



The United Nations
University

UNU/IIST

International Institute for
Software Technology

Duration Calculus Specification of Scheduling for Tasks with Shared Resources

Philip Chan and Dang Van Hung

June 20, 1995

UNU/IIST

UNU/IIST serves developing countries in attaining self-reliance in software technology: (i) own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry university advanced development projects in which rigorous techniques supported by semantics based tools are case-study applied to large scale software developments, (b) own and joint university and academy institute research in which new techniques for application domain and computing platform modeling, requirements capture, software engineering and programming are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach design calculi oriented software development techniques, (d) events [like panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as railways, manufacturing industries, health care systems, etc., and are thus aligned with UN and Intl. Aid System concerns. The research projects parallel and support the advanced development projects.

Technically speaking, the focus of UNU/IIST, in all of the above, at present, is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasizes techniques that permit proper development steps and interfaces. UNU/IIST endeavours to also promulgate sound project and product management principles.

UNU/IIST's dissemination strategy is to act as a clearing house for reports from primarily industrial country research and technology centres to primarily developing country industry, university and academy centres. At present more than 175 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Awareness of reports received (and produced) as well as of journal articles is to be disseminated regularly to developing country centres — who are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director

UNU/IIST Reports are either \mathcal{R} esearch, \mathcal{T} echnical, \mathcal{C} ompendia or \mathcal{A} dmistrative reports:

$\boxed{\mathcal{R}}$ Research Report • $\boxed{\mathcal{T}}$ Technical Report • $\boxed{\mathcal{C}}$ Compendium • $\boxed{\mathcal{A}}$ Administrative Report



The United Nations
University

UNU/IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macau

Duration Calculus Specification of Scheduling for Tasks with Shared Resources

Philip Chan and Dang Van Hung

Abstract

This paper presents a formalization in the duration calculus (DC) of scheduling policies for tasks with shared resources. Two frameworks are presented for specifying classes of schedulers. With these specifications, some properties of these schedulers were proved using the formal deduction of DC. This paper aims to encourage other researchers to formally treat real-time aspects of operating systems which in the past were conventionally a piece of ad hoc territory in computer science.

Philip Chan is a Fellow of UNU/IIST, on leave from the Department of Software Technology, College of Computer Studies, De La Salle University, Manila, Philippines, where he is an assistant professor. His current research interests include operating systems, distributed operating systems, and Duration Calculus. His e-mail address at UNU/IIST is pc@iist.unu.edu and at De La Salle University is ccspc@linux1.dlsu.edu.ph

Dang Van Hung is from the Institute of Information Technology of National Center for Natural Science and Technology of Vietnam, where he is a researcher. He is a Fellow of UNU/IIST from April 1994 to July 1995. His research interest is in Formal Technique of Programming, Real-Time System Specification, and Concurrent and Distributed Systems. Email: dvh@iist.unu.edu

Contents

1	Introduction	1
2	Duration Calculus	5
3	Formalization of Scheduling for Tasks with Shared Resources	8
3.1	Protocols that Permit Priority Inversion	14
3.2	Protocols that Prevent Priority Inversions	17
3.3	Requirements	18
4	Proving Properties	19
4.1	Blocked At Most Once Property of PCP	19
4.2	PCP is Deadlock Free	22
4.3	No Priority Inversion	23
5	Conclusions	24
A	Proof of Lemma 4.1	26

1 Introduction

Operating systems have been a subject of design and implementation for many years. Despite their widespread use in almost every computing environment, formal techniques have been applied only on relatively simple aspects of their dynamic behavior. This is mainly due to the fact that operating systems are complex and it is not a trivial task to formally describe properties of real-time operating systems and their components unless these are simplified.

Formally specifying and reasoning about properties of real-time systems requires the use of a notation with formal deduction capabilities. In this paper, we make use of the duration calculus (DC) because of its many desirable features that provide explicit support for modeling real-time behavior. DC has been applied in the context of operating systems, such as in [12, 16] and more recently, the work on proving formally the correctness of the *Deadline Driven Scheduler* [11]. This paper aims to be another step in this direction by focusing on more elaborate real-time scheduling systems.

Specifically, this paper is concerned with scheduling of tasks with shared resources. During execution, tasks may request-access to resources. We assume that resources are allocated on a mutually exclusive fashion, i.e. a resource can be held by at most one task at a time, and resource allocation is non-preemptive. Therefore, aside from satisfying the deadline requirements of tasks, scheduling policies have to deal with the potential for deadlocks and priority inversion.

Since resource allocation is nonpreemptive and access is mutually exclusive, it is possible for an undesirable condition called a deadlock to occur. The system is said to be in a deadlocked state when there is a set of tasks mutually waiting on some resource held by another task in the set. In short, deadlocks can occur if tasks are in a circular hold-and-wait situation. There are two basic approaches to handle deadlocks: (a) explicitly incorporate mechanisms into the protocol such that deadlocks are not possible; and (b) use of a deadlock detection / recovery procedure.

In the succeeding discussions, let T_i, T_j denote arbitrary tasks and let r_k denote any resource. A priority inversion occurs when a lower-priority task T_i is blocking the execution of a higher-priority task T_j . This occurs when T_i is holding a resource r_k that T_j needs and is preempted by the arrival of T_j . If T_j requests for r , T_j will be blocked since r is not available and cannot be preempted from T_i . Existing policies either ensure that the priority inversion is bounded or prevents it from happening at all [2, 3] by restricting the schedules.

Fig. 1.1 presents a Gantt chart that illustrates a priority inversion. Suppose that the priorities of the tasks T_i, T_j , and T_k are p_i, p_j , and p_k , respectively ($p_i < p_k < p_j$). Here are the sequence of events:

- $t = t_1$: task T_i arrives and executes;
- $t = t_2$: task T_i requests for resource r_k and is granted;
- $t = t_3$: task T_j arrives and preempts T_i since $p_j > p_i$;
- $t = t_4$: task T_j requests for resource r_k but is blocked since T_i is still holding r_k so T_i

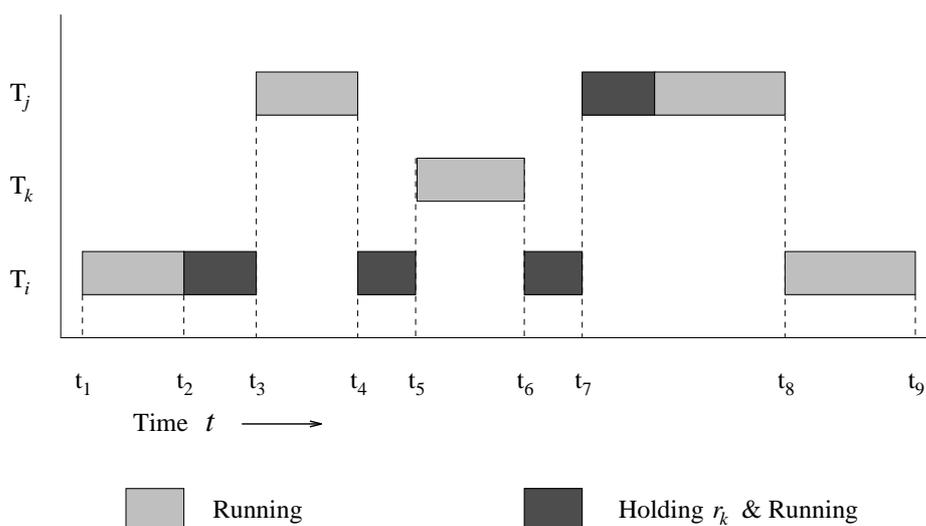


Figure 1.1: Example of Priority Inversion

resumes execution;

$t = t_5$: task T_k arrives and preempts T_i since $p_k > p_i$;

$t = t_6$: task T_k finishes and T_i resumes execution;

$t = t_7$: task T_i releases r_k and is preempted by T_j since it is now ready;

$t = t_8$: task T_j executes and finishes, T_i resume execution; and

$t = t_9$: task T_i finishes execution and leaves.

In this example the priority inversion occurs between times t_4 and t_7 . Note that it is possible for a steady arrival of medium-priority tasks after time t_4 to occur causing the priority inversion to become unbounded in length.

Our intention of using DC is threefold: (a) to show the expressive ability of DC in formally specifying more complex schedulers; (b) to present a framework for uniformly specifying schedulers within the same class; and (c) to use the specifications from these frameworks to reason about properties of these schedulers.

