Static Priority Schedulability Analysis of Graph-Based Real-Time Task Models with Resource Sharing

Zhuo Yuzhen
Abstract

Static Priority Schedulability Analysis of Graph-Based Real-Time Task Models with Resource Sharing

Zhuo Yuzhen

The correctness of real-time systems does not only depend on the validity of the output, but also the temporal validity. Tasks are typically designed with strict deadlines and they need to respond in time, which are the timing constraints of real-time systems. Schedulability analysis is one of the approaches to study the workload of the task system.

DRTRS (Digraph Real-Time task model with resource sharing) is introduced to describe the system task model, abstracting away most functional behaviour and focus on the timing properties. We have also developed an efficient schedulability analysis under different resource access protocols.
## Contents

1 Introduction ......................................................... 4  
   1.1 Background .................................................. 4  
   1.2 Contributions ............................................... 5  

2 Task Model ......................................................... 6  
   2.1 DRT Task Model with Resource Sharing .................... 6  
   2.2 Schedulability and Feasibility .............................. 8  

3 Resource Access Protocol ........................................ 9  
   3.1 Priority Inversion ........................................... 9  
   3.2 Resource Access Protocols ................................ 10  
      3.2.1 Non-preemptive Protocol .............................. 10  
      3.2.2 Immediate Priority Ceiling Protocol ................ 11  
      3.2.3 Priority Inheritance Protocol ....................... 11  
      3.2.4 Priority Ceiling Protocol ........................... 12  
   3.3 Summary ...................................................... 14  

4 Analysis ............................................................ 15  
   4.1 Overview ..................................................... 15  
   4.2 Preemption Function ........................................ 21  
   4.3 Blocking Function ........................................... 22  
      4.3.1 NPP .................................................... 22  
      4.3.2 HLP .................................................... 23  
      4.3.3 PCP .................................................... 24  
      4.3.4 PIP .................................................... 29  
   4.4 Full Algorithm ................................................ 29  

5 Conclusions and Future Work ..................................... 31
Chapter 1

Introduction

1.1 Background

The correctness of real-time systems does not only depend on the validity of the output, but also the temporal validity. Tasks are typically designed with strict deadlines and they need to respond in time, which are the timing constraints of real-time systems. If tasks miss deadlines, it could finally lead to a system failure or even catastrophe. Therefore, real-time systems have to be designed to guarantee that critical timing constraints will never be violated. One of the approaches is schedulability analysis. It studies the task workload together with a scheduler.

Task models are designed to describe the workload. A task model abstracts away most functional behavior and focuses more on the timing properties. The classical periodic task model was introduced in the early 1970s [2]. However, it is too restrictive to describe non-periodic tasks. A recently proposed expressive task model is the Digraph Real-Time task model by Stigge et al. [6]. It abstracts tasks with directed graphs. Meanwhile, Stigge and Yi [5] have also published an efficient analysis method for static priority schedulers, which shows promising performance.

Nevertheless, tasks are typically not independent. They usually cooperate through or compete for several resources. To keep synchronized, most operating systems provide semaphores to guard shared resources. Shared resources could cause a problem called priority inversion, where lower-priority tasks can block higher-priority tasks. It violates the fundamental rule of a static priority scheduler that always the highest priority task to be executed. To avoid the unbounded priority inversion problem, various resource access protocols have been proposed.

This project is to develop an efficient schedulability analysis under different resource access protocols. The DRT task model is extended to include the notion of resource sharing for better expressiveness.
1.2 Contributions

In this report, the schedulability of a dependent task set is carefully studied under different resource access protocols. The analysis is based on a key assumption that a job cannot end with a critical section. If this is the case, the analysis can be broken down into schedulability checking of each single job. In particular, the major contributions are as follows:

1. Based on the DRT task model, a new task model is developed to efficiently describe a dependent task set. It is called DRTRS (DRT task model with Resource Sharing).

2. Detailed schedulability analysis for protocols NPP, HLP and PCP is given. Particularly, we introduce blocking functions to abstract the interference time caused by lower-priority tasks. Further, we give the calculation of blocking functions when different protocols are applied, providing the possibility of schedulability analysis for DRTRS.
Chapter 2

Task Model

2.1 DRT Task Model with Resource Sharing

The classical periodic task model proposed by Liu and Layland [2] describes system workload as a collection of independent tasks. Each task is activated periodically and the run-time of each activation is bounded by the worst-case execution time. The model assumes that each task is a periodic task, which is often too simple. Modern models have been proposed to be more expressive to enable modeling of a system’s behavior as precisely as possible. Stigge et al. [6] introduced a new graph-based task model called the Digraph Real-Time (DRT) task model. It is based on arbitrary directed graphs for job releases. Vertices represent computational jobs with execution time bounds and deadline information.

Here we introduce shared resources into the the DRT task model and we call the new task model as DRTRS (Digraph Real-Time task model with Resource Sharing). A DRTRS task set \( \tau \) consists of \( N \) tasks \( \{T_1, ..., T_N\} \). They cooperate through \( M \) resources \( \{R_1, ..., R_M\} \) and each resource \( R_k \) is guarded by a distinct binary semaphore \( S_k \). Hence, all the critical sections on resource \( R_k \) begin with a lock operation \( P(S_k) \) and end with a release operation \( V(S_k) \). A DRTRS task \( T_i \) is characterized by a directed graph \( G(T_i) \). The vertices \( \{v_1, ..., v_n\} \) in \( G(T_i) \) represents the different job types that can be released by task \( T_i \). Each job type is labeled with an ordered pair \( (e, d, [optional]) \) denoting worst-case execution time demand \( e \) and relative deadline \( d \) of the corresponding job. Both values are assumed to be non-negative integers. If job type \( v_j \in T_i \) requires resources, it is specified in the optional field. The optional field is a list of pairs \( (S_k, Z_{i,j}^k) \) describing the semaphores, where \( S_k \) is the binary semaphore which guards the corresponding resource \( R_k \) and \( Z_{i,j}^k \) is the duration for how long \( v_j \in T_i \) needs to lock the semaphore \( S_k \) at most. The corresponding critical section is denoted as \( \delta_{i,j}^k \). Here we assume each job could lock the same semaphore at most once. In Figure 2.1, \( v_3 \) holds \( S_1 \) for 1 time unit and \( v_2 \) needs \( S_1 \) and \( S_2 \), both for 2 time units.
The edges in $G(T_i)$ represent the order in which job types generated by $T_i$ are released. Each edge $(u, v)$ is labeled with a non-negative integer $p(u, v)$ denoting the minimum job inter-release separation time. To avoid intra-task interference, we assume deadlines to be constrained to the inter-release separation period. In other words, for each job type $v$, its relative deadline $d$ is bounded by the minimal $p(u, v)$ for all outgoing edges $(u, v)$.

