scheme for ensuring the eventual running of any process. In priority-based schedulers, this is done by increasing a process's priority as time goes by, under the assumption that it will eventually be the highest priority process. However, the highest value a non-real-time process may assume is still less than the value a real-time process may have. If there is always some real-time process with a value in the range $[NRVMAX..RVMAX]$, then non-real-time processes can be starved. Note, though, that this is an overload condition; there is not enough processing bandwidth to support all the processes.

[2] This is one of those times when you get scooped – almost.

"Just-In-Time" Has Its Own Problems. Just-in-time scheduling has problems all its own, although it has been used to great advantage in manufacturing and shipping applications worldwide. The largest potential disadvantage is that, if a number of high-value real-time tasks with close deadlines enter the scheduler just before a real-time task is due to run, they may produce a transient overload which will cause the real-time process to miss its deadline. This is a problem endemic to this scheduler because it delays running real-time processes until the last minute. However, by stretching out the peak period of time $[PB_r..PE_r]$, each real-time task can specify some slack to account for system factors. The real-time application development system should support dynamic, run-time value function modifications to allow this sort of fine-tuning.

Summary

Soft-Real-Time systems require both real-time and non-real-time processes be scheduled. Scheduling in these systems is difficult due to qualitatively different requirements of the two classes of processes; schedulers which treat one class reasonably will usually do so at the expense of the other class. A unified model of processes, in which each process has an explicit value function which varies with time, allows us to construct a scheduler which can provide real-time scheduling and fair non-real-time scheduling at the same time.

Bibliography

[1] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[2] Scott Guthery. Self-Timing Programs and the Quantum Scheduler. *Real-Time Systems Newsletter*, 4(3):3–17, Spring 1988.

[3] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Real-Time Systems Symposium*, pages 112–122, December 1985.

[4] Aloysius K. Mok. The Design of Real-Time Programming Systems Based on Process Models. In *Real-Time Systems Symposium*, pages 5–17, December 1984.