A comprehensive review of techniques for controlling access to shared resources is presented in [1]. In this paper, we are specifically interested in two kinds of protocols: (a) those that allow priority inversion to occur; and (b) those that schedule the use of resources in such a way so that no priority inversions occur. Representative protocols under both types will be considered in this paper.

First, let us consider protocols that permit priority inversions to occur. In [1], these protocols are also classified as *preemptive blocking protocols*, since a task T_i holding a resource r can be preempted by higher priority tasks and T_i can block other tasks that request for the resource r . Three representative protocols are considered:

- (a) the trivial protocol A;
- (b) the priority inheritance protocol; and
- (c) the priority ceiling protocol.

Trivial Protocol A

We present this protocol in this paper as a simple case and to show its relationship with the more complex protocols. It works as follows: If a task T_i requests for a resource r_k , it is granted if r_k is available. Otherwise, T_i is blocked on r_k . When r_k becomes available, the protocol selects the highest priority task among the tasks that are blocked on r_k to lock and use the r_k .

There is no doubt that this protocol suffers from deadlock and unbounded priority inversion.

Priority Inheritance Protocol (PIP)

To solve the unbounded priority inversion, the priority inheritance protocol was introduced in [9]. It assumes that:

- (a) the priority assigned to tasks is static;
- (b) resources are accessed in a mutually exclusive manner;
- (c) accesses of resources are properly nested;
- (d) a preemptive priority-driven scheduler is used; and
- (e) the resources that a process accesses can be predetermined before runtime.

In this protocol, priority inversion is bounded by temporarily increasing the priority of tasks that block higher priority tasks. The protocol works as follows: A task T_i that wants to acquire a resource r_k must obey the following rule:

- (a) if r_k is available, it is granted to T_i ;
- (b) otherwise (i.e. there is a task, say T_j , that is currently holding r_k), T_i is blocked on r_k and if $p_i > p_j$ then task T_j inherits p_i .

Although priority inversions are bounded, this protocol does not ensure freedom from deadlock. Furthermore, it is possible for chained blocking to occur. Chained blocking occurs when tasks can be blocked for more than the duration of one critical section.

Priority Ceiling Protocol (PCP)

The priority ceiling protocol [8, 9] was designed to address the deadlock and chaining problems of the priority inheritance protocol. This protocol works as follows:

- (a) each resource r_k is assigned a priority ceiling, $c(r_k)$, which is equal to the highest priority of all tasks that can lock it; and

- (b) if a task T_i requests for a resource r_k , it is granted if $p_i > c(r_l)$, for all resources r_l currently locked by tasks T_j other than T_i

Similar to the priority inheritance protocol, a task that blocks a higher priority task T_j inherits T_j 's priority for the duration of the current critical section.

Now, we present resource control protocols that prevent the occurrence of priority inversions. Two representatives under this class are considered:

- (a) trivial protocol B; and
 (b) the reservation protocol.

Both of these protocols assume a multiprocessor environment.

Trivial Protocol B

The protocol works simply as follows: if a task T_i wants to access a resource r_k , access is granted only if all other higher priority tasks T_j that will use r_k have already acquired and released r_k .

Obviously, this protocol can delay lower priority tasks even if the higher priority tasks have not arrived.

Reservation Protocols

Presented by [2, 3], reservation protocols solve the priority inversion problem without changing priorities of tasks. Furthermore, these protocols allow tasks to have incomparable priorities.

One policy is selected in [3] which is comparable with the other protocols presented since it is most applicable for systems when all priority assignments are comparable. This protocol works as follows:

A request by a task T_i for resource r_k at time t is granted if

$$hold_i^k(t) \leq \min_{T_j \in HS_i} (next_j^k(t) - t)$$

Otherwise, the request is delayed.

where

- $hold_i^k(t)$ is the upper bound on the amount of time that task T_i will hold r_k the next time w.r.t. time t ;
 $next_i^k(t)$ is the lower bound on the next time (w.r.t. time t) T_i will request resource

r_k ; and
 HS_i is the set of tasks T_j such that $p_j > p_i$.

The rest of the paper is organized as follows: Section 2 presents a review of duration calculus. Section 3 presents the formalization of the schedulers grouped under two classes. Proofs of certain properties of these schedulers are presented in section 4. Finally, we present some conclusions and discussion in section 5.

2 Duration Calculus

The duration calculus (DC) [13], an extension of interval temporal logic, is a notation to specify and reason about properties of systems and is a logic to verify theorems about such specifications. Since its introduction in 1991, the original duration calculus has been extended into several calculi to increase its versatility in modeling various kinds of systems.

In the extended duration calculus (EDC) [17], piecewise continuous/differentiable functions were introduced, providing the capability to capture properties of continuous states. This calculus is especially useful for modeling and reasoning about hybrid systems. Another extension of DC is the mean-value duration calculus (MVDC) [14]. In MVDC, integrals of Boolean functions are replaced by their mean values, and as a result, δ -functions are used to represent instantaneous actions such as events. This calculus is suitable to model systems with state and events. DC has also been extended with probabilities. One such early attempt is the probabilistic duration calculus for discrete time [7]. In a more recent work [4], a probabilistic duration calculus for continuous time was presented. With these two probabilistic calculi, it is possible to reason about and calculate the dependability of a system with respect to its components. More recently, infinite intervals were introduced into the duration calculus (DCⁱ) [15]. By introducing limits, DCⁱ can deal with unbounded liveness and fairness and can also measure live states by duration limits.

A state in DC is a Boolean function over time. When a state P has a value of 1 at time t , P is said to be present, otherwise, it is absent. States can be combined by the usual Boolean connectives $\neg, \wedge, \vee, \dots$ to form composite states.

One of the most significant features of DC is its ability to express *integrals* of time durations of states. Given an arbitrary state P and an arbitrary observation interval $[b, e]$ ($b, e \in \mathbf{R}$ and $e \geq b$), the duration of P is defined as:

$$\int P \stackrel{\text{def}}{=} \int_b^e P(t) dt$$

Let ℓ denote the length of the observation interval, that is,

$$\ell[b, e] \stackrel{\text{def}}{=} e - b$$

then the duration of the trivial state 1 is defined as follows:

$$\int 1 \hat{=} \ell$$

In DC, $\int P$ and ℓ are terms and can be combined into formulae using arithmetic predicates and the interval modality *chop* (\frown) operator. The formula $B \frown C$ is satisfied by a non-point interval if it can be chopped into two subintervals such that the first subinterval satisfies B and the second satisfies C.

$$(B \frown C)[b, e] \stackrel{\text{def}}{=} \exists m \bullet (b \leq m \leq e) B[b, m] \wedge C[m, e]$$

Using the chop (\frown) operator, the conventional modalities can then be defined as follows:

$$\begin{aligned} \diamond A &\hat{=} true \frown A \frown true && \text{reads "for some subinterval } A \text{ holds"} \\ \square A &\hat{=} \neg \diamond \neg A && \text{reads "for all subintervals } A \text{ holds"} \end{aligned}$$

$$\begin{aligned} [P] &\hat{=} (\int P = \ell) \wedge (\ell > 0) \\ [\] &\hat{=} \ell = 0 \\ [P]^* &\hat{=} [\] \vee [P] \end{aligned}$$

The formula $[P]$ is true if state P holds everywhere (almost) in this non-point interval and $[\]$ is true for point intervals only. And it follows that:

$$\begin{aligned} [1] &\Leftrightarrow (\ell > 0) \\ [0] &\Leftrightarrow false \end{aligned}$$

We also use $\diamond A$ to mean that formula A holds for some initial subintervals of an interval.

$$\begin{aligned} \diamond A &\hat{=} A \frown true \\ \square A &\hat{=} \neg \diamond \neg A \end{aligned}$$

Proof System of DC:

By being an extension of *Interval Temporal Logic*, DC inherited the powerful proof system in ITL. Some of the theorems adopted by DC from ITL are as follows:

- (2.1) $A \Rightarrow B \vdash (A \frown C \Rightarrow B \frown C) \wedge (C \frown A \Rightarrow C \frown B)$ (Monotonicity)
- (2.2) $(A \frown B) \frown C \Leftrightarrow A \frown (B \frown C)$ (Associativity)
- (2.3) $(A \frown [\]) \Leftrightarrow ([\] \frown A) \Leftrightarrow A$ (Unit)
- (2.4) $(A \frown false) \Leftrightarrow (false \frown A) \Leftrightarrow false$ (Zero)
- (2.5) $\begin{aligned} (A \vee B) \frown C &\Leftrightarrow (A \frown C) \vee (B \frown C) \\ C \frown (A \vee B) &\Leftrightarrow (C \frown A) \vee (C \frown B) \end{aligned}$ (\vee -distributivity)

$$(2.6) \quad \Box A \wedge (B \frown C) \Rightarrow (\Box A \wedge B) \frown (\Box A \wedge C) \quad (\Box\text{-distributivity})$$

$$(2.7) \quad \Box A \wedge (B \frown C) \Rightarrow (\Box A \wedge B) \frown C \quad (\Box\text{-distributivity})$$