**Example 2.1.1.** Figure 2.1 illustrates an example of a DRTRS task.

![Figure 2.1](image)

Figure 2.1: An example of DRTRS task model

The semantics of a DRTRS task system is defined as a set of job sequences. Tasks could start from an arbitrary vertex in the digraph. It goes through the graph along the directed edges and another job is released when a new vertex is visited. The job releases are constrained by inter-release separation times specified by the edge labels.

A job is an abstraction of a sequential piece of code executed on the processor [4]. What we are interested in is its timing behavior in this paper. Formally, we use a basic $3$-tuple $(R, e, v)$ to denote a job $J \in T$ that is released at absolute time $R$, with execution time $e$ and $v$ is the job type which allows us to recognize the job type from the digraph. Further, the actual length of a critical section $z^k$ needs to be specified when the job uses resources. We assume dense time, i.e. $R, e, z^k \in \mathbb{R}_{\geq 0}$. A job sequence $\rho = [(R_1, e_1, \pi_1, [z^k_1]), (R_2, e_2, \pi_2, [z^k_2]), ...]$ is generated by $T$, if and only if there is a (potentially infinite) path $\pi = (\pi_1, \pi_2, ...)$ in $G(T)$ satisfying for all $i, k$:

1. $e_i \leq e(\pi_i)$,
2. $z^k_i \leq Z^k(\pi_i)$
3. $R_{i+1} - R_i \geq p(\pi_i, \pi_{i+1})$

For a task set $\tau$, a job sequence $\rho$ is generated by $\tau$, and it is a composition of sequences $\{\rho_t\}_{T_t \in \tau}$, which are individually generated by the tasks $T_t$ of $\tau$. 

7
2.2 Schedulability and Feasibility

In this report, we focus on static priority scheduling. Since resource access protocols modify the task priority, each task is characterized by the fixed nominal priority \( P_i = \text{prio}(T_i) \) (the static given priority order, the larger \( \text{prio}(T) \), the higher priority) and an active priority \( p_i \) which is dynamic and initially set to \( P_i \). For simplicity, higher-priority and lower-priority task will always stand for higher-nominal-priority task and lower-nominal-priority task respectively. Our work is based on the following assumption:

1. Tasks \( \{T_1, T_2, ..., T_N\} \) are assumed to have unique priorities and are listed in descending order of nominal priority, i.e., \( T_1 \) has the highest nominal priority.

2. Critical Sections are guarded by binary semaphores, meaning that only one task at a time can be within a critical section.

3. Jobs can enter critical sections at any time of the job except that it cannot end with a critical section. All the critical sections used by any task are properly nested.

4. Tasks do not suspend themselves on I/O operations or on explicit synchronization primitives (except on locked semaphores).

As is known, the correctness of a real-time system does not only depend on the output validity, but also satisfy all the timing constraints. Therefore, we may want to know the temporal validity of a given real-time task set. Will all the tasks always meet their deadlines and which scheduler can schedule such a task set? Respectively, the terms feasibility and schedulability are introduced.

**Definition 2.2.1.** (Schedulability) A task set \( \tau \) is schedulable with scheduler \( \text{Sch} \), if and only if for all job sequences generated by \( \tau \), all jobs meet their deadlines when scheduled with \( \text{Sch} \). Otherwise, \( \tau \) is unschedulable with \( \text{Sch} \).

**Definition 2.2.2.** (Feasibility) A task set \( \tau \) is feasible, if and only if there is a scheduler \( \text{Sch} \) such that \( \tau \) is schedulable with \( \text{Sch} \).

The difference between schedulability and feasibility is that schedulability fixes a specified scheduler while feasibility need find a scheduler which makes the task set schedulable. We could also extend the notion of schedulability and feasibility to a single task or job.

As mentioned at the beginning of this section, static priorities are assigned to each task, which means the scheduler is fixed. Hence, we are going to study the schedulability of the task set in this paper. According to Definition 2.2.1, if we could find a counter example where one or more jobs miss their deadlines, we can conclude that the task set is unschedulable with the static priority scheduler. Otherwise, it’s schedulable with the static priority and further more, it’s feasible for sure.
Chapter 3

Resource Access Protocol

3.1 Priority Inversion

Most Real Time Operating Systems (RTOSes) employ a priority-based preemptive scheduler. Given a priority order $\text{prio} : \tau \to \mathbb{N}$, which assigns a unique priority to each task. Such a static priority (SP) scheduler will always pick the highest priority task from the ready task pool for execution. Thus, a lower priority task could be preempted in mid-execution by a higher priority task.

Unfortunately, it’s not always true when tasks share resources. Higher-priority tasks could be blocked by lower-priority tasks when it should be executed. The term “priority inversion” is introduced to describe such a scenario in which the highest-priority ready task fails to run when it should.

Figure 3.1 illustrates an example of priority inversion. Three tasks are drawn in the figure with high($H$), medium($M$) and low($L$) priority. Task $H$ and $L$ share resource $S$. Task $H$ is blocked when it tries to lock resource $S$ because task $L$ has already occupied the resource. Shortly after that, task $M$ is released. Considering task $H$ is blocked and $M$ has a higher priority than $L$, the scheduler switches to execute task $M$. While $M$ runs, task $H$, the highest-priority task in the system, remains in a pending state. In other words, task $M$ preempts
This violates the static priority model that tasks can only be prevented from running by higher priority tasks. What’s worse, as long as there are jobs remaining with a higher priority than $L$, $H$ could be blocked for an indefinite period of time.

In some cases, priority inversion can occur without causing immediate harm. The delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high priority task is left starved of the resources, it would finally misses its deadline and lead to a system failure. The trouble experienced by the Mars lander "Mars Pathfinder" is a classic example of problems caused by priority inversion in real-time systems.

### 3.2 Resource Access Protocols

Several resource access protocols have been proposed to solve the un-bounded priority inversion problem. Tasks are forced to follow pre-defined rules when requesting and releasing resources. All the approaches developed in the context of fixed priority scheduling consist of raising the priority of a task when accessing a shared resource [1], so that always the “highest priority” task from the ready task pool will be executed.

#### 3.2.1 Non-preemptive Protocol

*Non-preemptive Protocol (NPP)* is the simplest solution that avoids the unbounded priority inversion problem. The main idea is to disable preemption during the execution of any critical sections. At run-time, it could also be implemented by raising the priority of the task to the highest level whenever it succeeds in locking a shared resource. The task is then reset to the nominal priority once it releases the resource.

Figure 3.2 illustrates how NPP solves the priority inversion phenomenon. However, it is only appropriate when tasks use short critical sections considering the unnecessary blocks that NPP could cause. Consider the case depicted in Figure 3.2, task $H$ is blocked for a long period of time when $L$ enters a long critical section even if $H$ does not use $S$ at all.