Note that although \frown is associative (2.2), has the point interval ($[]$) as unit (2.3), it only distributes through disjunction (\vee) (2.5), but not through conjunction (\wedge). The following, however, is a theorem:

$$(2.8) \quad \begin{aligned} (A \wedge B) \frown C &\Rightarrow (A \frown C) \wedge (B \frown C) \\ C \frown (A \wedge B) &\Rightarrow (C \frown A) \wedge (C \frown B) \end{aligned}$$

The following are the axioms of DC:

$$(2.9) \quad \int 0 = 0$$

$$(2.10) \quad \int P \geq 0$$

$$(2.11) \quad \int P + \int Q = (\int P \vee \int Q) + (\int P \wedge \int Q)$$

$$(2.12) \quad \int P = r + s \Leftrightarrow \int P = r \frown \int P = s$$

From these axioms, it is not difficult to show that the following are also theorems in DC:

$$(2.13) \quad [P] \frown [P] \Leftrightarrow [P]$$

$$(2.14) \quad [P] \Rightarrow \Box([P]^*)$$

$$(2.15) \quad [P] \wedge [Q] \Leftrightarrow [P \wedge Q]$$

$$(2.16) \quad \int P + \int \neg P = \ell$$

$$(2.17) \quad \int P \leq \ell$$

$$(2.18) \quad [P] \wedge [A] \frown [B] \Leftrightarrow ([P] \wedge [A]) \frown ([P] \wedge [B])$$

$$(2.19) \quad (r \leq \int P \leq s) \frown (t \leq \int P \leq u) \Rightarrow (r + t \leq \int P \leq s + u)$$

States in the duration calculus must satisfy the finite variability property. Informally, this property means that for every state expression P , any interval $[b, e]$ where $b < e$, can be divided or partitioned into finitely many subintervals such that P is constant on each open subinterval.

First let, $\Gamma_1, \Gamma_2, \dots, \Gamma_n \vdash \Gamma$ mean that we can make a formal deduction of the formula Γ from $\Gamma_1, \Gamma_2, \dots, \Gamma_n$ (called assumption formulas) using the proof system of DC. A formal deduction of the formula Γ from $\Gamma_1, \Gamma_2, \dots, \Gamma_n$ is a finite sequence of one or more formulas, such that each formula in the sequence is either one of the $\Gamma_1, \Gamma_2, \dots, \Gamma_n$ or an axiom, or an immediate consequence of the preceding formulas of the sequence. The formula Γ is the last formula in the sequence and is called the conclusion.

Then, this finite variability property is formalized as the following two induction rules:

Forward Induction: Let $\mathcal{H}(\mathcal{X})$ be a DC formula schema containing the formula letter \mathcal{X} , and let P be any state expression.

If $\mathcal{H}([\])$ and $\mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \vee (\mathcal{X} \wedge [P]) \vee (\mathcal{X} \wedge [\neg P]))$
 then $\mathcal{H}(true)$

Backward Induction: Let $\mathcal{H}(\mathcal{X})$ be a DC formula schema containing the formula letter \mathcal{X} , and let P be any state expression.

If $\mathcal{H}([\])$ and $\mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \vee ([P] \wedge \mathcal{X}) \vee ([\neg P] \wedge \mathcal{X}))$
 then $\mathcal{H}(true)$

3 Formalization of Scheduling for Tasks with Shared Resources

The approach here is to first define precisely a model for the tasks and shared resources. This is done by defining variables to represent certain attributes of tasks like arrival time, execution time, etc. Afterward, DC state variables and DC formulas based on these variables are specified to represent tasks and their behavior with respects shared resources.

The system is modeled as having a set \mathcal{T} of n tasks, $T_i, 1 \leq i \leq n$. Tasks are released at arbitrary predetermined times λ_i and we assume that tasks arrive only once and are non-recurrent. Each task T_i has computation time C_i , relative deadline d_i and priority p_i . We assume that the priority of a task depends on its deadline and is fixed.

The system also maintains a set \mathcal{R} of m non-preemptive resources, $r_k, 1 \leq k \leq m$. Associated with each task T_i is a set $R_i \subseteq \mathcal{R}$ of resources it will require during execution. For simplicity, we assume each resource $r \in R_i$ is requested and released by T_i once during its execution. Tasks may have only one outstanding request at a time. This means that tasks requests for resources in a sequential manner. Each resource $r_k \in R_i$ will be needed by T_i after having accumulated a runtime of $\chi_i(r_k)$. If a task T_i locks resource r_k , it will voluntarily release r_k only after accumulating a runtime of $v_i(r_k)$. For PIP and PCP, resource access is assumed to be properly nested, that is, while a task is acquiring resources, it cannot release any. But once it has started to release a resource, it cannot acquire resources anymore.

For the first class of protocols, we assume a uniprocessor model. For the second class of protocols, we assume that the number of processors is equal to n , the number of tasks.

Fig. 3.2 depicts the system model.

The following state variables are introduced for task T_i :

$T_i.arrived$	- task T_i is in the system
$T_i.rdy$	- task T_i is ready to be allocated the processor and is not blocked
$T_i.run$	- task T_i is running on the processor
$T_i.requests(r_k)$	- task T_i is accessing resource r_k

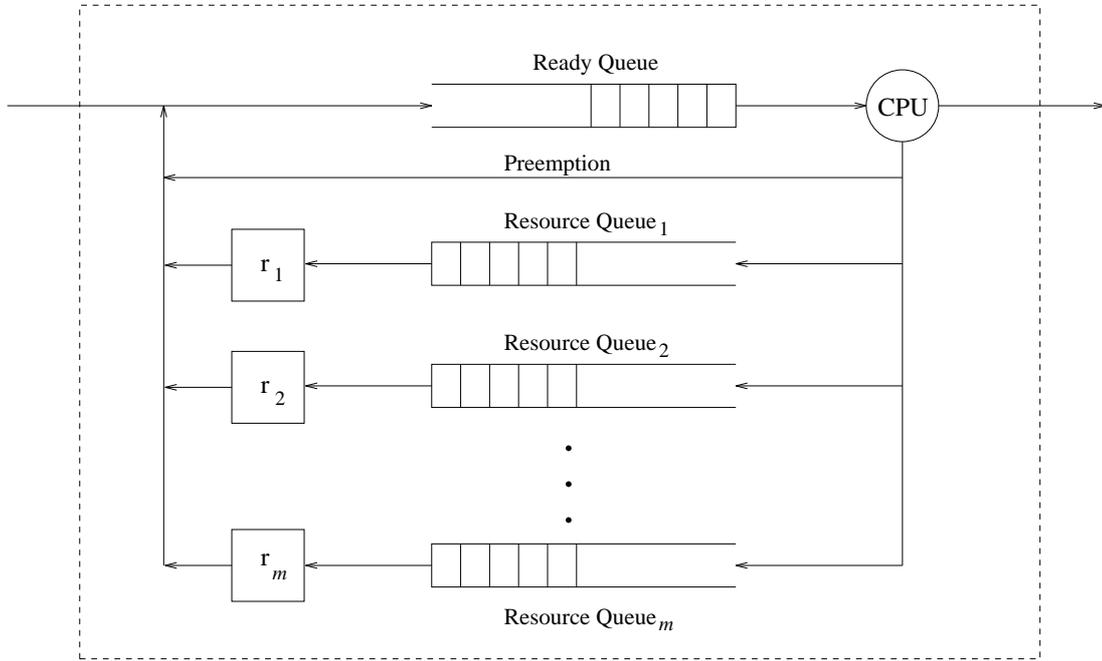


Figure 3.2: System Model

- $T_i.blocked(r_k)$ - task T_i is blocked on resource r_k
- $T_i.holds(r_k)$ - resource r_k is currently held by task T_i
- $T_i.done$ - task T_i is done and has left the system

Using the system model in Fig. 1.1, $T_i.arrived$ holds for any task that has entered the system (bounded by the dashed region) and has not left the system. $T_i.rdy$ means that T_i is in the ready queue waiting for the scheduler to allocate processor to it. When T_i is finally allocated processor time, $T_i.run$ holds and in the system model, this means that T_i is allocated CPU time. If T_i is preempted (due to the arrival of higher priority task), it simply returns to the ready queue. $T_i.requests(r_k)$ holds when a task needs to acquire a resource r_k . Depending on the resource allocation protocol, if r_k is not granted to T_i , it will be blocked in the r_k 's queue. On the other hand, if T_i is granted the resource r_k , it will return to the ready queue (and return to ready state) and wait for processor time. Finally, upon completion of its computation time, the task T_i leaves the system and consequently, $T_i.done$ holds.