![Figure 3.2: An example of Non-preemptive Protocol](image-url)
According to the protocol, a task cannot be preempted when it enters a critical section. In other words, it can block any task which arrives before it releases the resource. Therefore, a higher-priority task could be blocked by any critical section which belongs to a lower-priority task.

Moreover, preemption is disabled inside any critical section. Thus, only one task could enter a critical section at any time.

### 3.2.2 Immediate Priority Ceiling Protocol

Immediate Priority Ceiling Protocol improves NPP. The task which enters a critical section is raised to the highest priority among the tasks sharing that resource, rather than raised to the highest level of the system. For this reason, this protocol is also referred to as Highest Locker Priority (HLP).

The main idea is to define the ceiling $C(S)$ of a semaphore $S$ to be the highest priority of all tasks that could use $S$ during execution. Thus, $C(S)$ can be calculated offline for a given task set. Whenever a task succeeds in holding a resource $S$, its priority assumes the maximum of its current priority and $C(S)$. When it releases the resource, the priority will change back to what it was before the task entered the critical section.

![Figure 3.3: An example of Highest Locker Priority](image)

Figure 3.3 gives an example of HLP scheduling. The task $L$ is assumed to have medium priority when it enters the critical section. Hence, it could be preempted by task $H$ at $t = 6$ and block task $M$ at $t = 12$.

### 3.2.3 Priority Inheritance Protocol

Sha et al. [3] proposed the Priority Inheritance Protocol (PIP) to avoid unbounded priority inversion by modifying the priority of those tasks that cause blocking. The main idea of PIP is that whenever a lower-priority task blocks a higher priority task, it inherits the priority of the blocked task. When a task blocks one or more tasks, it would always assume the highest priority of the blocked tasks. It resets to its nominal priority when it releases the resource.
Figure 3.4 gives an example of a PIP schedule. From this example, we notice that a higher-priority task can experience two kinds of blocks (Buttazzo [1]):

- **Direct Blocking.** It occurs when a higher-priority task tries to acquire a resource already held by a lower-priority task. Direct blocking is necessary to ensure the consistency of the shared resources.

- **Push-through blocking.** It occurs when a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks. Push-through blocking is necessary to avoid unbounded priority inversion.

Though PIP can solve the priority inversion problem, a task can still suffer from substantial blocks [3]. Consider the example as shown in Figure 3.4. In this scenario, when attempting to use its resources, task $H$ is blocked for the duration of two critical sections, once to wait for task $L$ to release $S_2$ and then to wait for task $M$ to release $S_1$. This is called chained blocking. If a task shares $n$ distinct resources with lower-priority tasks, in the worst case, it could be blocked for $n$ times.

Another problem is that the task set can suffer from deadlocks under PIP. Consider a scenario where two tasks use two semaphores in a nested fashion but in reverse order, as illustrated in Figure 3.5. Deadlock happens when task $L$ attempts to lock $S_1$. However, the deadlock is not caused by the Priority Inheritance Protocol itself, but an erroneous use of semaphores. It can be solved by locking semaphores in an ascending order.

### 3.2.4 Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) was also introduced by Sha et al. [3]. PCP combines the idea of HLP and PIP so that PCP can avoid unnecessary blocking while solves the deadlock and chained blocking problem as well [1].

Similarly as HLP, each semaphore is assigned a ceiling to be the highest priority among the tasks that share that resource. What’s more, the term
system ceiling is introduced under PCP. When task $T$ tries to enter a critical section, assume $S$ to be the semaphore with the highest ceiling among all the semaphores which are locked by other tasks (exclude task $T$ itself). We define $C(S)$ as the current system ceiling $C_{sys}$. Task $T$ is allowed to enter a critical section if and only if its priority is strictly higher than the current system ceiling $C_{sys}$.

If the resource request is denied, $T$ will be blocked and it transmits its priority to the holder of semaphore $S$, say $T'$. Then, $T'$ executes the rest of its critical section with $\text{prio}(T)$ until it releases the resource or it locks another task which has a higher-priority than $T$. Task $T'$ is said to inherit the priority of $T$. When $T'$ exits a critical section, it unlocks the semaphore and among the tasks, if any, which is blocked by the semaphore, only the task with the highest active priority will be awakened. If there is no other task blocked by $T'$, its active priority will be reset to its nominal priority. Otherwise, $T'$ will be assumed the highest priority of the tasks which are blocked by $T'$.

If task $T$ succeeds in locking the semaphore, it keeps running with the same priority so that PCP can avoid unnecessary blocking.

Figure 3.6 illustrates an example of Priority Ceiling Protocol from Buttazzo [1]. Task $T_1$, $T_2$ and $T_3$ communicate with each other through three semaphores $S_1$, $S_2$ and $S_3$. $T_1$ has the highest nominal priority, whereas, $T_3$ has the lowest. Task $T_1$ will sequentially lock semaphore $S_1$ and $S_2$. $T_2$ will just request semaphore $S_3$ while $T_3$ uses semaphore $S_3$ first and then it makes a nested access to $S_2$. According to the resource requirements, the semaphore ceilings can be given as follows: $C(S_1) = \text{prio}(T_1), C(S_2) = \text{prio}(T_1), C(S_3) = \text{prio}(T_2)$.

In this example, $T_2$ arrives at $t = 6$ when $T_3$ has already locked semaphore $S_3$. Therefore, $T_2$ will be blocked when it tries to lock semaphore $S_3$ at $t = 8$. This is defined as direct blocking, the same as defined under PIP. When $T_1$ tries to lock $S_1$, the request is also denied due to the system ceiling. At $t = 18$, $S_2$ and $S_3$ are locked. Hence, $C_{sys} = \text{max}\{C(S_2), C(S_3)\} = \text{prio}(T_1)$, which is equal to the priority of $T_1$. Therefore, $T_1$ is blocked though $S_1$ is not used by anyone. This is called ceiling blocking. It is necessary to avoid deadlock and chained blocking [1].

As we can see from this example, a job can only be blocked when it makes its first resource request. This is always true under PCP. When a task is blocked, the semaphore holder will inherit its priority.
3.3 Summary

To sum up, resource sharing could cause priority inversion, where a high priority task is blocked by lower-priority tasks. To solve the priority inversion phenomenon, resource access protocols are proposed. Besides the nominal priority, tasks are also tagged with active priorities. All the protocols consist of raising the active priority when it enters critical sections, so that always the “highest active priority” task from the ready pool is to be executed.