The relationship between these states are expressed by the following state expressions:

- For all tasks T_i , either $T_i.arrived$ or $T_i.done$

$$T_i.arrived \vee T_i.done \tag{TS_1}$$

These two states are mutually exclusive:

$$T_i.arrived \Rightarrow \neg T_i.done \quad TS_2$$

- A task that is present in the system is either ready or blocked

$$T_i.arrived \Leftrightarrow T_i.rdy \vee T_i.blocked \quad TS_3$$

where $T_i.blocked \hat{=} \bigvee_{r_k \in \mathcal{R}} T_i.blocked(r_k)$

States $T_i.rdy$ and $T_i.blocked$ are mutually exclusive:

$$T_i.rdy \Rightarrow \neg T_i.blocked \quad TS_4$$

- A task T_i runs only if it is ready

$$T_i.run \Rightarrow T_i.rdy \quad TS_5$$

- When a task T_i is requesting for a resource r_k , it is either holding it or blocked on it

$$\bigwedge_{r_k \in \mathcal{R}} (T_i.requests(r_k) \Leftrightarrow T_i.holds(r_k) \vee T_i.blocked(r_k)) \quad TS_6$$

A task T_i can request resources if they are present in the system:

$$\bigwedge_{r_k \in \mathcal{R}} (T_i.requests(r_k) \Rightarrow T_i.arrived) \quad TS_7$$

States $T_i.holds(r_k)$ and $T_i.blocked(r_k)$ are mutually exclusive:

$$\bigwedge_{r_k \in \mathcal{R}} (T_i.holds(r_k) \Rightarrow \neg T_i.blocked(r_k)) \quad TS_8$$

Let us denote all these predicates as:

$$TSREL \hat{=} \bigwedge_{i=1}^8 TS_i$$

With these states, the behavior of tasks in any interval $[0, t]$ is specified as follows:

Arrival Time: A task T_i arrives after λ_i time units (assuming that $\lambda_i > 0$):

$$ARR \hat{=} \bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} \ell > \lambda_i \\ \Rightarrow (\ell = \lambda_i \wedge [\neg T_i.arrived]) \frown [T_i.arrived] \frown true \end{array} \right)$$

Completion Time: A task T_i stays in the system until it has accumulated C_i time units of processor time:

$$CPLT \hat{=} \bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} \int T_i.run = C_i \wedge \ell > 0 \\ \Leftrightarrow \int T_i.run = C_i \wedge [T_i.done] \wedge true \end{array} \right)$$

Resources

Resources are allocated in a mutually exclusive manner, i.e. a resource can be held by at most one task at any time:

$$MUTX \hat{=} \bigwedge_{T_i \in \mathcal{T}} \left(\bigwedge_{r_k \in \mathcal{R}} (T_i.holds(r_k) \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \neg T_j.holds(r_k)) \right)$$

Resource allocation is nonpreemptive:

$$NPRV \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in R_i} \square \left(\begin{array}{l} [T_i.holds(r_k)] \wedge [T_i.requests(r_k)] \\ \Rightarrow [T_i.holds(r_k)] \end{array} \right)$$

Conjoining the predicates above, we have:

$$RES \hat{=} MUTX \wedge NPRV$$

Behavior of Tasks and Resources

Now it is necessary to model how tasks behave with respect to resources. The first thing is representing precisely when a task will request for a particular resource it needs:

Acquisition Time A task T_i will need the resource $r_k \in R_i$ only after having accumulated processor time of exactly $\chi_i(r_k)$ time units.

Let

$$T_i.NOT-REQ(r_k) \hat{=} (\int T_i.run = \chi_i(r_k) \wedge [\neg T_i.requests(r_k)]^*)$$

then

$$ACQ \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in R_i} \square \left(\begin{array}{l} [\neg T_i.requests(r_k)]^* \wedge [T_i.requests(r_k)] \\ \Rightarrow T_i.NOT-REQ(r_k) \wedge [T_i.requests(r_k)] \end{array} \right)$$

The following formula specifies how long a task will hold onto a resource once it has been allocated it.

Hold Time A task T_i releases a resource r_k after it has received accumulated processor time of $v_i(r_k)$ time units while holding r_k .

$$HOLD \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in R_i} \square \left(\begin{array}{l} [\neg T_i.requests(r_k)]^* \wedge [T_i.requests(r_k)] \wedge [\neg T_i.requests(r_k)] \\ \Rightarrow \int (T_i.run \wedge T_i.holds(r_k)) = v_i(r_k) \end{array} \right)$$

It is natural to assume that a task needs to be allocated processor time in order to request access to any resource. This is expressed as:

$$REQ \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in R_i} \square ([\neg T_i.requests(r_k)] \wedge [T_i.requests(r_k)] \Rightarrow \diamond [T_i.run])$$

For PIP and PCP protocols, accesses to resources are properly nested. For each task T_i , there is a total order among resources R_i , denoted by $<_i$, such that:

$$NEST \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \neq r_l \in R_i} \square \left(\begin{array}{l} r_k <_i r_l \wedge [T_i.requests(r_k)] \\ \Rightarrow [T_i.requests(r_l)] \end{array} \right)$$

This formula means that given any two resources, r_k and r_l , if the request for r_k is nested within the request for r_l , then $r_k <_i r_l$.

It is also obviously reasonable for all tasks to release all resources they are holding on or before termination.

$$REL \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in R_i} (\chi_i(r_k) + v_i(r_k) \leq C_i)$$

The conjunction of the preceding formulas constitute our model for the tasks for the PIP and PCP.

$$TASK \hat{=} ARR \wedge CPLT \wedge ACQ \wedge HOLD \wedge REQ \wedge NEST \wedge \\ REL \wedge TSREL \wedge RES$$

However, for the second class of protocols, TPB and RP, this nested access assumption is relaxed. Hence we have:

$$TASK2 \hat{=} ARR \wedge CPLT \wedge ACQ \wedge HOLD \wedge REQ \wedge REL \wedge \\ TSREL \wedge RES$$

Environment

Priority assignment is a total order:

$$PRIO \hat{=} \bigwedge_{T_i \neq T_j \in \mathcal{T}} (p_i < p_j \vee p_i > p_j)$$

Since uniprocessor and multiprocessor schedulers are considered here, we will present DC formula to express these as environments:

Under a uniprocessor model, we have:

$$ONEPROC \cong \bigwedge_{T_i \neq T_j \in \mathcal{T}} \Box [T_i.run \Rightarrow \neg T_j.run]^*$$

Under our multiprocessor model (where there is one processor for each task), we have:

$$MULPROC \cong \bigwedge_{T_i \in \mathcal{T}} \Box [T_i.rdy \Rightarrow T_i.run]^*$$

This means that there is no need to separately specify the scheduler for the processor, since a task can run as soon as it is ready.

There is no overhead for the scheduler:

$$NOHD \cong \bigwedge_{T_i \in \mathcal{T}} \Box [T_i.rdy \Rightarrow \bigvee_{T_j \in \mathcal{T}} T_j.run]^*$$

Hence for the uniprocessor model, we have:

$$ENV1 \cong PRIO \wedge ONEPROC \wedge NOHD$$

For the multiprocessor model, the environment is:

$$ENV2 \cong PRIO \wedge MULPROC$$

Scheduler

The scheduler for tasks with shared resources can be expressed as having two components:

- Processor Scheduling: policy concerned with the selection of tasks to run on the processor; and
- Resource Control: policy concerning the selection of which task will be allocated a resource it is requesting.

There are three rules for resource control:

- Granting Rule: used to decide if the resource requested is granted or not.
- Blocking Rule: used to decide whether a task is blocked on its request for a resource or not; and

Unblocking Rule: used for deciding which among the blocked tasks is to be granted the resource.

With this general framework, the protocols identified earlier can now be precisely specified. There are two classes of protocols: (a) protocols that allow priority inversion to occur; and (b) protocols that explicitly prevent priority inversions from occurring.

3.1 Protocols that Permit Priority Inversion

Using the framework presented above, we present duration calculus formula schemas for specification of schedulers within this class of protocols. The schemas serve as templates for formulas in the sense that to specify a scheduler or protocol, the only thing to do is to define certain functions to be used with the formula schemas.

First, the formula schema for the preemptive priority scheduler is presented as follows:

Preemptive Priority Scheduler

Let $HiPri(T_i, T_j)$ be a Boolean-valued relation for denoting which task between T_i and T_j is to be executed by the processor. The preemptive priority scheduler can be expressed in terms of $HiPri$ as:

$$PPS(HiPri) \hat{=} \bigwedge_{T_i \neq T_j \in \mathcal{T}} \square([T_i.run] \wedge [T_j.rdy] \Rightarrow [HiPri(T_i, T_j)])$$

Resource control can be specified by defining the three rules mentioned earlier. The granting rule for this class of protocols is expressed as:

Granting Rule

$$PTCL1(Right) \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \square \left(\begin{array}{l} [\neg T_i.holds(r_k)] \wedge [T_i.holds(r_k)] \\ \Rightarrow \diamond [Right(T_i, r_k)] \end{array} \right)$$

where $Right(T_i, r_k)$ is a relation that holds if task T_i is granted access to resource r_k .

The blocking rule for this class of protocols can be expressed as:

Blocking Rule

$$PTCL2(Right) \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \square([T_i.blocked(r_k)] \Rightarrow [\neg Right(T_i, r_k)])$$

Then, the unblocking rule can be specified as:

Unblocking Rule

$$PTCL3(HiPri) \hat{=} \bigwedge_{T_i \neq T_j \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \square \left(\begin{array}{l} [T_i.blocked(r_k) \wedge T_j.blocked(r_k)] \wedge [\neg T_i.blocked(r_k)] \\ \Rightarrow HiPri(T_i, T_j) \end{array} \right)$$

By combining these formula schemas together, the scheduler, \mathcal{SCH} , is obtained:

$$\mathcal{SCH}(HiPri, Right) \hat{=} \left(\begin{array}{l} PPS(HiPri) \wedge PTCL1(Right) \\ \wedge PTCL2(Right) \wedge PTCL3(HiPri) \end{array} \right)$$

Trivial Protocol A

First, the relation $HiPri_{TPA}$ is defined using the original priority assigned to tasks:

$$HiPri_{TPA}(T_i, T_j) \hat{=} (p_i > p_j)$$

and the relation $Right_{TPA}$ is defined as:

$$Right_{TPA}(T_i, r_k) \hat{=} \bigwedge_{T_j \neq T_i \in \mathcal{T}} \neg T_j.holds(r_k)$$

From these, the TPA scheduler can be instantiated from the formula schema \mathcal{SCH} as:

$$TPA \hat{=} \mathcal{SCH}(HiPri_{TPA}, Right_{TPA})$$

Priority Inheritance Protocol

The difference between the priority inheritance protocol and the trivial protocol A above is in the assignment of priorities to tasks.

The priority inheritance protocol bounds the length of priority inversions by temporarily assigning a higher priority to tasks while they are blocking other tasks. This is to ensure that they are given enough processor time to use (and eventually release) the resource(s) granted to them. First we define the following abbreviation:

$$T_i.blockedby(T_j) \hat{=} \bigvee_{r_k \in \mathcal{R}} (T_j.holds(r_k) \wedge T_i.blocked(r_k))$$

From this, the state function $HiPri$ for PIP, $HiPri_{PIP}$, satisfies the following inheritance properties:

(a) $HiPri_{PIP}$ is a total order:

$$\bigwedge_{T_i \neq T_j \in \mathcal{T}} (HiPri_{PIP}(T_i, T_j) \Rightarrow \neg HiPri_{PIP}(T_j, T_i))$$

$$\bigwedge_{T_i \neq T_j \neq T_k \in \mathcal{T}} \left(\begin{array}{l} HiPri_{PIP}(T_i, T_k) \wedge HiPri_{PIP}(T_k, T_j) \\ \Rightarrow HiPri_{PIP}(T_i, T_j) \end{array} \right)$$

(b) $HiPri_{PIP}$ depends on the priority inherited by tasks:

$$\bigwedge_{T_i \neq T_j \neq T_k \in \mathcal{T}} \left(\begin{array}{l} T_k.blockedby(T_i) \\ \Rightarrow (HiPri_{PIP}(T_k, T_j) \Rightarrow HiPri_{PIP}(T_i, T_j)) \end{array} \right)$$

$$\bigwedge_{T_i \neq T_j \in \mathcal{T}} \left(\begin{array}{l} \bigwedge_{T_k \in \mathcal{T}} (\neg T_k.blockedby(T_i)) \\ \Rightarrow (HiPri_{PIP}(T_i, T_j) \Rightarrow p_i > p_j) \end{array} \right)$$

The first formula expresses that when a task T_i inherits the priority of another task T_k , if $HiPri_{PIP}(T_k, T_j)$ then $HiPri_{PIP}(T_i, T_j)$. The second formula states that if a task T_i does not inherit any priority, then the relation $HiPri_{PIP}$ is consistent with the original assigned priorities.

Hence, the formula for PIP is:

$$PIP \hat{=} SCH(HiPri_{PIP}, Right_{TPA})$$

It should be noted that the blocking protocol for PIP is the same as the one used in TPA, so the *Right* function defined for TPA is used here for PIP.

Priority Ceiling Protocol

First, the priority ceiling of resources is formalized. Since the number of tasks in the system and the priorities of the tasks are constant, therefore, the priority ceiling of resource r_k , $c(r_k)$ is also constant. Hence,

$$c(r_k) \hat{=} \max_i \{ p_i \mid r_k \in R_i \}$$

The *blockedby* state function for PCP is:

$$T_i.blockedby'(T_j) \hat{=} \bigvee_{r_k \in \mathcal{R}} (T_j.holds(r_k) \wedge T_i.blocked \wedge c(r_k) \geq p_i)$$

PCP is a form of priority inheritance protocol, so the relation $HiPri_{PCP}$ also satisfies the inheritance properties for $HiPri_{PIP}$ except that we use $T_i.blockedby'$ instead of $T_i.blockedby$.

The *Right* function for PCP is:

$$Right_{PCP}(T_i) \hat{=} \bigwedge_{r_k \in \mathcal{R}} \left(\bigwedge_{T_j \neq T_i \in \mathcal{T}} T_j.\text{holds}(r_k) \Rightarrow p_i > c(r_k) \right)$$

Hence, for PCP we have:

$$PCP \hat{=} SCH(HiPri_{PCP}, Right_{PCP})$$

3.2 Protocols that Prevent Priority Inversions

Under this class, protocols prevent priority inversions by delaying lower priority tasks sufficiently long to prevent them from blocking higher priority tasks. The environment here is a multiprocessor environment, *ENV2*. Both protocols presented below makes decisions using information regarding the past or the future.

So, we make some modifications to our duration formula schemas presented at the start of the previous section. Specifically, the state function *Right* will be replaced with a DC formula which serves the same purpose but can deal with conditions pertaining to a time interval and not just the current state of the system at a specific time point. This modification is necessary to properly specify protocols under this class.

DRight(T_i, r_k) is defined to be the DC formula that holds if task T_i is granted resource r_k based on the conditions in the observation interval.

The blocking and unblocking rule which depends of *DRight* are replaced by a single formula schema:

$$PTCL'(DRight) \hat{=} \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \square(\text{true} \wedge [\neg T_i.\text{blocked}(r_k)] \Leftrightarrow DRight(T_i, r_k))$$

From this, the scheduler SCH' for the second class of protocols is expressed as the following:

$$SCH'(DRight) \hat{=} PTCL'(DRight)$$

Trivial Protocol B

This protocol works as follows: a task T_i that requests a resource r_k is granted if all other higher priority tasks have finished using r_k .

The DC formula to denote when a task is granted a resource is defined as:

$$DRight_{TPB}(T_i, r_k) \hat{=} \left(\begin{array}{l} \text{true} \wedge [T_i.\text{requests}(r_k)] \\ \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} (p_j > p_i \wedge r_k \in R_j \Rightarrow \text{Finished}(T_j, r_k)) \end{array} \right)$$

where $Finished(T_j, r_k) \hat{=} \int T_j.run \geq \chi_j(r_k) + v_j(r_k)$.

Hence for TPB, the scheduler is defined as:

$$TPB \hat{=} SCH'(DRight_{TPB})$$

Reservation Protocol

Informally, this protocol works as follows: a task T_i that requests a resource r_k is granted if the time it will release r_k before the any higher priority task request r_k or all higher priority tasks have finished using r_k .