Different protocols have their own pros and cons. NPP is the simplest approach to realize the idea. However, it introduces lots of unnecessary blocking. Although HLP improves the NPP a little bit, tasks still suffer from unnecessary blocking more or less. PIP was introduced to avoid unnecessary blocking. Unfortunately, chained blocking and potential deadlocks come up alongside with the protocol. PCP is a more advanced protocol which can avoid the unnecessary blocking, chained blocking and deadlocks. Nevertheless, it’s difficult to implement.
Chapter 4

Analysis

4.1 Overview

Stigge and Yi [5] introduce a schedulability checking procedure based on the lowest-priority feasibility:

\textbf{Lemma 4.1.1.} (Lowest-priority Feasibility) For a task set $\tau = \{T_1, ..., T_N\}$, a task $T \in \tau$ is lowest-priority feasible in $\tau$ if there is a priority order $\text{prio}$ with $T$ as the lowest-priority task such that $T$ does not miss any deadlines if $\tau$ is SP scheduled with $P$.

However, the procedure does not work when there are resources shared by different tasks. In [5], tasks are assumed to be independent so that adding tasks of lower priority to a task set does neither introduce nor remove deadline misses of higher priority tasks. When there are communication and synchronization between tasks, the interference time for a task $T$ could result from two parts: the preemption from higher-priority tasks and blocking from lower-priority tasks. Taking the blocking time into consideration, the $SP$ schedulability of $\tau$ with a priority order $P$ could be decided as follows:

1. Create an empty task set $\tau' = \emptyset$;
2. Pick the lowest priority task $T_k$ from the task set $\tau$;
3. Check whether $T_k$ is lowest-priority feasible in $\tau$, taking the blocking time from $\tau'$ into consideration;
4. Move the $T_k$ from $\tau$ to $\tau'$;
5. Go back to \textit{Step 2} and rerun the LP feasibility check until $\tau$ is an empty task set.

Based on the above procedure, we can focus on checking whether each single task is schedulable with an $SP$ scheduler.
A single task could contain various jobs while they share the same priority. The static information about a job \( J \in T \) consists of an absolute release time \( R \), a worst-case execution time \( e \) and the job type \( v \) with its relative deadline \( d \) and critical section information, as we have discussed in Section 2.1. We name the time interval \([R, R + d]\) the scheduling window of the job. If other tasks could cause more than \((d - e)\) time units of interference in the scheduling window, \( J \) would, for sure, miss its deadline. In other words, \( J \) is unschedulable. Otherwise, \( J \) is schedulable and feasible.

For an independent task set, it’s sufficient to test all jobs of a task separately in order to conclude that the task is schedulable.

**Definition 4.1.2.** For two scheduling case \( A \) and \( B \), if job \( J \in T \) will suffer more interference time in case \( A \), we say \( A \) is a worse case for job \( J \), written as \( A > B \). Otherwise, we say \( B \) is a non-better case than \( A \), written as \( B \geq A \).

**Theorem 4.1.3.** For independent tasks, one of the worst scheduling cases for job \( J \) with job type \( v \in T_i \) is that all the higher-priority tasks are released at the arrival of \( J \).

**Proof.** Say the job \( J \in T_i \) is released at \( R \) with absolute deadline \( D \). For an arbitrary scheduling case \( A \), search backward from \( t = R \). Stop searching when the CPU is idle or the priority of the running task is no higher than \( \text{prio}(T_i) \). Say the search stops at \( t = R' \).

Since it is always the highest priority task among the ready tasks to be executed, we can assert that a higher-priority task is released at \( R' \). What’s more, there can not be a job with a higher priority, which is released before \( R' \) and finished after \( R' \). Otherwise, the busy interval will start earlier than \( R' \), which contradicts with our assumption(Lemma 4.1.4).

Now release the job \( J \in T_i \) at \( R' \), denoted as scheduling case \( B \). Compared with case \( A \), job \( J \) will suffer no less interference time in case \( B \). Hence, \( B \geq A \).

In case \( B \), all the higher-priority jobs are released no earlier than \( R' \). However, there might be some higher-priority tasks which are released later than \( R' \). If all the higher-priority tasks arrive at \( R' \), say case \( C \), it will cause as much interference as, if not more than, case \( B \). Therefore, case \( C \geq B \). Considering that \( B \geq A \), \( C \geq A \) as well.

To conclude, for an arbitrary scheduling case \( A \), we can always find a no-better case \( C \), where all the higher-priority tasks are released at the arrival
of \( J \). Thus, we can say it is one of the worst scheduling cases that all the higher-priority tasks are released at the arrival of \( J \). The theorem follows.

**Lemma 4.1.4.** If a job \( J \in T \) with nominal priority \( P \) is released at \( R \) and finishes at \( F \), then the tasks running inbetween \([R,F]\) is always executed with an active priority no lower than \( P \).

**Proof.** Say there is a job \( J' \) runs with lower priority than \( P \) inside \([R,F]\). If this is the case, job \( J \) shall be able to preempt \( J' \) with a \( SP \) scheduler. Thus \( J \) can finish earlier than \( F \), which contradicts with the assumption. The lemma follows.

When it comes to dependent tasks, the situation becomes more complex. Lower-priority tasks could cause interference time as well due to communication and synchronization between tasks. What’s worse, the previous jobs could also cause interference indirectly. Example 4.1.5 is a nice example where \( u_2 \in T_2 \) will miss its deadline due to the interference caused by \( u_1 \in T_2 \). Worth mentioning, the problem remains under NPP, HLP, PIP and PCP.

**Figure 4.2:** Previous jobs of the same task can cause interference

**Example 4.1.5.** A task set \( \tau = \{T_1, T_2\} \) synchronize through a semaphore \( S \), as illustrated in Figure 4.2(a).

All the job types can meet the timing constraints if we just focus on their own scheduling windows. However, Figure 4.2(b) gives a scheduling case where \( u_2 \in T_2 \) misses its deadline. This is due to the indirect interference caused by vertex \( u_1 \in T_2 \).

Fortunately, if we agree that jobs cannot end with a critical section, the problem can be avoided. We will have the following lemma.
Figure 4.3: Examples of busy interval search under different protocols.
Lemma 4.1.6. For a job $J \in T_i$ from a dependent task set $\tau$, where all jobs do not end in critical section, one of the worst scheduling cases is that all the higher-priority tasks release the job sequences at the arrival of $J$ and one or several, if possible, jobs belonging to lower-priority tasks just enter critical sections.

Proof. The fundamental assumption for this to hold is that all the other job types in $T_i$ are schedulable with $SP$ scheduler. Once one of the other jobs are unschedulable, the entire task set is unschdulable with $SP$ scheduler and there is no need to analysis the schedulability of job $J$. Hence, this shall be a proper assumption.

Say $T_i$ is a task from the task set $\tau = \{T_1, ..., T_N\}$. Job $J \in T_i$ with nominal priority $P_i = prio(T_i)$ and it’s released at $R$ while the absolute deadline is $D$.

For an arbitrary scheduling case $A$, search backward from $t = R$. Stop searching when the CPU is idle or the active priority of the running task is no higher than $prio(T_i)$. Say the search stops at $t = R'$.

Figure 4.3 illustrates search examples under different resource access protocols. Considering that the normal execution of lower-priority task can not introduce inference, only the critical section part is displayed in the figure. Worth mentioning, in Figure 4.3(a), the search stops at $t = 4$ rather than $t = 0$. Although $T_{i+1}$ will be assumed a higher active priority once it enters the critical section, the lock semaphore operation itself is done with a low priority. That’s why the search shall stop at $t = 4$, when $T_{i+1}$ tries to lock the resource. This shall also work under $PIP$ and $PCP$.

Now, let’s focus on the busy interval. There could be two possibilities in the interval $[R', R]$:

1. There is no critical section belonging to previous jobs of the same task

Now, what we want to do is to "move" the higher-priority job sequences into the scheduling window of $J$.

The first step is to remove the prefix of higher-priority job sequences which are released before $R'$. Based on Lemma 4.1.4, higher-priority jobs which are released before $R'$ won’t affects the job sequence after $R'$ directly. However, it can delay the job sequence through preempting the lower priority tasks. Figure 4.3(a) is a nice example. Based on this consideration, we would "delay" the release of lower-priority jobs so that the critical section which appears in the busy interval will be locked by the holder just before $R'$. Name the generated scheduling case as case $B$. $B$ shall cause as much interference as, if not more than, case $A$, which is an arbitrary case. Hence, $B \geq A$

The second step is to release the job $J$ at $R'$, denoted as case $C$, so that all the higher-priority job sequence arrives no earlier than $R'$. Since $[R', R]$ is a busy interval with higher active priority than $J$, those tasks can preempt/block job $J$. Therefore, case $C$ shall be no better than case $B$, $C \geq B \geq A$. 

19
In a word, for an arbitrary scheduling case $A$, we can always find case $C$ where all the higher-priority job sequence is released after the arrival of $J$ and $C \geq A$.

2. **There is a critical section belonging to previous jobs of the same task**

Say the critical section belongs to $J_{\text{pre}}$, which is also generated by $T_i$ and is just before $J_{\text{pre}}$ in the job sequence. As assumed, jobs can only end with normal execution rather than critical sections. However, $[R', R]$ is a higher-active-priority interval, which means $J_{\text{pre}}$ did not finish before $R$.

On the other hand, relative deadlines are constrained to the inter-release separation time, which means $J_{\text{pre}}$ shall finish before $R$. Therefore, $J_{\text{pre}}$ is infeasible in this case, which contradicts with the assumption.

In summary of the two different possibilities, we could draw the following conclusion: we can focus on the scheduling window of single job types, and if they are all schedulable, the task is schedulable. Otherwise, the task is infeasible.

Based on Lemma 4.1.6, checking the schedulability of task $T_i$ could be broken down into checking the schedulability of each single job type that could be released by $T_i$. Further on, a worst scheduling case for $J \in T_i$ is that all the higher-priority job sequences arrives together with $J$ and jobs are released as soon as the inter-release separation time. Theorem 4.1.7 gives a sufficient and necessary condition for a single job to be feasible.

**Theorem 4.1.7.** For a job $J = \langle e, d \rangle$, it is feasible when the given task set can not cause more than $(d - e)$ time units of interference in any time window of $d$ time units. Otherwise, it’s infeasible.

However, we don’t need to check the entire scheduling window all the time. In fact, if there is a time point $t \leq d$, where the interference is no more than $(t - e)$, then $J$ is schedulable. Denote $I(t)$ as the accumulated interference function of $t$. To simplify, the absolute release $R$ is denoted as the zero-point of the timing axis.

**Lemma 4.1.8.** A job type $J = \langle e, d \rangle$ is schedulable with interference set $\tau$ if and only if for all interference function $I(t)$:

$$\exists t \leq d : e + I(t) \leq t$$

(4.1)

**Proof.** Assume that condition 4.1 holds, say at $t = t_0, e + I(t_0) \leq t_0$. So $I(t_0) \leq t_0 - e$. Considering the fact that there can not be more than $(d - t_0)$ interference in interval $[t_0, d]$, the maximum interference in the scheduling window shall be constrained to $(t_0 - e) + (d - e)$. Therefore, $I(d) \leq (d - e)$, which means $J$ is schedulable according to Theorem 4.1.7.

Now assume that $J$ is schedulable with the interference set $\tau$, which means $J$ can always meet its deadline, say it finishes at $F \leq d$. Then we can always find
Here, the main problem that needs to be solved is how to compute the interference function. As discussed previously, the interference in a DRTRS task system could be divided into two parts, the preemption \( P(t) \) of high-priority tasks and blocking time \( B(t) \) caused by lower-priority tasks.

\[
I(t) = P(t) + B(t) \tag{4.2}
\]

Hence, we will discuss the computation of the preemption and blocking time in the following sections. Note, different sets of \( P(t) \) and \( B(t) \) are possible. The interference which we are interested in is a combinational problem.

### 4.2 Preemption Function

A key assumption for us is that jobs cannot end with a critical section. In other words, a job always ends with normal execution and the active priority is equal to its own nominal priority. Therefore, for a job \( J \in T_i \), higher-priority jobs, which are released before \( J \) finishes, can always preempt \( J \).

A naïve approach to calculate the preemption time is as follows. For each task \( T_i \in \tau \), list all the potential paths \( \Pi(T_i) \) and pick one \( \pi_k \). Release the first job at time 0 and all the following jobs as soon as the inter-release separation times. Calculate the maximum accumulated execution time.

Request functions [5] is introduced to abstract a path \( \pi \) and for each \( t \), they return the accumulated execution time of all the potential jobs that \( \pi \) may release during the first \( t \) time units.

**Definition 4.2.1.** (Sigge and Yi [5]) For a path \( \pi = (v_0, v_1, ..., v_l) \) through the digraph \( G(T) \) of task \( T \), the request function is defined as:

\[
rf_\pi(t) := \max\{e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } p(\pi') < t\} \tag{4.3}
\]

where \( e(\pi) := \sum_{i=0}^{l} e(v_i) \) and \( p(\pi) := \sum_{i=0}^{l-1} p(v_i, v_{i+1}) \)

In particular, \( rf_\pi(0) = 0 \) and \( rf_\pi(1) = e(v_0) \), assuming that all edge labels are strictly positive [5]. Considering a path \( \pi = (v_1, v_3, v_2) \) in Example 2.1.1, we could abstract \( \pi \) with the following request function in Figure 4.4.