The expression $DRight_{RP}$ is defined as:

$$DRight_{RP}(T_i, r_k) \hat{=} \left(\begin{array}{l} true \wedge [T_i.requests(r_k)] \\ \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(\begin{array}{l} p_j > p_i \wedge r_k \in R_j \\ \Rightarrow NotReq(T_i, T_j, r_k) \vee Finished(T_j, r_k) \end{array} \right) \end{array} \right)$$

where

$$\begin{aligned} NotReq(T_i, T_j, r_k) &\hat{=} \ell - \int (T_i.holds(r_k) \wedge T_i.run) + v_i(r_k) \leq \lambda_j + \chi_j(r_k) + v_j(r_k) \\ Finished(T_j, r_k) &\hat{=} \int T_j.run \geq \chi_j(r_k) + v_j(r_k) \end{aligned}$$

The formula $NotReq(T_i, T_j, r_k)$ holds on an observation interval if T_i will release resource r_k before task T_j requests for it.

Finally, we have

$$RP \hat{=} SCH'(DRight_{RP})$$

3.3 Requirements

The requirement that deadlines of all tasks are to be met can be expressed as:

$$RQT \hat{=} \bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} true \wedge ([T_i.arrived] \wedge \ell = d_i) \\ \Rightarrow \int T_i.run = C_i \end{array} \right)$$

Other requirements for schedulers are: deadlock freedom and no priority inversion.

One way to expressed deadlock freedom in DC is:

$$NODLCK \hat{=} \square \neg \left(\left[\bigwedge_{T_i \in \mathcal{T}} (T_i.done \vee T_i.blocked) \wedge \bigvee_{i \in \mathcal{T}} T_i.blocked \right] \right)$$

and no priority inversion can be expressed as:

$$NOINV \cong \bigwedge_{T_i \neq T_j \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \square \left(\begin{array}{l} [T_i.holds(r_k) \wedge T_j.blocked(r_k)] \\ \Rightarrow HiPri(T_i, T_j) \end{array} \right)$$

In the next section, we present formal proofs that some schedulers satisfy some of these requirements through formal proofs.

4 Proving Properties

The advantages of using a formal specification language like the duration calculus is in the ability to prove properties from the specifications. In this section, we will show how we can use some of these specifications in the previous section to prove properties like blocked at most once, deadlock freedom and no priority inversion.

4.1 Blocked At Most Once Property of PCP

In order to prove this property, we need to make a distinction between a task being in the preempted state and blocked state. We make the assumption that while a task is preempted by a higher priority task, it is not blocked.

$$A1 \cong \bigwedge_{T_i \neq T_j \in \mathcal{T}} \square \left(\begin{array}{l} [T_i.run] \wedge [\bigvee_{T_j \neq T_i \in \mathcal{T}} (T_j.run \wedge p_j > p_i)] \\ \Rightarrow [T_i.run] \wedge [\neg T_i.blocked] \end{array} \right)$$

First, we define the notion of *critical region* for a task. Since resources are accessed in a properly nested manner, the critical region for a task T_i is defined as the time interval beginning at the acquisition of its first resource and ending at the release of the last resource it is holding.

Formally,

$$\text{DEFINITION 4.1} \quad T_i.in-cr \cong \bigvee_{r_k \in \mathcal{R}} T_i.holds(r_k)$$

First we have the following lemma:

$$\text{LEMMA 4.1} \quad \begin{array}{l} A1 \wedge PCP \wedge TASK \wedge ENV \vdash \\ \bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} [T_i.in-cr] \\ \Rightarrow \left[\bigwedge_{T_j \neq T_i \in \mathcal{T}} \bigwedge_{r_l \in \mathcal{R}} \left(\begin{array}{l} T_j.holds(r_l) \wedge HiPri_{PCP}(T_i, T_j) \\ \Rightarrow p_i > c(r_l) \end{array} \right) \right] \end{array} \right) \end{array}$$

PROOF.

The proof of this LEMMA is presented in Appendix A. \square

Then, we can prove the following lemma:

$$\text{LEMMA 4.2} \quad A1 \wedge PCP \wedge TASK \wedge ENV \vdash \bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} [\neg \text{Right}_{PCP}(T_i) \wedge T_i.in-cr] \\ \Rightarrow [\bigvee_{T_j \neq T_i \in \mathcal{T}} (T_j.run \wedge \text{HiPri}_{PCP}(T_j, T_i))] \end{array} \right)$$

PROOF. For all tasks T_i :

$$\begin{aligned} & [\neg \text{Right}_{PCP}(T_i) \wedge T_i.in-cr] \\ \Rightarrow & \left[\begin{array}{l} \neg \text{Right}_{PCP}(T_i) \\ \wedge \bigwedge_{T_j \neq T_i \in \mathcal{T}} \bigwedge_{r_l \in \mathcal{R}} \left(\begin{array}{l} T_j.holds(r_l) \wedge \text{HiPri}_{PCP}(T_i, T_j) \\ \Rightarrow p_i > c(r_l) \end{array} \right) \end{array} \right] & \text{LEMMA 4.1} \\ \Rightarrow & \left[\begin{array}{l} \bigvee_{T_j \neq T_i \in \mathcal{T}} \bigvee_{r_l \in \mathcal{R}} (T_j.holds(r_l) \wedge p_i \leq c(r_l)) \\ \wedge \bigwedge_{T_j \neq T_i \in \mathcal{T}} \bigwedge_{r_l \in \mathcal{R}} \left(\begin{array}{l} T_j.holds(r_l) \wedge \text{HiPri}_{PCP}(T_i, T_j) \\ \Rightarrow p_i > c(r_l) \end{array} \right) \end{array} \right] & \begin{array}{l} \text{Def. of} \\ \text{Right}_{PCP} \end{array} \\ \Rightarrow & \left[\bigvee_{T_j \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_j, T_i) \wedge T_j.in-cr) \right] & \begin{array}{l} \text{Prop. Logic,} \\ \text{Def. 4.1} \end{array} \end{aligned}$$

We have established this:

$$\bigwedge_{T_i \in \mathcal{T}} \square \left(\begin{array}{l} [\neg \text{Right}_{PCP}(T_i) \wedge T_i.in-cr] \\ \Rightarrow [\bigvee_{T_j \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_j, T_i) \wedge T_j.in-cr)] \end{array} \right) \quad (1)$$

For a given T_i :

$$\begin{aligned} & [\neg \text{Right}_{PCP}(T_i) \wedge T_i.in-cr] \\ \Rightarrow & \left[\bigvee_{T_k \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_k, T_i) \wedge T_k.in-cr) \right] & \text{Formula (1)} \\ \Rightarrow & \left[\begin{array}{l} \bigvee_{T_k \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_k, T_i) \wedge T_k.in-cr \wedge \text{Right}_{PCP}(T_k)) \\ \vee \bigvee_{T_k \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_k, T_i) \wedge T_k.in-cr \wedge \neg \text{Right}_{PCP}(T_k)) \end{array} \right] & \text{Prop. Logic} \end{aligned}$$

Since \mathcal{T} is finite and HiPri_{PCP} is a total order, repeated application (at most n times, where n is the number of tasks) of formula (1) on:

$$\bigvee_{T_k \neq T_i \in \mathcal{T}} (HiPri_{PCP}(T_k, T_i) \wedge T_k.in-cr \wedge \neg Right_{PCP}(T_k))$$

we have that:

$$\begin{aligned} & \bigvee_{T_k \neq T_i \in \mathcal{T}} (HiPri_{PCP}(T_k, T_i) \wedge T_k.in-cr \wedge \neg Right_{PCP}(T_k)) \\ & \Rightarrow \bigvee_{T_k \neq T_i \in \mathcal{T}} (HiPri_{PCP}(T_k, T_i) \wedge T_k.in-cr \wedge Right_{PCP}(T_k)) \end{aligned}$$

Finally, for all tasks T_i , we have:

$$\begin{aligned} & [\neg Right_{PCP}(T_i) \wedge T_i.in-cr] \\ & \Rightarrow [\bigvee_{T_j \neq T_i \in \mathcal{T}} (Right_{PCP}(T_j) \wedge HiPri_{PCP}(T_j, T_i) \wedge T_j.in-cr)] && \text{Prop. Logic} \\ & \Rightarrow [\bigvee_{T_j \in \mathcal{T}} (T_j.rdy \wedge HiPri_{PCP}(T_j, T_i) \wedge T_j.in-cr)] && \begin{array}{l} PCP - \\ PTCL2(Right_{PCP}) \end{array} \\ & \Rightarrow [\bigvee_{T_j \in \mathcal{T}} (T_j.run \wedge HiPri_{PCP}(T_j, T_i))] && \begin{array}{l} PCP - \\ PPS(HiPri_{PCP}), \\ NOHD \end{array} \end{aligned}$$

The LEMMA is then proved. □

The property of PCP where a task is blocked at most once before entering its critical region can be expressed as follows:

$$\text{THEOREM 4.1} \quad A1 \wedge PCP \wedge TASK \wedge ENV \vdash \bigwedge_{T_i \in \mathcal{T}} \Box([T_i.in-cr] \Rightarrow [\neg T_i.blocked])$$

PROOF.