With this path abstraction, we can give a precise preemption calculation of a vertex \( v \in T \). Consider all combinations of request functions in all tasks of higher-priority. Write \( \Pi(T) \) for the path set of \( G(T) \) and \( \Pi(\tau) \) for the set of all combinations paths from all higher-priority tasks. Further, let \( \hat{\pi} = (\pi(T_1), ..., \pi(T_N)) \) denote an element of \( \Pi(\tau) \), i.e. a single combination of paths. Hence, for each \( \hat{\pi} \) the preemption time could be calculated as follows:

\[
P_\hat{\pi}(t) = \sum_{T_i \in \tau} rf_{\pi(T_i)}(t) \tag{4.4}
\]

21
4.3 Blocking Function

Similarly, we introduce a new function to describe the blocking time that a job can suffer:

**Definition 4.3.1.** In the scheduling window of a job \( J = (e, d) \), if it is blocked by a set of critical sections \( \{\delta_1, \delta_2, \ldots\} \) sequentially at \( \{t_1, t_2, \ldots\} \), then the blocking function is:

\[
B(t) = \sum z_i \text{ when } t \in [t_i, t_{i+1}] \text{ and } t \leq d. \tag{4.5}
\]

where \( z_0 = t_0 = 0 \) and \( z_i \) is the length of the corresponding critical section.

Further on, we name \( \{(t_1, \delta_1), (t_2, \delta_2), \ldots\} \) the blocking pairs.

As we can see from Equation 4.5, the blocking function is decided by the blocking pairs. A straightforward idea to work out the blocking function is to find out all the blocking pairs. As discussed in Chapter 3, different resource access protocols will result in different blocking. Therefore, we will discuss the blocking time calculation separately.

4.3.1 NPP

**Theorem 4.3.2.** (Buttazzo [1]) When NPP is applied, a job \( J \in T_i \) could be blocked at most once, and the maximum blocking time that \( J \) might suffer is constrained to the duration of the longest critical sections belonging to a lower-priority job.

**Proof.** Preemption is disabled inside any critical section under NPP. Thus, only one task could enter critical section at any time. If there is a job in critical section at the arrival of \( J \), then \( J \) would be block until the end of the critical section.

After the release of \( J \), it’s impossible for a lower-priority job to lock another semaphore until \( J \) finishes. Therefore, \( J \) could be blocked at most once. \( \Box \)

As Theorem 4.3.2 implies, a job \( J \in T_i \) could be blocked at most once, which means there is only one blocking pair under NPP, and the blocking time
is constrained to the longest critical sections among all those potential jobs which can block $J$, which is given by

$$B_{\text{max}} = \max \{ Z^k_{j,-} \mid \text{prio}(T_j) < \text{prio}(T_i) \} \quad (4.6)$$

where $Z^k_{j,-}$ is the length of a critical section guarded by semaphore $S_k$, and belonging to $T_j$. The detailed job type is default here.

Note that dense time is assumed in this paper. To block job $J$, the lower-priority tasks need to lock the semaphore just before the arrival of $J$. Therefore, the exact blocking time shall be $(Z^k_{j,-} - \varepsilon)_{\varepsilon \to 0}$. To simplify, we take the supremum as the maximum blocking time.

Once a task enters its critical section, it will be assumed the highest priority in the system until it releases the resource. Hence, the blocking time is not affected by the path combination of higher-priority tasks. Then the blocking pair shall be $(0, B_{\text{max}})$ and the blocking function for job $J = (e, d) \in T_i$ is given as follows:

$$B(t) = \max \{ Z^k_{j,-} \mid \text{prio}(T_j) < \text{prio}(T_i) \}, \ 0 \leq t \leq d \quad (4.7)$$

The worst case is that the longest critical section among the lower-priority tasks is locked just before the release of $J$.

### 4.3.2 HLP

**Theorem 4.3.3.** (Buttazzo [1]) Under HLP, a task $T_i$ can only be blocked by critical sections belonging to lower priority tasks which has a resource ceiling higher or equal to $\text{prio}(T_i)$.

**Proof.** Say $T_j$ has a critical section which is able to block $T_i$. Since only lower-priority task can block $T_i$ (higher-priority tasks always preempt $T_i$), then $\text{prio}(T_j) < \text{prio}(T_i)$.

To block $T_i$, $T_j$ shall have no lower active priority when it enters critical section. Therefore, $C(S) \geq \text{prio}(T_i)$.

To sum up, we can conclude: $\text{prio}(T_j) < \text{prio}(T_i) \leq C(S)$.

**Theorem 4.3.4.** (Buttazzo [1]) Under HLP, a job $J \in T_i$ can only be blocked at most once.

**Proof.** Assume that $J \in T_i$ is blocked twice by two critical sections $\delta_j \in T_j$ and $\delta_k \in T_k$. According to the Theorem 4.3.3, we can get:

$$\text{prio}(T_j) < \text{prio}(T_i) \leq C(S_j), \quad (4.8a)$$

$$\text{prio}(T_k) < \text{prio}(T_i) \leq C(S_k). \quad (4.8b)$$

To block $T_i$ twice, both $T_j$ and $T_k$ shall be inside the critical section at the arrival of $T_i$. Considering the fact that they can not lock the corresponding semaphore at the same time, one shall be preempted by the other inside its critical section. Say $T_j$ is preempted by $T_k$. Then $\text{prio}(T_k) \geq C(S_j)$, which counters to Equation 4.8. Hence, the theorem follows. \qed
Based on Theorem 4.3.3 and 4.3.4, when HLP is applied, a job $J = (e, d) \in T_i$ could be blocked at most once as well. The unique blocking pair, which is $(0, B_{\text{max}})$, looks similar as the one under NPP. The only difference from NPP is that the critical section which blocks $J \in T_i$ belong to lower-priority jobs with a resource ceiling higher than or equal to $P_i$. Hence,

$$B_{\text{max}} = \max \{Z_{j-}^k \mid \text{prio}(T_j) < \text{prio}(T_i) \leq C(S_k)\} \quad (4.9)$$

To block $J = (e, d) \in T_i$, the corresponding critical section $\delta_{j, S_k}$ should start before the arrival of $J \in T_i$. As defined in HLP, the priority will be assumed $C(S_k)$ once it enters critical section. Therefore, the block function under HLP is given by $B_{\text{max}}$ as well.

$$B(t) = \max \{Z_{j-}^k \mid \text{prio}(T_j) < \text{prio}(T_i) \leq C(S_k)\} \quad 0 \leq t \leq d \quad (4.10)$$

### 4.3.3 PCP

**Maximum Blocking Time under PCP**

**Theorem 4.3.5.** (Sha et al. [3]) Under the Priority Ceiling Protocol, a job $J \in T_i$ can be blocked for at most the duration of one critical section.

According to this theorem, a job can be blocked at the most of the longest critical section among those that can block the job. The following lemma identifies all the potential critical sections which are able to block $J \in T_i$.

**Lemma 4.3.6.** (Sha et al. [3]) Under the Priority Ceiling Protocol, a critical section $\delta_{j, S_k}$ (belonging to task $T_j$ and guarded by semaphore $S_k$) can block a job $J \in T_i$ if and only if $\text{prio}(T_j) < \text{prio}(T_i) \leq C(S_k)$.

In other words, a job $J \in T_i$ can only be blocked by critical sections belonging to lower priority tasks with a resource ceiling higher than or equal to $\text{prio}(T_i)$.

Denote $\gamma_i$ as the collection of critical sections which can block $J \in T_i$:

$$\gamma_i = \{\delta_{j,} \mid \text{prio}(T_j) < \text{prio}(T_i) \leq C(S_k)\} \quad (4.11)$$

Based on the Theorem 4.3.5, the block time that $J \in T_i$ can suffer is always constrained to the duration of one critical section. Therefore, we can give the longest blocking time that $J \in T_i$ can suffer as follows:

$$B_{\text{max}} = \max \{Z_{j-}^k \mid \delta_{j-}^k \in \gamma_i\} \quad (4.12)$$

**Difficulties**

Although under PCP, jobs can suffer from no more than one block as well, the block function is not simply equal to the maximum block time. We can first have a look at the following two examples.
**Example 4.3.7.** Figure 4.5(a) gives a task set $\tau = \{T_1, T_2, T_3\}$, job $u_2 \in T_1$ and $w_1 \in T_3$ share resource $S$.

According to Equation 4.12, the maximum blocking time $v_1 \in T_2$ can suffer:

$$B_{\text{max}} = \max \{Z_{3,1}\} = 1$$

There is no blocking when $T_1$ goes through path $\pi_1 = (u_1, u_2, u_2, ...)$ while $v_1 \in T_2$ might suffer from the longest blocking time when $T_1$ goes through path $\pi_2 = (u_2, u_2, ...)$, as shown in Figure 4.5(b). Though $v_1 \in T_2$ is blocked longer under path $\pi_2$, path $\pi_1$ will cause more interference, taking preemption into consideration. Figure 4.5(c) gives the total interference under $\pi_1$ and $\pi_2$ respectively.

![Figure 4.5](image-url)

**Figure 4.5:** Different blocking function when different path combinations

**Example 4.3.8.** A task set $\tau = \{T_1, T_2, T_3, T_4\}$ cooperate with each other through two semaphores $S_1$ and $S_2$, as shown in Figure 4.6(a).

Now, assume that $\pi(T_1) = (u_1, ...)$, $\pi(T_2) = (v_1, v_2)$ and $w_1 \in T_3$ arrive at the same time. There are two different scenarios where $T_4$ locks $S_1$ and $S_2$, respectively, just before the arrival of $w_1 \in T_3$. 

25
(a) The task set

(b) Blocking function for different critical sections

(c) Interference function for different critical sections

Figure 4.6: Different blocking functions under the same higher-priority path combination
When \( T_4 \) locks \( S_1 \), the maximum blocking time is 2 time units and \( w_1 \in T_3 \) will miss its deadline in this case. In the other case, although the potential maximum blocking time is longer, \( w_1 \in T_3 \) can finish the job at \( t = 5 \). This is because the accumulated interference is 3 time units and \( e + I(t) \leq t \) holds. As Theorem 4.1.8 implies, \( w_1 \in T_3 \) is schedulable in this case.

As we can see from Example 4.3.7, different path combinations of higher-priority tasks could suffer from different blocking functions. Example 4.3.8 is another example, where jobs may go through different blocking functions even when the higher-priority job sequences are exactly the same.

**Critical Blocking Functions**

It’s essential to check the blocking functions for different higher-priority path combinations separately. However, it’s not always necessary to check all different blocking functions with the same path combination. Instead, we just need to check the set of critical blocking functions.

**Definition 4.3.9.** For two blocking functions \( bf \) and \( bf' \) belonging to the same path combination \( \hat{\pi} \) on domain \([0, d]\), we say that \( bf \) dominates \( bf' \), written as \( bf \succ bf' \) if and only if

\[
\forall t \in [0, d] : \quad bf(t) \geq bf'(t) \tag{4.13}
\]

The maximal set of blocking functions \( bf \) which contains no other \( bf' \) with \( bf' \succ bf \) is called a set of critical blocking functions.

Say \( BF(\hat{\pi}) \) is the collection of all potential blocking functions for a path combination \( \hat{\pi} = \{\pi(T_1), \pi(T_2), \ldots\} \). Let \( BF^*(\hat{\pi}) \) denote the set of critical blocking functions for \( \hat{\pi} \).

Since jobs could be blocked at most once under PCP, then the blocking function could always be described by a single blocking pair. The blocking functions in Example 4.3.8 could be described by \((0, 2)\) and \((5, 3)\). Say two blocking pairs, \((t_1, z_1)\) and \((t_2, z_2)\), can respectively describe \( bf_1 \) and \( bf_2 \), which belong to the same path combination. Then, \( bf_1 \succ bf_2 \) shall be equivalent to:

\[
t_1 \leq t_2 \text{ and } z_1 \geq z_2 \tag{4.14}
\]

**Procedure**

As we know, the element \( z_i \) is determined by the length of the critical section and \( t_i \) is decided by the higher-priority path combination. Therefore, we can first decide all potential \( t_i \) and then filter the critical sections belonging to lower-priority tasks according to Condition 4.14.

Let \( \hat{\pi} = (\pi(T_1), \ldots, \pi(T_i)) \) denote as an arbitrary path combination. For each task, search for the first job, which shares resources with lower-priority tasks, and sort these jobs in the order of release time. Say we can get a job list \( L = \{(J_1, R_1), (J_2, R_2), \ldots\} \) in timing order.
Lemma 4.3.10. Given a path combination $\hat{\pi} = (\pi(T_1), ..., \pi(T_i))$, the number of its critical blocking functions $BF^*(\hat{\pi})$ for $J \in T_i$ is constrained to $i$, which is the number of tasks with a nominal priority no lower than $T_i$.

**Proof.** Assume there are more than $i$ blocking functions in the set $BF^*(\hat{\pi})$. Then there should be at least two jobs, $J_1$ and $J_2$, which belong to the same task, in $L$. Considering the fact that the same task cannot release two jobs at the same time, we can assume $J_1$ arrives earlier. Hence, $R_1 < R_2$ and $\text{prio}(J_2) = \text{prio}(J_1)$.

For any critical section belonging to lower-priority tasks, if it can deny $J_1$’s resource request, it shall meet $C(S) \geq \text{prio}(J_1)$. So it will deny $J_1$ first. In other words, there is no chance for any critical section to block $J_2$ rather than $J_1$, which contradicts with the assumption. 

Nevertheless, not all the jobs are “valid” in the list. We can have a look at the following example.