We prove this by induction: First, assume:

$$\begin{aligned} \mathcal{H}(\mathcal{X}) & \hat{=} \mathcal{X} \wedge [T_i.in-cr] \Rightarrow [\neg T_i.blocked] \\ \Gamma & \hat{=} A1 \wedge PCP \wedge TASK \wedge ENV \end{aligned}$$

Base case: $\Gamma \vdash \mathcal{H}([\])$

$$\begin{aligned} & [\] \wedge [T_i.in-cr] \Rightarrow [\neg T_i.blocked] \\ & \Rightarrow false \Rightarrow [\neg T_i.blocked] && \text{ITL} \\ & \Rightarrow true && \text{ITL} \end{aligned}$$

For the inductive step, we must establish:

$$\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \vee (\mathcal{X} \frown [\mathit{Right}_{PCP}(T_i)]) \vee (\mathcal{X} \frown [\neg \mathit{Right}_{PCP}(T_i)]))$$

We now consider two cases:

1. $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\mathit{Right}_{PCP}(T_i)])$
2. $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\neg \mathit{Right}_{PCP}(T_i)])$

Case 1: $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\mathit{Right}_{PCP}(T_i)])$

$$\mathcal{X} \frown [\mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr]$$

$$\Rightarrow (\mathcal{X} \wedge [T_i.in-cr]) \frown ([\mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr])$$

Theorem
2.18

$$\Rightarrow [\neg T_i.blocked] \frown ([\mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr])$$

$\mathcal{H}(\mathcal{X})$

$$\Rightarrow [\neg T_i.blocked] \frown [\neg T_i.blocked]$$

PCP

$$\Rightarrow [\neg T_i.blocked]$$

Axiom 2.13

Case 2: $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\neg \mathit{Right}_{PCP}(T_i)])$

$$\mathcal{X} \frown [\neg \mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr]$$

$$\Rightarrow (\mathcal{X} \wedge [T_i.in-cr]) \frown ([\neg \mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr])$$

Theorem
2.18

$$\Rightarrow [\neg T_i.blocked] \frown ([\neg \mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr])$$

$\mathcal{H}(\mathcal{X})$

$$\Rightarrow [\neg T_i.blocked] \frown ([\neg \mathit{Right}_{PCP}(T_i)] \wedge [T_i.in-cr])$$

Rule 2.15

$$\Rightarrow [\neg T_i.blocked] \frown \left[\bigvee_{T_j \in \mathcal{T}} (T_j.run \wedge \mathit{HiPri}_{PCP}(T_j, T_i)) \right]$$

LEMMA 4.2

$$\Rightarrow [\neg T_i.blocked] \frown [\neg T_i.blocked]$$

$A1$

$$\Rightarrow [\neg T_i.blocked]$$

Axiom 2.13

This completes the proof of the theorem. □

4.2 PCP is Deadlock Free

We prove this property by contradiction.

THEOREM 4.2 $A1 \wedge PCP \wedge TASK \wedge ENV \vdash NODLCK$

PROOF.

- | | |
|---|---|
| (1) $\neg \Box \neg (\bigwedge_{T_i \in \mathcal{T}} (T_i.done \vee T_i.blocked) \wedge \bigvee_{i \in \mathcal{T}} T_i.blocked)$ | Assume |
| (2) $\Diamond (\bigwedge_{T_i \in \mathcal{T}} [(T_i.done \vee T_i.blocked)] \wedge \bigvee_{T_i \in \mathcal{T}} [T_i.blocked])$ | ITL |
| (3) $\Box (\bigwedge_{T_i \in \mathcal{T}} ([T_i.blocked] \Rightarrow [\bigvee_{T_j \in \mathcal{T}} T_j.in-cr]))$ | <i>PCP</i> -
<i>PTCL2(RightPCP)</i> ,
Def. of <i>RightPCP</i> ,
Def. 4.1 |
| (4) $\Box ([\bigvee_{T_i \in \mathcal{T}} T_i.in-cr] \Rightarrow [\bigvee_{T_i \in \mathcal{T}} (\neg T_i.blocked \wedge T_i.in-cr)])$ | (3), THEOREM 4.1,
ITL |
| (5) $\Box (\bigwedge_{T_i \in \mathcal{T}} ([T_i.blocked] \Rightarrow [\bigvee_{T_j \in \mathcal{T}} (\neg T_j.blocked \wedge T_j.in-cr)]))$ | (3), (4), Prop. Logic |
| (6) $\Diamond (\bigwedge_{T_i \in \mathcal{T}} [(T_i.done \vee T_i.blocked)] \wedge \bigvee_{T_i \in \mathcal{T}} [\neg T_i.blocked \wedge T_i.in-cr])$ | (2), (5), Prop. Logic |
| (7) $\Diamond (\bigvee_{T_i \in \mathcal{T}} [\neg T_i.blocked \wedge T_i.in-cr \wedge (T_i.done \vee T_i.blocked)])$ | (6), Prop. Logic |
| (8) <i>false</i> | (7), <i>TASK</i> , Prop.
Logic |
| (9) <i>NODLCK</i> | (1), (8), Prop. Logic |

□

4.3 No Priority Inversion

Now, we will prove that the Reservation Protocol does not permit priority inversions to occur. First, we consider a lemma:

LEMMA 4.3

$$RP \wedge TASK2 \wedge ENV2 \vdash \bigwedge_{T_i \in \mathcal{T}} \bigwedge_{r_k \in \mathcal{R}} \Box \left(\begin{array}{l} [\neg T_i.holds(r_k)] \wedge [T_i.holds(r_k)] \\ \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(\begin{array}{l} p_j > p_i \wedge r_k \in R_j \\ \Rightarrow [\neg T_i.holds(r_k)] \wedge [\neg T_j.blocked(r_k)] \end{array} \right) \end{array} \right)$$

PROOF.

$$\begin{aligned}
& true \wedge [T_i.holds(r_k)] \\
& \Rightarrow true \wedge [\neg T_i.blocked(r_k)] && \text{Def. of } holds \\
& \Rightarrow DRight_{RP}(T_i, r_k) && RP \\
& \Rightarrow \left(true \wedge [T_i.requests(r_k)] \right. \\
& \quad \left. \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(p_j > p_i \wedge r_k \in R_j \right. \right. \\
& \quad \quad \left. \left. \Rightarrow NotReq(T_i, T_j, r_k) \vee Finished(T_j, r_k) \right) \right) && \text{Def. of } \\
& && DRight_{RP} \\
& \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(p_j > p_i \wedge r_k \in R_j \right. \\
& \quad \left. \Rightarrow NotReq(T_i, T_j, r_k) \vee Finished(T_j, r_k) \right) && TASK2 - \\
& && TS_6, \text{ Prop.} \\
& && \text{Logic} \\
& \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(p_j > p_i \wedge r_k \in R_j \right. \\
& \quad \left. \Rightarrow true \wedge [\neg T_j.requests(r_k)] \right) && ENV2, \\
& && TASK2 - \\
& && ACQ, \\
& && HOLD \\
& \Rightarrow \bigwedge_{T_j \neq T_i \in \mathcal{T}} \left(p_j > p_i \wedge r_k \in R_j \right. \\
& \quad \left. \Rightarrow true \wedge [\neg T_j.blocked(r_k)] \right) && TASK2 - \\
& && TS_6
\end{aligned}$$

□

The Reservation Protocol ensures no priority inversions.

THEOREM 4.3 $RP \wedge TASK2 \wedge ENV2 \vdash NOINV$

PROOF.

The proof follows directly from LEMMA 4.3. □

5 Conclusions

In this paper, we presented formal specifications of 5 schedulers for tasks with shared resources. With these specifications, we were able to prove properties like deadlock freedom, blocked at most once, and no priority inversion.

More importantly, we presented two frameworks by which the specifications were based on. The first framework is applicable to schedulers which makes decisions based on the current state of the system, e.g. state of the resources, etc. The second framework concerns schedulers which makes decisions based on events that happened already or future events that will take place shortly. These frameworks can be used in the future for specifying a variety of other protocols [1] such as the Ceiling Semaphore Protocol, Semaphore Control Protocol, etc.

References

- [1] Neil C.Audsley, "Resource Control for Hard Real-Time Systems: A Review" Technical Report YCS 159, Department of Computer Science, University of York, U.K., August 1991.
- [2] Ozalp Babaoglu, Keith Marzullo and Fred B.Schneider, "Priority Inversion and its Prevention in Real-Time Systems", TR-90-1088, Department of Computer Science, Cornell University, March 1990.
- [3] Ozalp Babaoglu, Keith Marzullo and Fred B.Schneider, "A Formalization of Priority Inversion", *Real-Time Systems*, **5**(4), pp. 285-303, October 1993.
- [4] Dang Van Hung and Zhou Chao Chen, "Probabilistic Duration Calculus for Continuous Time", UNU/IIST Report No. 25, May 1994. Submitted to *Formal Aspects of Computing*.
- [5] He Wei Dong and Zhou Chao Chen, "A Case Study of Optimization," UNU/IIST Report No. 34, December 1994. Submitted to *The Computer Journal*.
- [6] C.L. Liu and James W.Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, **20**(1), pp 46-61, 1973.
- [7] Liu Zhiming, Anders P.Ravn, Erling V.Sørensen and Zhou Chao Chen, "A Probabilistic Duration Calculus," In *Dependable Computing and Fault-Tolerant Systems, Vol. 7: Responsive Computer Systems*, Edited by H.Kopetz and Y.Kakuda, pp. 30-52, Springer-Verlag, 1993.
- [8] L.Sha, R.Rajkumar and J.P.Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", CMU-CS-87-181, Computer Science Department, Carnegie-Mellon University, December 1987.
- [9] L.Sha, R.Rajkumar and J.P.Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, **39**(9), pp. 1175-1185, September 1990.
- [10] William Stallings, *Operating Systems*, Macmillan Publishing Company, 1992.
- [11] Zheng Yuhua and Zhou Chao Chen, "A Formal Proof of the Deadline Driven Scheduler" UNU/IIST Research Report No. 16, February 1994. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 863*. Edited by H.Langmaack, W.-P.de Roever and J.Vytopil, pp. 756-775, Springer-Verlag, 1994.
- [12] Liu Zhiming, Mathai Joseph and Tomasz Janowski, "Verification of Schedulability for Real-Time Programs" Technical Report RR245, Department of Computer Science, University of Warwick, 1993.
- [13] Zhou Chao Chen, C.A.R.Hoare and Ander P.Ravn, "A Calculus of Durations" *Information Processing Letters*, **40**(5), pp 269-276, 1991.
- [14] Zhou Chao Chen and Li Xiao Shan, "A Mean Value Calculus of Durations" In *A Classical Mind (Essays in Honour of C.A.R. Hoare)*, Edited by A.W.Roscoe, Prentice-Hall, pp. 431-451, 1994.

- [15] Zhou Chao Chen, Dang Van Hung and Li Xiao Shan, “A Duration Calculus with Infinite Intervals, ” UNU/IIST Report No. 40, February 1995.
- [16] Zhou Chao Chen, Michael R.Hansen, Ander P.Ravn and Hans Rischel, “Duration Specifications for Shared Processors” In *Proc. of the Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, Nijmegen, January 1992. LNCS 571, pp 21-32, 1992.
- [17] Zhou Chao Chen, Ander P.Ravn and Michael R.Hansen, “An Extended Duration Calculus for Hybrid Real-Time Systems” In *Hybrid Systems*, LNCS 736, Springer-Verlag, pp 36-59, 1993.

A Proof of Lemma 4.1

We prove this LEMMA by induction: First, let :

$$\begin{aligned} \mathcal{F} &\hat{=} [\bigwedge_{T_j \neq T_i \in \mathcal{T}} \bigwedge_{r_l \in \mathcal{R}} (T_j.\text{holds}(r_l) \wedge \text{HiPri}_{PCP}(T_i, T_j) \Rightarrow p_i > c(r_l))] \\ \mathcal{H}(\mathcal{X}) &\hat{=} (\mathcal{X} \wedge [T_i.\text{in-cr}] \Rightarrow \mathcal{F}) \\ \Gamma &\hat{=} A1 \wedge PCP \wedge TASK \wedge ENV \end{aligned}$$

Base case: $\Gamma \vdash \mathcal{H}([\])$

$$\begin{aligned} [\] \wedge [T_i.\text{in-cr}] &\Rightarrow \mathcal{F} \\ &\Rightarrow \text{false} \Rightarrow \mathcal{F} && \text{ITL} \\ &\Rightarrow \text{true} && \text{ITL} \end{aligned}$$

For the inductive step, we must establish:

$$\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \vee (\mathcal{X} \frown [\text{Right}_{PCP}(T_i)]) \vee (\mathcal{X} \frown [\neg \text{Right}_{PCP}(T_i)]))$$

We now consider two cases:

1. $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\text{Right}_{PCP}(T_i)])$
2. $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\neg \text{Right}_{PCP}(T_i)])$

Case 1: $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\text{Right}_{PCP}(T_i)])$

$$\begin{aligned} &\mathcal{X} \frown [\text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}] \\ &\Rightarrow (\mathcal{X} \wedge [T_i.\text{in-cr}]) \frown ([\text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}]) && \text{Theorem 2.18} \\ &\Rightarrow \mathcal{F} \frown ([\text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}]) && \mathcal{H}(\mathcal{X}) \end{aligned}$$

$$\begin{aligned}
&\Rightarrow \mathcal{F} \frown \left[\bigwedge_{T_j \in \mathcal{T}} \bigwedge_{r_l \in \mathcal{R}} (T_j.\text{holds}(r_l) \Rightarrow p_i > c(r_l)) \right] && \text{Def. of } \text{Right}_{PCP} \\
&\Rightarrow \mathcal{F} \frown \mathcal{F} && \text{Prop. Logic} \\
&\Rightarrow \mathcal{F} && \text{Axiom 2.13}
\end{aligned}$$

Case 2: $\Gamma, \mathcal{H}(\mathcal{X}) \vdash \mathcal{H}(\mathcal{X} \frown [\neg \text{Right}_{PCP}(T_i)])$

$$\begin{aligned}
&\mathcal{X} \frown [\neg \text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}] \\
&\Rightarrow (\mathcal{X} \wedge [T_i.\text{in-cr}]) \frown ([\neg \text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}]) && \text{Theorem 2.18} \\
&\Rightarrow \mathcal{F} \frown ([\neg \text{Right}_{PCP}(T_i)] \wedge [T_i.\text{in-cr}]) && \mathcal{H}(\mathcal{X}) \\
&\Rightarrow \mathcal{F} \frown \left(\left[\bigwedge_{T_j \neq T_i \in \mathcal{T}} \wedge r_k \in \mathcal{R}_i (\text{HiPri}_{PCP}(T_i, T_j) \Rightarrow p_j \leq c(r_k)) \right] \right) && \text{Def. of } \text{Right}_{PCP}, c(r) \\
&\Rightarrow \mathcal{F} \frown \left(\left[\bigwedge_{T_j \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_i, T_j) \Rightarrow \neg \text{Right}_{PCP}(T_j)) \right] \right) && \text{PCP, Def. of } \text{Right}_{PCP}
\end{aligned}$$

It is easy to prove that

$$\begin{aligned}
&\bigwedge_{r \in \mathcal{R}} \bigwedge_{T_j \in \mathcal{T}} \text{true} \frown [\neg \text{Right}_{PCP}(T_j) \wedge T_j.\text{holds}(r)] \\
&\Rightarrow \text{true} \frown [T_j.\text{holds}(r)] \frown [\neg \text{Right}_{PCP}(T_j) \wedge T_j.\text{holds}(r)]
\end{aligned}$$

Thus,

$$[T_j.\text{holds}(r) \Rightarrow p_i \geq c_r] \frown [\neg \text{Right}_{PCP}(T_j)] \Rightarrow [T_j.\text{holds}(r) \Rightarrow p_i \geq c_r]$$

Hence,

$$\begin{aligned}
&\mathcal{F} \frown \left(\left[\bigwedge_{T_j \neq T_i \in \mathcal{T}} (\text{HiPri}_{PCP}(T_i, T_j) \Rightarrow \neg \text{Right}_{PCP}(T_j)) \right] \right) \\
&\Rightarrow \mathcal{F} \frown \mathcal{F}
\end{aligned}$$

This completes the proof of the lemma. □