**Example 4.3.11.** Figure 4.7 give a job list $L$ containing 5 jobs from 5 distinct tasks. Among the jobs, $J_2$ has the highest nominal priority but arrives later than $J_1$. Say a critical section $\delta_{S_j}$ is able to deny $J_2$’s resource request, then $\text{prio}(J_2) \leq C(S)$ should hold. Further on, we can get $C(S) \geq \text{prio}(J_2) \geq \text{prio}(J_1)$. In other words, $\delta_{j_{-}}$ can deny $J_1$ as well. Since $J_1$ comes earlier, the blocking time caused by $\delta_{j_{-}}$ counts at the arrival of $J_1$. Hence, $(J_2, R_2)$ can be removed from the job list. The same story for job $J_5$.

Inspired by Example 4.3.11, we can clean the job list a little bit by removing jobs which has a higher nominal priority than one or more jobs before it. Therefore, the job list in Example 4.3.11 could be reduced into three elements $\{(J_1, R_1), (J_3, R_3), (J_4, R_4)\}$, which map to three candidate blocking pairs.

Now that we have worked out all potential $t_i$ for critical blocking pairs, it’s time to calculate the corresponding blocking time. Consider the job $J_3$ in Example 4.3.11. If we want to get a blocking function that steps at $R_3$, the critical section shall block $J_3$, whereas, it cannot block $J_1$. Let $\gamma_3$ denotes the set of potential critical sections. Then it shall be:
\( \gamma_3 = \{ \delta^k_{j} \mid \text{prio}(T_j) < \text{prio}(T_i) < \text{prio}(T_3) \leq C(S_k) < \text{prio}(T_1) \} \) \hspace{1cm} (4.15)

Therefore, the corresponding blocking pair could be written as \((R_3, \max\{Z^k_j \mid \delta^k_{j} \in \gamma_3\})\). Similarly, we can give the set of critical sections corresponding to \(R_4\).

\[ \gamma_4 = \{ \delta^k_{j} \mid \text{prio}(T_j) < \text{prio}(T_i) < \text{prio}(T_4) \leq C(S_k) < \text{prio}(T_3) \} \] \hspace{1cm} (4.16)

More generally, denote \((R_m, B_m)\) as the corresponding blocking pair for \((J_m, R_m)\) in the critical job list \(L^* = \{(J_1, R_1), (J_2, R_2), \ldots\}\). Then

\[ B_m = \max\{Z^k_j \mid \delta^k_{j} \in \gamma_m\} \] \hspace{1cm} (4.17)

where \( \gamma_m = \{ \delta^k_{j} \mid \text{prio}(T_j) < \text{prio}(T_i) < \text{prio}(T_m) \leq C(S_k) < \text{prio}(T_{m-1}) \} \)

So far, we can get a set of blocking pairs \(BF = \{(R_1, B_1), (R_2, B_2), \ldots\}\). To get the set of critical blocking functions for a given path \(\hat{\pi}\), just filter \(BF\) with Condition 4.14.

### 4.3.4 PIP

Unfortunately, even without nested resources, it’s still difficult to calculate the blocking function in DRTRS task system for PIP. Instead of giving the exact computation, we will discuss the difficulties here.

As discussed in Section 3.2.3, tasks can suffer from chained blocking under PIP. For job \(J \in T_i\), if the higher-priority tasks use \(n\) distinct resources in \(J\)'s scheduling window, then \(J\) could be blocked \(n\) times at most. In other words, the blocking function under PIP is a \(n\)-step function. This can lead to an exponential explosion of the blocking combinations.

### 4.4 Full Algorithm

In Section 4.1, we have discussed how to check the schedulability of a task set based on \(LP\) feasibility and as a conclusion we can focus on single tasks schedulability checking. Further, we discussed the schedulability of vertices, as the task is schedulable when all the vertices are schedulable. Lemma 4.1.8 gives a necessary and sufficient condition when job is feasible. It’s tightly related to the interference time. As we know, a job can be preempted by higher-priority task and it can also be blocked by lower-priority task. Hence, the interference can be divided into preemption and blocking, which are discussed in Section 4.2 and 4.3 respectively.

Here, we will give the pseudo code of the full algorithm under \(PCP\). It looks similar under \(NPP\) and \(HLP\).

Assumed is three functions: \(\text{generate\_paths}(\tau)\) returns all path combinations of task set \(\tau\); \(\text{generate\_preemption}(\pi, T)\) returns the preemption function.
for task $T$ under $\pi$; and $\text{generate\_blocking}(\pi, T, \tau_{<T})$ returns all the potential blocking functions that $T$ might suffer from $\tau'$ under $\pi$. What’s more, $\tau_{>T}$ and $\tau_{<T}$ are denoting the task set of higher-priority tasks and lower-priority tasks respectively.

Algorithm 4.1 SP schedulability of a task set $\tau$ with priorities $P$

1: function SP-schedulable($\tau$)
2:     for all $T \in \tau$ do
3:         if not $\text{Schedulable}(T, \tau_{>T}, \tau_{<T})$ then
4:             return FALSE
5:     end if
6: end for
7: return TRUE
8: end function

10: function Schedulable($T, \tau_{>T}, \tau_{<T}$)
11:     $\Pi \leftarrow \text{generate\_paths}(\tau_{>T})$
12:     for all $\hat{\pi} \in \Pi$ do
13:         $P_{\hat{\pi}}(t) \leftarrow \text{generate\_preemption}(\hat{\pi}, T)$
14:         $\hat{B} \leftarrow \text{generate\_blocking}(\hat{\pi}, T, \tau_{<T})$ A group of blocking functions
15:         for all $B(t) \in \hat{B}$ do
16:             for all $v \in T$ do
17:                 if $\forall t \leq d(v): e(v) + P_{\hat{\pi}}(t) + B(t) > t$ then
18:                     return FALSE
19:                 end if
20:             end for
21:         end for
22:     end for
23: return TRUE
24: end function
Chapter 5

Conclusions and Future Work

In this report, the DRTRS task model is developed from the existing DRT task model for describing real-time tasks with resource sharing. We have also introduced efficient methods for static priority schedulability analysis when the task system applies NPP, HLP and PCP. Blocking function is introduced to efficiently abstract the interference caused by lower-priority tasks.

However, all this results work only if jobs do not use resources in the end, which obviously is not the general case. Therefore, further studies will be focusing on intra-task interference when jobs can finish with a critical section. A straightforward idea is to check the schedulability of short job sequences apart from each single job. Note that, these short job sequences do not use resources in the end so that they cannot cause interference for other jobs and job sequences belong to the same task.

Further, we did not give the blocking function under PIP due to the exponential explosion in this project. This can be another interesting topic to study in the future. Apart from the protocols we discussed in the report, there is another advanced resource access protocol, called Sharing Runtime Stack (SRS). It will also be interesting to analysis the schedulability of DRTRS task system under SRS.
Bibliography


